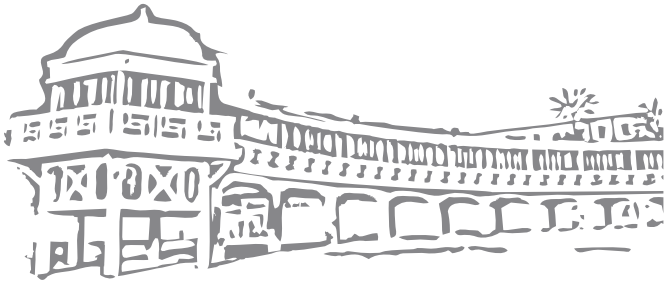


Jornadas Sistedes 2014

Cádiz, del 16 al 19 de septiembre

JISBD PROLE JCIS DC

XIV Jornadas sobre Programación y Lenguajes

ACTAS

XIV Jornadas sobre Programación y Lenguajes

Editor:

Santiago Escobar

Preface

This report contains the informal proceedings of the *XIV Jornadas sobre Programación y Lenguajes (PROLE 2014)*, held at Cádiz, Spain, during September 17th-19th, 2014. Previous editions of the workshop were held in Madrid (2013), Almería (2012), A Coruña (2011), València (2010), San Sebastián (2009), Gijón (2008), Zaragoza (2007), Sitges (2006), Granada (2005), Málaga (2004), Alicante (2003), El Escorial (2002), and Almagro (2001).

Programming languages provide a conceptual framework which is necessary for the development, analysis, optimization and understanding of programs and programming tasks. The aim of the PROLE series of conferences (PROLE stems from the spanish PROgramación y LEnguajes) is to serve as a meeting point for spanish research groups which develop their work in the area of programming and programming languages. The organization of this series of events aims at fostering the exchange of ideas, experiences and results among these groups. Promoting further collaboration is also one of the main goals of PROLE.

PROLE welcomes both theoretical and practical works concerning the specification, design, implementation, analysis, and verification of programs and programming languages. More precisely, the topics of interest include, but are not restricted to:

- Programming paradigms (concurrent, functional, imperative, logic, agent-, aspect-, object oriented, visual, ...) and their integration,
- Specification and specification languages,
- Type systems,
- Languages and techniques for new computation and programming models (DNA and quantum computing, genetic programming, ...),
- Compilation; programming language implementation (tools and techniques),
- Semantics and their application to the design, analysis, verification, and transformation of programs,
- Program analysis techniques,
- Program transformation and optimization, and
- Tools and techniques for supporting the development and connectivity of programs (modularity, generic programming, markup languages, WWW, ...).

The Program Committee of PROLE 2014 collected four reviews for each paper and held an electronic discussion during July 2014. The contributions included in this informal proceedings belong to one of the following categories:

1. Original works (3 contributions)
2. Tutorials (1 contribution)
3. Tool systems (2 contributions)
4. High-level already published papers (10 contributions).
5. Work in progress (7 contributions)

In addition to the selected contributions, the scientific program includes an invited lecture by Michael Ernst from the University of Washington, USA. We would like to thank him for having accepted our invitation.

We would also like to thank all the members of the Program Committee and all the referees for their careful work in the review and selection process. Many thanks to all authors who submitted papers and to all conference participants. Finally, we express our gratitude to all the members of the local organization of SISTEDES 2014 in Cádiz.

Valencia, Spain, Santiago Escobar
July 2014 PROLE 2014 Editor

Organization

Program Committee

Jesús Almendros	Universidad de Almería, Spain
María Alpuente	Universitat Politècnica de València, Spain
Puri Arenas	Universidad Complutense de Madrid, Spain
Manuel Carro	Universidad Politécnica de Madrid, Spain
Laura Castro	Universidade da Coruña, Spain
Francisco Durán	Universidad de Málaga, Spain
Santiago Escobar	Universitat Politècnica de València, Spain
María del Mar Gallardo	Universidad de Málaga, Spain
Raúl Gutiérrez	Universitat Politècnica de València, Spain
Lars-Ake Fredlund	Universidad Politécnica de Madrid, Spain
Salvador Lucas	Universitat Politècnica de València, Spain
Paqui Lucio	Euskal Herriko Unibertsitatea, Spain
Enrique Martín	Universidad Complutense de Madrid, Spain
Ginés Moreno	Universidad de Castilla la Mancha, Spain
Marisa Navarro	Euskal Herriko Unibertsitatea, Spain
Albert Oliveras	Universitat Politècnica de Catalunya, Spain
Fernando Orejas	Universitat Politècnica de Catalunya, Spain
Yolanda Ortega	Universidad Complutense de Madrid, Spain
Ricardo Peña	Universidad Complutense de Madrid, Spain
Adrián Riesco	Universidad Complutense de Madrid, Spain
Enric Rodríguez	Universitat Politècnica de Catalunya, Spain
Josep Silva	Universitat Politècnica de València, Spain
Alberto Verdejo	Universidad Complutense de Madrid, Spain
Alicia Villanueva	Universitat Politècnica de València, Spain

Contents

Verification games: Making software verification fun

Michael Ernst
University of Washington, USA
mernst@cs.washington.edu

Program verification is the only way to be certain that a given piece of software is free of (certain types of) errors – errors that could otherwise disrupt operations in the field. To date, formal verification has been done by specially-trained engineers. Labor costs make formal verification too costly to apply beyond small, critical software components.

Our goal is to make software verification more cost-effective by reducing the skill set required for verification and increasing the pool of people capable of performing verification. Our approach is to transform the verification task (a program and a goal property) into a visual puzzle task – a game – that gets solved by people. The solution of the puzzle is then translated back into a proof of correctness. The puzzle is engaging and intuitive enough that ordinary people can through game-play become experts. It is publicly available to play, and game players have produced proofs of security properties for real programs.

This talk will present the design goals and choices for both the game that the player sees and for the underlying program analysis. It will conclude with implications to gaming, programming, and beyond.

Submitted to:
PROLE 2014

EDD: A Declarative Debugger for Sequential Erlang Programs (High-level Work)*

Rafael Caballero Enrique Martin-Martin Adrián Riesco
Dpto. Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid
Madrid, Spain
rafa@sip.ucm.es emartinm@fdi.ucm.es ariesco@fdi.ucm.es

Salvador Tamarit
Babel Research Group
Universidad Politécnica de Madrid
Madrid, Spain
stamarit@babel.ls.fi.upm.es

Declarative debuggers are semi-automatic debugging tools that abstract the execution details to focus on the program semantics. This paper presents a tool implementing this approach for the sequential subset of Erlang, a functional language with dynamic typing and strict evaluation. Given an erroneous computation, it first detects an erroneous function (either a "named" function or a lambda-abstraction), and then continues the process to identify the fragment of the function responsible for the error. Among its features it includes support for exceptions, predefined and built-in functions, higher-order functions, and trusting and undo commands.

*Appeared in the Proceedings of the *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'14). Springer, Lecture Notes in Computer Science Volume 8413, 2014, pp 581-586.

Submitted to:
PROLE 2014

Using Big-step and Small-step Semantics to Perform Declarative Debugging (High-level Work)*

Adrián Riesco
Dpto. Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid
Madrid, Spain
ariesco@fdi.ucm.es

Declarative debugging is a semi-automatic debugging technique that abstracts the execution details to focus on results. This technique builds a debugging tree representing an incorrect computation and traverses it by asking questions to the user until the error is found. In previous works we have presented a declarative debugger for Maude specifications. Besides a programming language, Maude is a semantic framework where several other languages can be specified. However, our declarative debugger is only able to find errors in Maude specifications, so it cannot find bugs on the programs written on the languages being specified. We study in this paper how to modify our declarative debugger to find this kind of errors when defining programming languages using big-step and small-step semantics, two generic approaches that allow to specify a wide range of languages in a natural way. We obtain our debugging trees by modifying the proof trees obtained from the semantic rules. We have extended our declarative debugger to deal with this kind of debugging, and examples have been developed to test its feasibility.

*Appeared in M. Codish and E. Sumii, editors, Proceedings of the *12th International Symposium on Functional and Logic Programming* (FLOPS 2014). Lecture Notes in Computer Science Volume 8475, 2014, pp 52-68. Springer, 2014. Research supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

Submitted to:
PROLE 2014

Correctness of Incremental Model Synchronization with Triple Graph Grammars (High-level Work)

Fernando Orejas
Universitat Politècnica de Catalunya
Spain
orejas@lsi.upc.edu

Elvira Pino
Universitat Politècnica de Catalunya
Spain
pino@lsi.upc.edu

In model-driven development, we may have several models describing the same system or artifact, by providing different views on it. Then, we say that these models are consistently integrated. Similarly, we say that two models are consistent if they are complementary descriptions of some system. In this context, given two integrated models, *model synchronization* is the problem of restoring consistency when one of these models has been updated by propagating that update to the other model.

Model synchronization is studied in different areas in Computer Science. In particular, in databases (e.g., [1]), programming languages (e.g., [7]) and in model-driven software development (MDD). In the former two areas the kind of models considered are very specific, however in the latter area the kind of models considered may be very different. For this reason, in MDD we need general approaches. Triple Graph Grammars (TGGs) [9, 10] are a general (in the sense that they can be used for dealing with most kinds of models) and powerful tool to describe (bidirectional) model transformations. On the one hand, a TGG allows us to describe classes of consistently integrated models and, on the other hand, given some source model M_1, using the so-called *derived operational rules* associated to the TGG, we can find a corresponding consistent target model M_2. There are different approaches to describe model synchronization in terms of TGGs, but most of them have a computational cost that depends on the size of the given models. This may be rather inefficient since the given models may be large. To avoid this problem, Giese and Wagner [2] have advocated for the need of *incremental* synchronization procedures, meaning that their cost should depend only on the size of the given update. In particular they proposed one such procedure. Unfortunately, the correctness of this approach is not studied and, anyhow, it could only be ensured under severe restrictions on the kind of TGGs considered, since the approach only works for the case when source and target models are bijective.

In this paper we address the problem from a different point of view. First, we discuss what it means for a procedure to be incremental. Specifically, given a derivation used to create a model and an update on it, we establish what does it mean incrementality with respect to a consistent submodel not affected by the update. Essentially, it requires the existence of a derivation that builds the new model preserving that consistent submodel. Then, this idea is formulated as a correctness notion, that we call *incremental consistency*. This may be considered a first contribution of the paper.

Our second and main contribution is the introduction of a new general incremental synchronization procedure. In principle, the input for this procedure would be given by an integrated model $\overline{G}$, a derivation of $\overline{G}$ representing its structure, and an update on the source model of $\overline{G}$. However, since storing a derivation may be expensive in terms of the amount of storage needed, we replace the derivation by dependence relations on the elements of $\overline{G}$ that are shown to be equivalent, in the adequate sense for our purposes, to the derivation. Specifically, we prove a theorem that guarantees that the largest consistent submodel not affected by the update can be obtained from that dependencies without cost depending on the model. Then, the procedure consists of five steps.

1. In the first one, based on the above result, we identify the part of the model that needs to be reconstructed and we mark all the elements that may need to be deleted.
2. In the second step, if needed, we enlarge the part of the model that needs to be reconstructed. As we will discuss, this second step is only needed in some cases when the update does not

Submitted to:
PROLE 2014

allow incremental consistency with respect to the largest consistent submodel not affected by the update, but with respect to a smaller one.

3. In the third step, following the same idea presented informally in [3], we build a model that is already consistent, by applying a variation of forward translation rules [6, 4] allowing us to reuse most relevant information from the target model. For this reason, we call these rules *forward translation rules with reuse*.
4. However, the resulting model from the third step may not include elements from the target model that do not have a correspondence in the source model. To avoid this, in the fourth step we recover these elements by just using our dependence relations.
5. Finally, in the fifth step we effectively delete target elements that are still marked to be deleted.

We prove that the results of this procedure are always incrementally correct and complete in the sense that, if there is an incrementally correct solution, the procedure will find it.

We have to say that our procedure sometimes makes use of *user interaction* to take some decisions that may be not obvious, however, we want to point out that, from a theoretical point of view, this is equivalent to considering that our procedure is nondeterministic. On the other hand, it is important to say that we do not assume any restriction on the kind of grammars or graphs considered in this paper. This is not the case of most other approaches that impose reasonable restrictions to ensure efficiency. As a consequence, the implementation of our procedure may be computationally costly since, at some points some exhaustive search may be needed. However, our ideas could also be used in the context of the restrictions considered by other authors. In that case, our procedure would be as efficient (or more efficient) than these other approaches. Anyhow, it must be understood that our contribution is related to the study of when and how we can proceed incrementally in the synchronization process in the most general case, rather than restricting its application to the cases where a certain degree of efficiency is ensured.

Moreover, as we say before, there are several approaches based on TGGs that propose a solution to the model synchronization problem (that we know [2, 5, 3, 8] and some variations on them) but all of them are, in our opinion, not completely satisfactory. Let's see the cases of the most related works.

The approach in [5] has to analyze the complete graph $\overline{G}$ to know what parts must be modified, so its cost depends on the size of the given model, even if the construction of the solution does not start from scratch but from the given integrated model $\overline{G}$. In addition, in [5] not all elements of the original graph that could be reused are indeed reused. In particular, there could be parts that would have been deleted and created again. This means that, if this parts would have included some additional information, this information would have been lost. On the other hand, the only restriction considered in [5] is that the given TGG should be deterministic, to ensure that their procedure is deterministic.

In contrast, the approach in [2] does not need to analyze the complete graph $\overline{G}$ to check which parts must be modified, so its cost only depends on the size of the modification. However, their approach only works for the case when source and target models are bijective, which excludes the case where source models are views of target models (or vice versa). Moreover, rules with empty source graph, are forbidden. In addition, this approach shares with [5] the information loss problem. Finally, that approach has not been fully formalized.

The approach in [3] proposes a technique to avoid the loss of information in [2] that is essentially similar to our forward rules with reuse. Unfortunately, even if it is based on [2], it needs to analyze the complete graph $\overline{G}$ to check which parts must be modified, so its cost depends on the size of $\overline{G}$. Moreover, the approach imposes the same restrictions as [2] and lacks formality.

Finally, in [8], like us, the authors use precedence relations to avoid having to analyze the complete graph $\overline{G}$ to find which parts must be modified. However, their relation is coarser than ours. The reason is that our relations are directly based on a given derivation while in [8], their relation is based on the dependences established by the rules of the TGG. In particular, this means that two given elements of a model may be independent, but their relation may say that one depends on the

other. This has some important consequences. In particular, their synchronization procedure only works if the given triple graph is *forward precedence preserving* and if, when adding new elements, the resulting precedence graph includes no cycles. In addition, to ensure correctness, the approach also requires that the given TGG is *source-local complete*. On the other hand, the procedure needs to use a data structure that encodes how the given graph $\overline{G}$ has been derived with the given TGG. No details are given about this structure, but we suppose that it is more complex than our dependency relations. Finally, this approach also shares with [5] the information loss problem.

To conclude, in this paper we have presented a new approach for incremental model synchronization based on TGGs that has been shown to be incremental, correct and complete. Moreover, our approach is general, in the sense that we do not restrict the class of TGGs considered. As pointed out before, we do not assume any restriction on the kind of grammars or graphs, as other approaches does. On the contrary, we have focussed on the study of when and how we can proceed incrementally in the synchronization process in the most general case, rather than on finding out specific conditions and limitations on graphs and grammars that could make some techniques more efficient. As a consequence, it is difficult to provide an accurate evaluation of its performance: for some TGGs our procedure may exhibit an exponential (on the size of the updated part) behavior. But for the kind of more restricted TGGs, as the ones considered in other approaches, the behavior could be close to linear. Anyhow, what obviously remains to be done is to implement the approach and evaluate it in practice.

Acknowledgements This work has been partially supported by the CICYT project (ref. TIN2007-66523) and by the AGAUR grant to the research group ALBCOM (ref. 00516)

References

[1] Umeshwar Dayal & Philip A. Bernstein (1982): *On the Correct Translation of Update Operations on Relational Views*. *ACM Trans. Database Syst.* 7(3), pp. 381–416.

[2] Holger Giese & Robert Wagner (2009): *From model transformation to incremental bidirectional model synchronization*. *Software and System Modeling* 8(1), pp. 21–43.

[3] Joel Greenyer, Sebastian Pook & Jan Rieke (2011): *Preventing Information Loss in Incremental Model Synchronization by Reusing Elements*. In: *ECMFA 2011*, *Lecture Notes in Computer Science* 6698, Springer, pp. 144–159.

[4] Frank Hermann, Hartmut Ehrig, Ulrike Golas & Fernando Orejas (2012): *Formal Analysis of Model Transformations based on Triple Graph Grammars*. *Software and System Modeling* To appear.

[5] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin & Yingfei Xiong (2011): *Correctness of Model Synchronization Based on Triple Graph Grammars*. In: *MODELS 2011*, *Lecture Notes in Computer Science* 6981, Springer, pp. 668–682.

[6] Frank Hermann, Hartmut Ehrig, Fernando Orejas & Ulrike Golas (2010): *Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars*. In: *ICGT 2010*, *Lecture Notes in Computer Science* 6372, Springer, pp. 155–170.

[7] Martin Hofmann, Benjamin C. Pierce & Daniel Wagner (2011): *Symmetric lenses*. In: *POPL 2011*, ACM, pp. 371–384.

[8] Marius Lauder, Anthony Anjorin, Gergely Varró & Andy Schürr (2012): *Efficient Model Synchronization with Precedence Triple Graph Grammars*. In: *ICGT 2012*, *Lecture Notes in Computer Science* 7562, Springer, pp. 401–415.

[9] Andy Schürr (1994): *Specification of Graph Translators with Triple Graph Grammars*. In: *WG '94*, *Lecture Notes in Computer Science* 903, Springer, pp. 151–163.

[10] Andy Schürr & Felix Klar (2008): *15 Years of Triple Graph Grammars*. In: *ICGT 2008*, pp. 411–425.

Modular DSLs for flexible analysis: An e-Motions reimplementation of Palladio (High-level Work)

Antonio Moreno-Delgado Francisco Durán Steffen Zschaler
University of Málaga, Spain King's College London, UK
{amoreno,duran}@lcc.uma.es szschaler@acm.org

Javier Troya
Vienna University of Technology, Austria
troya@big.tuwien.ac.at

We summarize the main contributions of the work [11] presented in the *Modelling Foundations and Applications - 10th European Conference, ECMFA 2014*. In [11], we addressed some of the limitations for extending and validating implementations of Non Functional Properties (NFP) analysis tools by presenting a modular, model-based partial reimplementation of one well-known analysis framework, namely the Palladio Architecture Simulator. We specified the key DSLs from Palladio in the e-Motions system, describing the basic simulation semantics as a set of graph transformation rules. Different properties to be analyzed are then encoded as separate, parameterized DSLs, independent of the definition of Palladio. These can then be composed with the base Palladio DSL to generate specific simulation environments. Models created in the Palladio IDE can be fed directly into this simulation environment for analysis. We demonstrate two main benefits of our approach: 1) The semantics of the simulation and the non-functional properties to be analysed are made explicit in the respective DSL specifications, and 2) because of the compositional definition, we can add definitions of new non-functional properties and their analyses.

1 Introduction

It has been generally recognized that the non-functional properties (NFPs)—for example, performance or reliability—of a system are central to the success of a software development project. The later in the process an error in NFPs is discovered, the more costly will it be to repair. There is, therefore, a need for early predictive analysis of NFPs. Model-driven engineering (MDE) advocates the use of models as the primary artifacts in software development. It has been recognized that this provides opportunities for very early analysis of NFPs based on early design models. These models can often be transformed into analysis models (e.g., in the form of Petri nets or queuing networks) that can be analyzed or simulated by standard tooling [1, 2, 3, 8, 9, 10].

Typically, in these approaches a design model is translated into an analysis model which is then evaluated by a dedicated analysis tool. Alternatively, the design model is translated into a simulation of the system to be built. In both cases, however, the semantics of the non-functional property to be analyzed and of the analysis technique are only represented implicitly as encoded in the transformations or analysis tools. This causes two problems:

1. *Validation of analysis.* As there is no explicit specification of the analysis nor a high-level representation of the NFPs to be analyzed, it is difficult for users to be sure that they are analyzing

Submitted to:
XIV Jornadas sobre Programacin y Lenguajes PROLE '2014

the correct property of their system (see, e.g., [14] for a discussion of some of the subtleties that might need to be considered). Conversely, it is also very difficult for tool providers to validate the correctness of their tooling, which has a direct impact on the correctness of their predictions.

2. *Maintainability and extensibility of analyses.* The tool implementations, especially in the transformations producing simulations, often tangle code concerned with different NFPs. For example, the transformations used in the Palladio Architecture Simulator [10] tangle code for performance and reliability simulations. This makes the code very difficult to maintain and, in particular, extend to support new NFPs.

In previous work [7, 5, 12, 17], we have explored the modular definition of non-functional properties as parameterized domain specific languages (DSLs) in the e-Motions framework [13]. In [11], we demonstrate how these ideas can be integrated with predictive analysis of architectural software models by providing a modular reimplementation of a substantive part of the Palladio Architecture Simulator [10]. In particular, we have reimplemented the Palladio Component Model [3], its workload model, and parts of its stochastic expressions model. However, instead of implementing transformations to analysis models or simulators as done in Palladio, we have explicitly modeled the simulations as graph transformations in the e-Motions framework. Each NFP to be analyzed is then modeled as an independent, parameterized DSL ready to be composed with the base Palladio model. This addresses the above two problems in the following ways:

1. There is an explicit specification of both the simulation mechanism and the NFPs to be analyzed. These models can be inspected and reasoned about separately giving more assurance of correctness of the simulation results.

2. Modular definition of NFPs as separate, parameterized DSLs allows its reuse, but also makes it easy to define additional NFPs to be analyzed. For a particular analysis problem, the relevant NFP DSLs can then be selected from a library and composed as required. Our previous work in [6] provides guarantees for preservation of semantics under composition, that is, the consideration of additional NFPs (satisfying certain restrictions) do not change the behavior of the system being modeled.

2 e-Motions

e-Motions [13] is a graphical framework that supports the specification, simulation, and formal analysis of real-time systems. It provides a way to graphically specify the dynamic behavior of DSLs using their concrete syntax, making this task very intuitive. The abstract syntax of a DSL is specified as an Ecore metamodel, which defines all relevant concepts—and their relations—in the language. Its concrete syntax is given by a GCS (Graphical Concrete Syntax) model, which attaches an image to each language concept. Then, its behavior is specified with (graphical) in-place model transformations. e-Motions provides a model of time, supporting features like duration, periodicity, etc., and mechanisms to state action properties. From a DSL definition e-Motions generates an executable Maude [4] specification which can be used for simulation and analysis. Other tools in the Maude formal environment, as its model checker or its reachability analysis tool, can also be used on this specification.

3 Palladio

The Palladio Architecture Simulator [10] is a predictive software analysis tool which consists of a number of metamodels, foremost the Palladio Component Model (PCM) [3], that allow the high-level modeling of component-based architectures and their properties relevant for performance and reliability analysis. Instances of these metamodels are then transformed in preparation for analysis. Palladio supports two kinds of predictive analyses: 1) by transformation into a program that runs a simulation of the architecture's behaviour and 2) by transforming to a formalism more amenable to analysis—for example, Queuing Petri Nets [15]. Both models' semantics, and in particular of the non-functional properties being analyzed, is encapsulated in the transformations. This makes it very difficult to understand and validate these semantics. This is particularly problematic as more non-functional properties are supported: the current transformations support performance and reliability, but already are quite complex. In [11] we provided more details and a very basic example that ease the comprehension of Palladio.

4 Palladio into e-Motions

The PCM is a DSL [3], and therefore we may define it in e-Motions. As for any DSL, the definition of the PCM includes its abstract syntax, its concrete syntax and its behavior.

Since the Palladio system has been developed following MDE principles, and specifically it is implemented using the Eclipse Modeling Framework (EMF), its metamodel may be directly used as abstract syntax definition of Palladio in e-Motions. For the concrete syntax, we have used the same images that the PCM Bench uses to represent these concepts.

As for the behavior, in e-Motions we describe how systems evolve by describing all possible changes of the models by corresponding visual rewrite rules, that is, time-aware in-place transformation rules. Since the PCM metamodel only specifies those concepts relevant for the PCM language and the models obtained from the PCM Bench cannot be directly simulated or analyzed, we have conservatively enriched the PCM metamodel with new concepts to handle the control flow. Specifically, we have added a new concept `Token`—with a boolean attribute `completed`. `Token` objects represent requests or tasks in the system. A Palladio action with a `Token` associated, that has not been *completed*, may be executed and the `completed` attribute becomes *true*. A completed `Token` is passed to the following action. Thus, we may visualize that the execution of a Palladio model has a token "moving around" such model.

5 NFPs by modular definition and composition of observers

In previous work, we have proposed an approach for the specification and monitoring of non-functional properties using *observers* [16]. They are objects with which we extend the e-Motions definition of systems for the analysis of NFPs by simulation, such as mean and max cycle times, busy and idle cycles of operation units, throughput, mean-time between failures, etc. We also explored in [7, 17] how to define observers generically and independently from any system, so that they can afterwards be woven and merged with different systems. Given systems described as DSLs and generic DSLs defining the different observers, we can use these composition mechanisms to combine them. The result is that we can use the combined enriched system DSL to monitor NFPs of our systems.

We proved in [6] that, given very natural requirements on the observers and the instantiating mappings, the system thus obtained was a conservative enrichment of the original system, in the sense that the observers added *do not change the behavior of the system*.

We have defined DSLs for monitor the *response time* and *throughput* (the former is already implemented in PCM Bench while the latter is a new property not monitored in PCM Bench). Given a set of explicit bindings we have enriched conservatively out e-Motions specification of Palladio with such observers. Once we have enriched specification we may perform simulations and obtain results to be analyzed.

Acknowledgments. This work is partially funded by Project TIN2012-35669.

References

[1] S. Balsamo, A. DiMarco, P. Inverardi & M. Simeoni (2004): *Model-Based Performance Prediction in Software Development: A Survey*. IEEE Transactions on Software Engineering 30(5), pp. 295–310.

[2] Steffen Becker, Lars Grunske, Raffaela Mirandola & Sven Overhage (2006): *Performance Prediction of Component-Based Systems: A Survey from an Engineering Perspective*. In: Dagstuhl Seminar 04511: Architecting Systems with Trustworthy Components, LNCS 3938, Springer.

[3] Steffen Becker, Heiko Koziolek & Ralf Reussner (2007): *Model-Based Performance Prediction with the Palladio Component Model*. In: Proc. 6th Int'l Workshop on Software and Performance (WOSP'07), ACM.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott (2007): *All About Maude*. LNCS 4350, Springer.

[5] Francisco Durán (2014): *On the composition of graph-transformation-based DSL definitions*. In Santiago Escobar, editor: 10th International Workshop, WRLA 2014 @ ETAPS 2014, Grenoble, France. (To appear).

[6] Francisco Durán, Fernando Orejas & Steffen Zschaler (2013): *Behaviour Protection in Modular Rule-Based System Specifications*. In WADT 2012, Salamanca, Spain, LNCS 7841, Springer, pp. 24–49.

[7] Francisco Durán, Steffen Zschaler & Javier Troya (2013): *On the Reusable Specification of Non-functional Properties in DSLs*. In SLE 2012, LNCS 7745, Springer, pp. 332–351.

[8] Mathias Fritzsche, Jendrik Johannes, Steffen Zschaler, Anatoly Zherebtsov & Alexander Terekhov (2008): *Application of Tracing Techniques in Model-Driven Performance Engineering*. In 4th ECMDA Traceability.

[9] Vincenzo Grassi & Raffaela Mirandola (2004): *A Model-driven Approach to Predictive Non Functional Analysis of Component-based Systems*. In NfC 2004.

[10] Jens Happe, Heiko Koziolek & Ralf Reussner (2011): *Facilitating Performance Predictions Using Software Components*. IEEE Software 28(3), pp. 27–33.

[11] Antonio Moreno-Delgado, Francisco Durán, Steffen Zschaler & Javier Troya (2014): *Modular DSLs for Flexible Analysis: An e-Motions Reimplementation of Palladio*. In Jordi Cabot & Julia Rubin, editors: Modelling Foundations and Applications - 10th European Conference, ECMFA 2014 @ STAF 2014, York, UK, LNCS 8569, pp. 132–147.

[12] Antonio Moreno-Delgado, Javier Troya, Francisco Durán & Antonio Vallecillo (2013): *On the Modular Specification of NFPs: A Case Study*. In: JISBD 2013, pp. 302–316.

[13] José E. Rivera, Francisco Durán & Antonio Vallecillo (2009): *A Graphical Approach for Modeling Time-Dependent Behavior of DSLs*. In: Proc. of VL/HCC'09, Corvallis, Oregon (US).

[14] Simone Röttger & Steffen Zschaler (2007): *Tool Support for Refinement of Non-functional Specifications*. Software and Systems Modeling journal (SoSyM) 6(2), pp. 185–204.

[15] Simon Spinner, Samuel Kounev & Philipp Meier (2012): *Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0*. In Petri Nets 2012, LNCS 7347, Springer, pp. 388–397.

[16] Javier Troya, José E. Rivera & Antonio Vallecillo (2010): *Simulating Domain Specific Visual Models by Observation*. In: SpringSim'10, ACM, New York, NY, pp. 128:1–8.

[17] Javier Troya, Antonio Vallecillo, Francisco Durán & Steffen Zschaler (2013): *Model-driven performance analysis of rule-based domain specific visual models*. Inf. and Software Technology 55(1), pp. 88–110.

A fuzzy approach to cloud admission control for safe overbooking (High-level Work)*

Carlos Vázquez[1], Luis Tomás[2], Ginés Moreno[1], Johan Tordsson[2]
[1]Dept. of Computing Systems.
University of Castilla–La Mancha. Spain.
{Carlos.Vazquez,Gines.Moreno}@uclm.es
[2]Dept. of Computing Science.
Umeå University. Sweden.
{luis,tordsson}@cs.umu.se

Cloud computing enables elasticity - rapid provisioning and deprovisioning of computational resources. Elasticity allows cloud users to quickly adapt resource allocation to meet changes in their workloads. For cloud providers, elasticity complicates capacity management as the amount of resources that can be requested by users is unknown and can vary significantly over time. Overbooking techniques allow providers to increase utilization of their data centers. For safe overbooking, cloud providers need admission control mechanisms to handle the tradeoff between increased utilization (and revenue), and risk of exhausting resources, potentially resulting in penalty fees and/or lost customers. We propose a flexible approach (implemented with fuzzy logic programming) to admission control and the associated risk estimation. Our measures exploit different fuzzy logic operators in order to model optimistic, realistic, and pessimistic behaviour under uncertainty. An experimental evaluation confirm that our fuzzy admission control approach can significantly increase resource utilization while minimizing the risk of exceeding the total available capacity.

Keywords: Cloud Computing; Admission Control; Fuzzy Logic Programming; Risk Assessment

Cloud computing is a recently emerged paradigm where computational resources are leased over the Internet in a self-service manner under a pay-per use pricing scheme. Organizations and individuals, the cloud users, can thus continuously adjust their cloud resource allocations to their current needs, so called elasticity [6]. Consequently, it is common for cloud providers to require users to specify upper and lower limits to the number of VMs to be used in a *service request* [5]. For data centers, elasticity results in a long-term capacity allocation problem. Running too few VMs in total results in poor data center hardware utilization and lowered incomes from users, whereas having too many VMs may lead to low performance and/or crashes, poor user experience, and may also have financial consequences if Service Level Agreements (SLAs) regarding user performance expectations are violated. In our previous work [7], we demonstrate how *resource overbooking* can be used to increase provider utilization and revenue, with acceptable risks of running out of hardware capacity. Further examples of previous work in this area includes an algorithmic framework [2] that uses cloud effective demand to estimate the total physical capacity required for performing the overbooking, including probability of launching additional VMs in the future.

Admission control is associated with several uncertainties, including limited knowledge of future workloads, potential side effects from co-locating particular VMs, and exact impact on applications of potential resource shortage. Based on these properties of the admission control problem, we propose a fuzzy approach to admission control. Since its initial development by L. A. Zadeh in the sixties [9],

*This work was partially supported by the Swedish Research Council under grant number 2012- 5908, as well as by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-45732-C4-2-P.

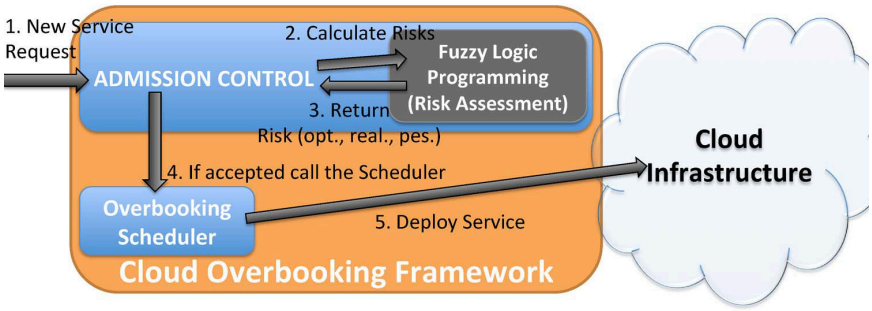


Figure 1: Conceptual picture of the system.

fuzzy logic has become a powerful theoretic tool for reaching elegant solutions to problems in various fields of software, industry, etc. In this sense, the MALP language represents a fuzzy extension of the popular Prolog language in the field of pure (crisp) logic programming [3]. In this fuzzy declarative framework, each program is accompanied with a lattice for modeling truth-degrees beyond the simpler case of the (crisp) *Boolean* pair $\{true, false\}$. Our proposal has been implemented with MALP using the tool FLOPER [4]. A conceptual overview of how our cloud overbooking framework use fuzzy logic during admission control is shown in Figure 1. The risks are calculated for the three capacity dimensions that we consider for each VM: *CPU*, *memory* and *I/O*, based on predicted information about future available capacity (referred to as *Free* in the rest of the paper), future amount of unrequested capacity (denoted *Unreq*) and the capacity requested by the incoming service (denoted *Req*). *Unreq* is the inverse difference between what users requested and what they really used (*Free*).

Our application uses a refined version of the usual lattice for fuzzy logic programming, $([0,1], \leq)$, as we try to identify the notion of truth-degree with the one for "*overbooking risk along a time period*". This means that instead of single values, our program manipulates lists of real numbers as truth-degrees[1] after analyzing the behaviour's curves representing "*free, unrequested, and requested (CPU/memory/net) resources*" also expressed as input lists to the tool. We have also implemented extended versions managing lists of the connectives defined by the logics of *Gödel*, *Łukasiewicz* and *Product*, that are useful for fine-tuning the more pessimistic or optimistic shape of the answers produced by our application under this uncertain scenario. Our MALP program receives as input parameters three lists representing the curves associated to free, unrequested and requested values, as well as a fourth argument indicating which resource, or `Field`, (CPU, network or memory) is considered.

This definition of predicate "`risk`" produces a truth degree that is a list of numbers obtained after contrasting the input curves "`Free`", "`Unreq`" and "`Req`". This evaluation is recursively performed by calling predicate "`combine`" with three concrete values each time in order to compare the requested resources with the free and unrequested values. The output of the program has the following shape:

`[avg(`n_1`), min(`n_2`), max(`n_3`),over([peak(`h_1`,`l_1`,`a_1`),...]), opt(`n_4`), real(`n_5`), pes(`n_6`)]`

Here, labels "`avg`", "`min`", and "`max`" contain the average (n_1), minimum (n_2), and maximum (n_3) values, respectively, of the input list; "`over`" gives the list of peaks (each one is represented by its maximum height (h_j), length (l_j), and area (a_j)) and finally, "`opt`", "`real`", and "`pes`" labels provide an optimistic (n_4), realistic (n_5), and pessimistic (n_6) estimation -based on the previous elements- about

[1] Sometimes accompanied with annotations like *max*, *avg*, *peak* and so on, for readability reasons.

Table 1: Performance Summary (Figure 2).

	Average utilization	Node capacity overpassed (%)	Aggregated node capacity overpassed (%)
No Risk	38.9 % (1)	0	0
Pessimistic	69.1 % (1.78)	0	0
Realistic	84.6 % (2.17)	6.99	0.43
Optimistic	92.5 % (2.38)	11.88	0.84

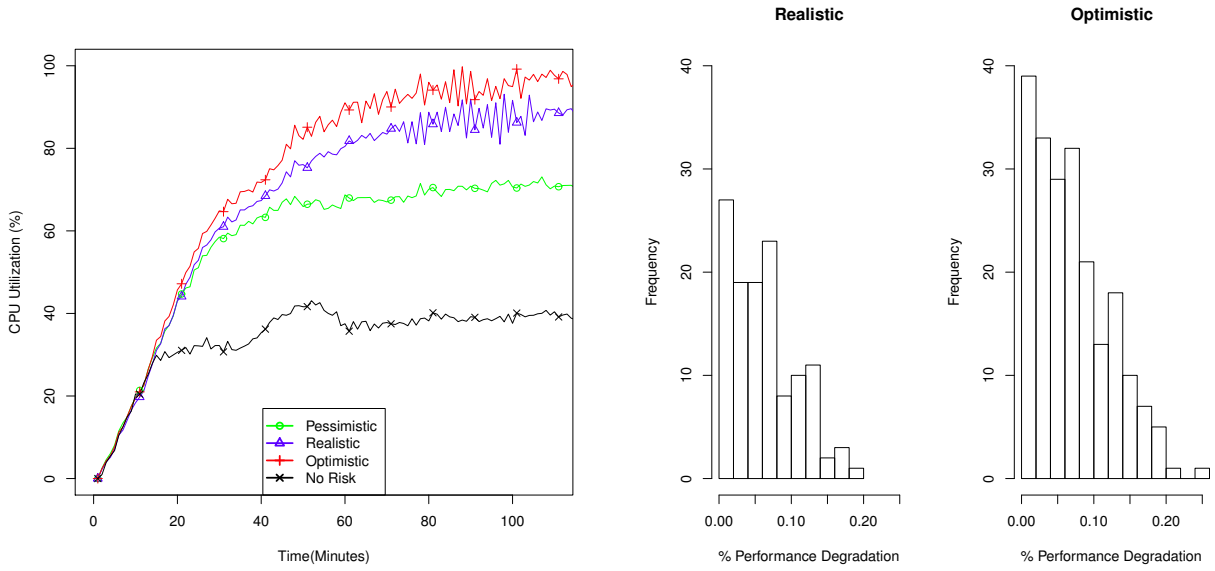
the risk of accepting the requested task. These estimations are produced by combining the average measure with the disjunctions of all the peaks by using different versions of the disjunction operators.

To evaluate our proposal, the fuzzy risk assessment is included into the framework presented in [7], which only included a simple admission control technique. The cloud infrastructure simulated for testing the different risk evaluators consists of 16 nodes where each one of them has 32 cores. We consider four different types of VMs (S, M, L and XL), where each one doubles the capacity of the previous one, starting from the S VM (1 CPU and 1.7GB of memory). Those VMs simulate the execution of a dynamic workload made of different kind of applications (some of them with steady behavior and others with bursty one), profiled by using monitoring tools after running the real applications. The workload is a mixture of applications, following a Poisson distribution for submission rates. See [7] for more details about the testbed and workload generation. The accepted requests (by the admission control) are scheduled and run on the 16 nodes. During this execution, we measure the *utilization* and *resource shortage*. The different risk values provided by the fuzzy logic engine are compared against each other and also against a base case where no overbooking is performed – no risks being taken. Those risk assessments from least risky to most are labeled as *"Pessimistic"*, *"Realistic"*, and *"Optimistic"* – mapping them to the respective values calculated by the fuzzy logic engine with those names. The base case is labeled *"No Risk"*.

Figure 2 (a) shows the resource utilization achieved by using the different risk values at the admission control. Clearly, the more risks we take, the higher utilization is achieved. However, this may have a negative impact regarding running out of resources if total capacity is overpassed, not only regarding the whole data center utilization but also regarding every single node into the system. Owing to that fact, Figure 2 (b) shows a histograms over how many times one of the nodes has overpassed its total capacity, and how large the impact on the performance is – performance degradation that may end up in resource SLA violations. The smaller the bars are, the better (less frequent risk situations). Notably, as shown in Figure 2 (a), the total infrastructure capacity is not overpassed. This means that *VM migration* can be used to decrease the risks by moving VMs from the overloaded nodes to the ones that still have enough available capacity. This way certain overload situations can be avoided, as has been proposed by Beloglazov et al. [1].

Finally, Table 1 highlights the improvement obtained thanks to performing resource overbooking and the cost that this entails. Pessimistic has the lowest improvement but without any performance degradation, while the other two techniques present higher utilization rates but at expense of higher performance degradation The evaluation shows significant increases in resource utilization obtained by our risk-aware fuzzy admission control methods. Even for the most optimistic estimates, available resources are exhausted as little as 0.84% of the time, while increasing utilization by 138%. Thus, our fuzzy methods are a promising approach to help the admission control to evaluate the risks associated with accepting a new service.

This work has sketched the material originally presented in [8]. Further direction include to extend our work by taking the risk assessment into account together with the SLA information. One such extension could be to specify different costs depending on the risk to be taken or using the different risk values depending on the penalty that is to be paid in case of SLA violation, i.e., the greater the penalty



(a) Data center utilization.

(b) Capacity overpassing (node level).

Figure 2: Resource utilization and risk assessment comparison.

the more pessimistic the admission control should be.

References

[1] Anton Beloglazov & Rajkumar Buyya (2013): *Managing Overloaded Hosts for Dynamic Consolidation of Virtual Machines in Cloud Data Centers Under Quality of Service Constraints*. IEEE Transactions on Parallel and Distributed Systems 24(7), pp. 1366–1379.

[2] D. Breitgand, Z. Dubitzky, A. Epstein, A. Glikson & I. Shapira (2012): *SLA-aware resource over-commit in an IaaS cloud*. In: *Proc. of the 8th Intl. Conference on Network and Service Management (CNSM)*, pp. 73–81.

[3] J.W. Lloyd (1987): *Foundations of Logic Programming*. Springer-Verlag, Berlin. 2nd edition.

[4] P.J. Morcillo, G. Moreno, J. Penabad & C. Vázquez (2010): *A Practical Management of Fuzzy Truth Degrees using FLOPER*. In: *Proc. of 4th Intl. Symposium on Rule Interchange and Applications, RuleML'10. Springer Verlag, LNCS 6403*, pp. 119–126. Available at `http://dx.doi.org/10.1007/978-3-642-16289-3_4`.

[5] B. Rochwerger, D. Breitgand & et al. (2009): *The Reservoir model and architecture for open federated cloud computing*. *IBM J. Res. Dev.* 53(4), pp. 535–545.

[6] The NIST Definition of Cloud Computing (Visited 2013-05-30): Web page at `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[7] Luis Tomás & Johan Tordsson (2013): *Improving Cloud Infrastructure Utilization through Overbooking*. In: *Proc. of the ACM Cloud and Autonomic Computing Conference (CAC)*, p. 5. Available at `http://doi.acm.org/10.1145/2494621.2494627`.

[8] C. Vázquez, L. Tomás, G. Moreno & J. Tordsson (2013): *A fuzzy approach to cloud admission control for safe overbooking*. In: *Proc. of 10th International Workshop on Fuzzy Logic and Applications, WILF 2013.*, Springer Verlag, LNAI 8256, pp. 212–225. Available at `http://dx.doi.org/10.1007/978-3-319-03200-9_22`.

[9] L. A. Zadeh (1965): *Fuzzy Sets*. *Information and Control* 8(3), pp. 338–353.

An operational framework to reason about policy behavior in trust management systems (High-level Work)

Edelmira Pasarella
Departament de Ciències de la Computació
Universitat Politècnica de Catalunya *
edelmira@cs.upc.edu

Jorge Lobo
Institució Catalana de Recerca i Estudis Avançats (ICREA)
Universitat Pompeu Fabra †
jorge.lobo@upf.edu

In this paper we show that the logical framework proposed by Becker et al. to reason about security policy behavior in a trust management context can be captured by an operational framework that is based on the language proposed by Miller to deal with scoping and/or modules in logic programming in 1989. The framework of Becker et al. uses propositional Horn clauses to represent both policies and credentials, implications in clauses are interpreted in counterfactual logic, a Hilbert-style proof is defined and a system based on SAT is used to proof whether properties about credentials, permissions and policies are valid in trust management systems, i.e. formulas that are true for all possible policies. Our contribution is to show that instead of using a SAT system, this kind of validation can rely on the operational semantics (derivability relation) of Miller's language, which is very close to derivability in logic programs, opening up the possibility to extend Becker et al.'s framework to the more practical first order case since Miller's language is first order.

1 Introduction

Trust Management Systems (TMS) [2] are perhaps the most common model to describe distributed access control. In this model, there are (1) policies that define under what conditions a subject is able to access resources, (2) credentials that are provided by the subject in order to fulfill policies and (3) decisions of whether a subject has particular permissions. One of the most popular ways to describe TMS is to use logic programming-like languages for definition of policies and credentials (see for example [3], [4],[5],[6]). Then, permissions are decided by inferences done combining policy and credentials.

Working under this framework, Becker et al. [1] have recently proposed a logic system in which one can reason about TMS in general. In this system, a Hilbert-style axiomatization is defined and a system based on SAT solvers is used to prove automatically TMS properties. In their logic, policies and credentials are formalized using propositional logic programs, while permissions are propositional Boolean formulas. Trust management behavior, i.e. to determine whether a permission p is true under a policy P when presenting a set of credentials Q, is captured by proving that the statement $Q \supset p$ *holds in the policy* P.[1] This statement is true if p holds in the policy P extended with those clauses in the credentials Q: $P \cup Q \vdash p$.

Example 1.1 *To illustrate how policies, credentials and permissions are expressed, let us consider a very simple example about purchasing digital goods. The policy of the seller is:*

*Partially supported by the Spanish CICYT project FORMALISM (Ref. TIN2007-66523) and by the AGAUR Research Grant ALBCOM (Ref. SGR20091137).

†Partially supported by the US Army Research Lab and the UK Ministry of Defence under agreement number W911NF-06-3-0001.

[1] In [1] formulas like $Q \supset G$ are written as $\Box_Q G$ but we will follow Miller's notation.

To appear in EPTCS.

$\{X.paid(Music, 1.99\$) \supset X.download(Music);\ paypal(X,V);\ wire\_\text{transfer}(X,V)\}$

*that says "if X paid the fee Fee for the music Music, X is able to download Music" and, there are two ways in which X can pay a fee V: either by means of paypal or by means of a wire*_transfer".[2] *These statements represent schemes of policies for specific values of the arguments. Credentials for paying the fee presented by a subject X to request download permission can be written as follows:*

$(paypal(X, 1.99\$) \supset X.paid(song, 1.99\$)) \supset X.download(song)$

To allow the subject X to download Music, the system has to join the policy and the credential $(paypal(X, 1.99\$) \supset X.paid(song, 1.99))$ *in a single program and then verify that the permission* $X.download(song)$ *holds in it.*

The important contribution of Becker et al.'s work is the definition of valid formulas in TMS. Informally, these are formulas that are true regardless of the policies and credentials that can be defined in the TMS. Hence, they are able to describe how to approach proofs such as *proving attacks* (i.e. discovering policies) in specific TMS systems, or general properties such as the transitivity of credential-based derivations.

The authors, however, argue that Hilbert style axiomatizations are difficult for building proofs because they are not goal oriented. Hence, they resort to an algorithm that interleaves syntactic transformations of formulas and calls to SAT solvers in order to do automatic verifications. In their paper there is an argument but not a proof that the mechanization is correct. A proof may be possible but probably not easy.

In this work we show that the logical framework proposed by Becker et al. can be captured by an operational framework that is based on a language proposed by Miller in 1989 to deal with scoping and/or modules in logic programming. *Our contribution is to show that we can rely on the operational semantics (derivability relation) of Miller's language, which is very close to derivability in logic programs, to do goal oriented formula verification.* This connection also open the possibility of extending Becker et al.'s framework to the more practical first order case since Miller's language is first order.

2 Trust management systems

In the following we assume the existence of an underlying propositional signature Σ that consists of a countable set of propositional variables. We call these propositional symbols Σ-atoms. For the sake of simplicity, in the following, we omit the prefix Σ- when it is clear from the context.

Definition 2.1 *Programs, clauses and goals are defined using the BNF presented below, where A, F, C, P and G range over (1) atoms, conjunctions of atoms, (2) clauses, (3) programs and (4) goals, respectively.*

(1) $F ::= \texttt{true} \mid A \mid F \wedge F$ *(2)* $C ::= F \supset A$

(3) $P ::= C \mid C;P$ *(4)* $G ::= F \mid \neg G \mid P \supset G \mid G \wedge G \mid G \vee G$

Perhaps the most unusual definition is the definition of goals. A *goal* is either an expression of the form $P \supset G$, where P is a program and (inductively) G a goal or an expression corresponding to a propositional formula built with the standard connectives $\neg$, $\wedge$ and $\vee$. We note that our goals are the formulas defined in Becker et al.'s. As usual, we define implication, $G_1 \rightarrow G_2$, as $\neg G_1 \vee G_2$ and equivalence, $G_1 \leftrightarrow G_2$, as $(G_1 \rightarrow G_2) \wedge (G_2 \rightarrow G_1)$. Throughout the rest of the paper, we adopt the following conventions: P and Q denote programs. Clauses may be embraced in parenthesis. In a clause of the form $\texttt{true} \supset p$ we simply write p.

[2] We make the simplifying assumption that no third party is involved in the TMS and that the seller has access to paypal and the bank.

2.1 Reasoning in trust management systems

In this section we introduce an operational framework to reason about policies, credentials and permissions. We denote this framework by $\hat{O}$-TMF. The $\hat{O}$-TMF framework is based on the language introduced in [7]. The semantics is presented in terms of a derivation relation over sequents. A *sequent* is a pair of the form $P \vdash G$, where the program P is called the *antecedent* and the goal G the *succedent*. As explained earlier, the intuitive interpretation of embedded implications is that, given a program P, to prove the query $Q \supset G$ it is necessary to prove G with the program $P \cup Q$. This is formalized in [7] using the following inference rule: $\frac{P \cup Q \vdash G}{P \vdash Q \supset G}$

2.2 $\hat{O}$- proof rules

The inference rules for sequents in $\hat{O}$-TMF are defined over *Programs* $\times$ *Goals* as follows

$$\frac{P \vdash_{\hat{O}} G_i}{P \vdash_{\hat{O}} G_1 \vee G_2}\ i = 1,2 \qquad \frac{P \vdash_{\hat{O}} G_1 \quad P \vdash_{\hat{O}} G_2}{P \vdash_{\hat{O}} G_1 \wedge G_2} \qquad \frac{P \cup Q \vdash_{\hat{O}} G}{P \vdash Q \supset G} \qquad \frac{P \vdash_{\hat{O}} Q \supset \neg G}{P \vdash_{\hat{O}} \neg(Q \supset G)}$$

$$\frac{P \vdash_{\hat{O}} \neg G_1 \wedge \neg G_2}{P \vdash_{\hat{O}} \neg(G_1 \vee G_2)} \qquad \frac{P \vdash_{\hat{O}} \neg G_1 \vee \neg G_2}{P \vdash_{\hat{O}} \neg(G_1 \wedge G_2)} \qquad \frac{P \vdash_{\hat{O}} A}{P \vdash_{\hat{O}} \neg\neg A}$$

An $\hat{O}$-proof for $P \vdash_{\hat{O}} G$ is a tree in which nodes are labeled with sequents such that (i) the root node is labeled with $P \vdash_{\hat{O}} G$, (ii) the internal nodes are instances of one of the above inference rules and (iii) the leaf nodes are labeled with *initial sequents*. An *initial sequents* is a sequent of the form $P' \vdash_{\hat{O}} G'$ where G' is a propositional formula that is *true* in the minimal model of P'.

We can prove the validity of formulas as follow. First, we need the following definition:

Definition 2.2 *Let G be a goal and Σ_G be the signature formed by the set of propositional atoms occurring in G. G is valid, $\vdash_{\hat{O}} G$, in the $\hat{O}$-TMF if and only if it is not possible to find a Σ_G-policy Δ such that $\Delta \vdash_{\hat{O}} \neg G$*

As we will see in the next section, our definition of validity in TMS is equivalent to Becker et al.'s definition.

3 Equivalence of $\hat{O}$-TMF and Becker et al.'s TMS

In this section we briefly described how to show that our definition of validity is equivalent to Becker et al.'s. First, we recall some definitions from [1]. Let *basic* goals be atoms and classical propositional compound formulas expressed in terms of $\wedge$ and $\neg$. Let P be a policy, M_P its minimal model and G a basic goal. Then, $P \Vdash G$ if and only if G holds in M_P. Additionally, Becker et al. inductively define that $P \Vdash Q \supset G$ if and only if $P \cup Q \Vdash G$, where Q is a policy. A goal G is valid, $\Vdash G$, if and only if for every policy P, $P \Vdash G$. In order to deal with goals that allow the evaluation of policies together with credentials, we need the following lemma.

Proposition 3.1 *Let $Q \supset G$ be a goal. Then, for all policies P*

$$P \Vdash Q \supset G \quad \textit{if and only if} \quad P \vdash_{\hat{O}} Q \supset G \quad \blacksquare$$

The proof follows by induction by showing that $P \Vdash Q_1 \supset \ldots Q_k \supset G$ iff $P \vdash_{\hat{O}} Q_1 \supset \ldots Q_k \supset G$ using the definition of $\Vdash$ and the $\supset$-$\hat{O}$ rule. As a corollary we also have

$$P \Vdash G \quad \text{if and only if} \quad P \vdash_{\hat{O}} G \qquad (1)$$

We use one of the main results in [1] as well. This is, the equivalence between their derivability and their proof system validity . Given a formula φ,

$$\Vdash \varphi \quad \text{if and only if} \quad \vdash \varphi \qquad (2)$$

Theorem 3.1 *Let G be a goal. Then,* $\vdash G$ *if and only if* $\vdash_{\hat{O}} G$
Proof 3.1 *From (1) and (2) it follows that* $\vdash G$ *if and only if* $\Vdash G$ *if and only if* $\Vdash_{\hat{O}} G$ ■

4 Final remarks

In this work we have presented a very operational definition of validity in TMS. Based on this result we have designed a top-down proof procedure of validity. This procedure works similar to abduction in logic programs with the addition that not only atoms but also rules can be assumed in order to find Δs (see Def. 2). We are able also to describe a model theoretic semantics based on Kripke structures following Miller's models. In particular, Miller interprets a world of a Kripke's model as a program and the knowledge at each world as its minimal model. This intuition can be explained in terms of two basic ideas of modal logic. The first one is the notion that a *world* may be considered to represent the "knowledge" that we have at a certain moment. The second idea is that a formula can be considered to hold if we can infer its truth from the knowledge that we have now or one that we may acquire in the "future", capturing the idea of credentials. Details will appear in the full version of this paper.

An important consequence of the connections between Miller's language and the propositional logic for reasoning in TMS is the possibility of lifting the results to policies, credentials and permissions with variables. We cannot apply directly Miller's results because his logic doesn't deal with negation. There is, however, an extensions to Miller's logic that deals with normal logic programs [8], but we need to work out the details of the axiomatization since the approach in [8] uses a notion similar to Clark's completion as opposed to minimal models. Complementary to these extensions we will also like to check how an implementation of validity using our approach will compare to the implementation of Becker et al.

References

[1] Moritz Y. Becker, Alessandra Russo & Nik Sultana (2012): *Foundations of Logic-Based Trust Management*. In: *IEEE Symposium on Security and Privacy*, pp. 161–175. Available at http://doi.ieeecomputersociety.org/10.1109/SP.2012.20.

[2] Matt Blaze, Joan Feigenbaum & Jack Lacy (1996): *Decentralized Trust Management*. In: *IEEE Symposium on Security and Privacy*, pp. 164–173. Available at http://dx.doi.org/10.1109/SECPRI.1996.502679.

[3] John DeTreville (2002): *Binder, a Logic-Based Security Language*. In: *IEEE Symposium on Security and Privacy*, pp. 105–113. Available at http://doi.ieeecomputersociety.org/10.1109/SECPRI.2002.1004365.

[4] Trevor Jim (2001): *SD3: A Trust Management System with Certified Evaluation*. In: *IEEE Symposium on Security and Privacy*, pp. 106–115. Available at http://doi.ieeecomputersociety.org/10.1109/SECPRI.2001.924291.

[5] Ninghui Li, Benjamin N. Grosof & Joan Feigenbaum (2000): *A Practically Implementable and Tractable Delegation Logic*. In: *IEEE Symposium on Security and Privacy*, pp. 27–42. Available at http://doi.ieeecomputersociety.org/10.1109/SECPRI.2000.848444.

[6] Ninghui Li & John C. Mitchell (2003): *DATALOG with Constraints: A Foundation for Trust Management Languages*. In: *PADL*, pp. 58–73. Available at http://dx.doi.org/10.1007/3-540-36388-2_6.

[7] Dale Miller (1989): *A Logical Analysis of Modules in Logic Programming*. *J. Log. Program.* 6(1&2), pp. 79–108. Available at http://dx.doi.org/10.1016/0743-1066(89)90031-9.

[8] Edelmira Pasarella, Fernando Orejas, Elvira Pino & Marisa Navarro (2012): *Semantics of structured normal logic programs*. *J. Log. Algebr. Program.* 81(5), pp. 559–584. Available at http://dx.doi.org/10.1016/j.jlap.2012.03.001.

Site-Level Template Extraction Based on Hyperlink Analysis (Original Work)

Julián Alarte David Insa Josep Silva
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València, Valencia, Spain
jalarte@dsic.upv.es dinsa@dsic.upv.es jsilva@dsic.upv.es

Salvador Tamarit
Babel Research Group
Universidad Politécnica de Madrid, Madrid, Spain
stamarit@babel.ls.fi.upm.es

Web templates are one of the main development resources for website engineers. Templates allow them to increase productivity by plugin content into already formatted and prepared pagelets. For the final user templates are also useful, because they provide uniformity and a common look and feel for all webpages. However, from the point of view of crawlers and indexers, templates are an important problem, because templates usually contain irrelevant information such as advertisements, menus, and banners. Processing and storing this information is likely to lead to a waste of resources (storage space, bandwidth, etc.). It has been measured that templates represent between 40% and 50% of data on the Web. Therefore, identifying templates is essential for indexing tasks. In this work we propose a novel method for automatic template extraction that is based on similarity analysis between the DOM trees of a collection of webpages that are detected using menus information. Our implementation and experiments demonstrate the usefulness of the technique.

1 Introduction

A web template (in the following just template) is a prepared HTML page where formatting is already implemented and visual components are ready so that we can insert content into them. Templates are used as a basis for composing new webpages that share a common look and feel. This is good for web development because many tasks can be automated thanks to the reuse of components. In fact, many websites are maintained automatically by code generators that generate webpages using templates. Templates are also good for users, which can benefit from intuitive and uniform designs with a common vocabulary of colored and formatted visual elements.

Templates are also important for crawlers and indexers, because they usually judge the relevance of a webpage according to the frequency and distribution of terms and hyperlinks. Since templates contain a considerable number of common terms and hyperlinks that are replicated in a large number of webpages, relevance may turn out to be inaccurate, leading to incorrect results (see, e.g., [1, 17, 19]). Moreover, in general, templates do not contain relevant content, they usually contain one or more pagelets [5, 1] (i.e., self-contained logical regions with a well defined topic or functionality) where the main content must be inserted. Therefore, detecting templates can allow indexers to identify the main content of the webpage.

Modern crawlers and indexers do not treat all terms in a webpage in the same way. Webpages are preprocessed to identify the template because template extraction allows them to identify those pagelets that only contain noisy information such as advertisements and banners. This content should not be indexed in the same way as the relevant content. Indexing the non-content part of templates not only

Submitted to:
PROLE 2014

affects accuracy, it also affects performance and can lead to a waste of storage space, bandwidth, and time.

Template extraction helps indexers to isolate the main content. This allows us to enhance indexers by assigning higher weights to the really relevant terms. Once templates have been extracted, they are processed for indexing—they can be analyzed only once for all webpages using the same template—. Moreover, links in templates allow indexers to discover the topology of a website (e.g., through navigational content such as menus), thus identifying the main webpages. They are also essential to compute pageranks.

Gibson et al. [8] determined that templates represent between 40% and 50% of data on the Web and that around 30% of the visible terms and hyperlinks appear in templates. This justifies the importance of template removal [19, 17] for web mining and search.

Our approach to template extraction is based on the DOM [6] structures that represent webpages. Roughly, given a webpage in a website, we first identify a set of webpages that are likely to share a template with it, and then, we analyze these webpages to identify the part of their DOM trees that is common with the original webpage. This slice of the DOM tree is returned as the template.

Our technique introduces a new idea to automatically find a set of webpages that potentially share a template. Roughly, we detect the template's menu and analyze the links of the menu to identify a set of mutually linked webpages. One of the main functions of a template is in aiding navigation, thus almost all templates provide a large number of links, shared by all webpages implementing the template. Locating the menu allows us to identify in the topology of the website the main webpages of each category or section. These webpages very likely share the same template. This idea is simple but powerful and, contrarily to other approaches, it allows the technique to only analyze a reduced set of webpages to identify the template.

The rest of the paper has been structured as follows: In Section 2 we discuss the state of the art and show some problems of current techniques that can be solved with our approach. In Section 3 we provide some preliminary definitions and useful notation. Then, in Section 4, we present our technique with examples and explain the algorithms used. In Section 5 we give some details about the implementation and show the results obtained from a collection of benchmarks. Finally, Section 6 concludes.

2 Related Work

Template detection and extraction are hot topics due to their direct application to web mining, searching, indexing, and web development. For this reason, there are many approaches that try to face this problem. Some of them have been presented in the CleanEval competition [2], which periodically proposes a collection of examples to be analyzed with a gold standard. The examples proposed are especially thought for boilerplate removal and content extraction.

Content Extraction is a discipline very close to template extraction. Content extraction tries to isolate the pagelet with the main content of the webpage. It is an instance of a more general discipline called *Block Detection* that tries to isolate every pagelet in a webpage. There are many works in these fields (see, e.g., [10, 18, 4, 11]), and all of them are directly related to template extraction.

In the area of template extraction, there are three main different ways to solve the problem, namely, (i) using the textual information of the webpage (i.e., the HTML code), (ii) using the rendered image of the webpage in the browser, and (iii) using the DOM tree of the webpage.

The first approach is based on the idea that the main content of the webpage has more density of text, with less labels. For instance, the main content can be identified selecting the largest contiguous

text area with the least amount of HTML tags [7]. This has been measured directly on the HTML code by counting the number of characters inside text, and characters inside labels. This measure produces a ratio called CETR [18] used to discriminate the main content. Other approaches exploit densitometric features based on the observation that some specific terms are more common in templates [14, 12]. The distribution of the code between the lines of a webpage is not necessarily the one expected by the user. The format of the HTML code can be completely unbalanced (i.e., without tabulations, spaces or even carriage returns), specially when it is generated by a non-human directed system. As a common example, the reader can see the source code of the main Google's webpage. At the time of writing these lines, all the code of the webpage is distributed in only a few lines without any legible structure. In this kind of webpages CETR is useless.

The second approach assumes that the main content of a webpage is often located in the central part and (at least partially) visible without scrolling [3]. This approach has been less studied because rendering webpages for classification is a computational expensive operation [13].

The third approach is where our technique falls. While some works try to identify pagelets analyzing the DOM tree with heuristics [1], others try to find common subtrees in the DOM trees of a collection of webpages in the website [19, 17]. Our technique is similar to these last two works.

Even though [19] uses a method for template extraction, its main goal is to remove redundant parts of a website. For this, they use the Site Style Tree (SST), a data structure that is constructed by analyzing a set of DOM trees and recording every node found, so that repeated nodes are identified by using counters in the SST nodes. Hence, an SST summarizes a set of DOM trees. After the SST is built, they have information about the repetition of nodes. The most repeated nodes are more likely to belong to a noisy part that is removed from the webpages.

In [17], the approach is based on discovering optimal mappings between DOM trees. This mapping relates nodes that are considered redundant. Their technique uses the RTDM-TD algorithm to compute a special kind of mapping called *restricted top-down mapping* [15]. Their objective, as ours, is template extraction, but there are two important differences. First, we compute another kind of mapping to identify redundant nodes. Our mapping is more restrictive because it forces all nodes that form pairs in the mapping to be equal. Second, in order to select the webpages of the website that should be mapped to identify the template, they pick random webpages until a threshold is reached. In their experiments, they approximated this threshold as a few dozens of webpages. In our technique, we do not select the webpages randomly, we use a method to identify the webpages linked by the main menu of the website because they very likely contain the template. We only need to explore a few webpages to identify the webpages that implement the template. Moreover, contrarily to us, they assume that all webpages in the website share the same template, and this is a strong limitation for many websites.

3 Preliminaries

The Document Object Model (DOM) [6] is an API that provides programmers with a standard set of objects for the representation of HTML and XML documents. Our technique is based on the use of DOM as the model for representing webpages. Given a webpage, it is completely automatic to produce its associated DOM structure and vice-versa. In fact, current browsers automatically produce the DOM structure of all loaded webpages before they are processed.

The DOM structure of a given webpage is a tree where all the elements of the webpage are represented (included scripts and CSS styles) hierarchically. This means that a table that contains another table is represented with a node with a successor that represents the internal table.

In the following webpages are represented with a DOM tree $T = (N,E)$ where N is a finite set of nodes and E is a set of edges between nodes in N (see Figure 1). $root(T)$ denotes the root node of T. Given a node $n \in N$, $link(n)$ denotes the hyperlink of n when n is a node that represents a hyperlink (HTML label `<a>`). $parent(n)$ represents node $n' \in N$ such that $(n',n) \in E$. Similarly, $children(n)$ represents the set $\{n' \in N \mid (n,n') \in E\}$. $subtree(n)$ denotes the subtree of T whose root is $n \in N$. $path(n)$ is a non-empty sequence of nodes that represents a *DOM path*; it can be defined as $path(n) = n_0 n_1 \ldots n_m$ such that $\forall i, 0 \leq i < m.\ n_i = parent(n_{i+1})$.

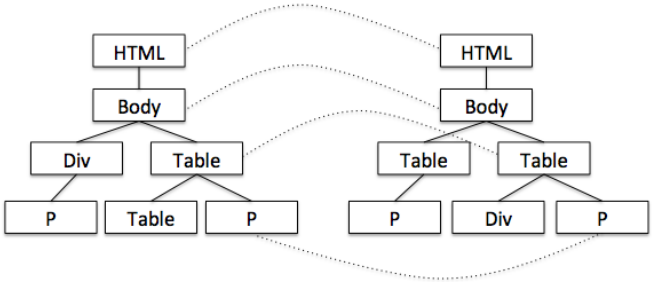


Figure 1: Exact top-down mapping between DOM trees

In order to identify the part of the DOM tree that is common in a set of webpages, our technique uses an algorithm that is based on the notion of mapping. A mapping establishes a correspondence between the nodes of two trees.

Definition 3.1 *(based on Kuo's definition of mapping [16]) A* mapping *from a tree $T = (N,E)$ to a tree $T' = (N',E')$ is any set M of pairs of nodes $(n,n') \in M$, $n \in N, n' \in N'$ such that, for any two pairs (n_1,n'_1) and (n_2,n'_2) in M, $n_1 = n_2$ iff $n'_1 = n'_2$.*

In order to identify templates, we are interested in a very specific kind of mapping that we call *exact top-down mapping* (ETDM).

Definition 3.2 *Given an equality relation $\triangleq$ between tree nodes, a mapping M between two trees T and T' is said to be* exact top-down *if and only if*

- *exact: for every pair $(n,n') \in M$, $n \triangleq n'$.*
- *top-down: for every pair $(n,n') \in M$, with $n \neq root(T)$ and $n' \neq root(T')$, there is also a pair $(parent(n), parent(n')) \in M$.*

Note that this definition is parametric with respect to the equality relation $\triangleq$. We could simply use the standard equality (=), but we left this relation open, to be general enough as to cover any possible implementation. In particular, other techniques consider that two nodes n_1 and n_2 are equal if they have the same label. However, in our implementation we use a notion of node equality much more complex that uses the label of the node, its CSS classes, its HTML identifier, its children, its position in the DOM tree, etc.

This definition of mapping allows us to be more restrictive than other mappings such as, e.g., the *restricted top-down mapping* (RTDM) introduced in [15]. While RTDM permits the mapping of different nodes (e.g., a node labelled with *table* with a node labelled with *div*), ETDM can force all pairwise mapped nodes to have the same label. Figure 1 shows an example of an ETDM using: $n \triangleq n'$ if and only if n and n' have the same label. We can now give a definition of template using ETDM.

Figure 2: Webpages of BBC sharing a template

Definition 3.3 *Let p_0 be a webpage whose associated DOM tree is $T_0 = (N_0, E_0)$, and let $P = \{p_1 \dots p_n\}$ be a collection of webpages with associated DOM trees $\{T_1 \dots T_n\}$. A* template *of p_0 with respect to P is a tree (N, E) where*

- *nodes: $N = \{n \in N_0 \mid \forall i, 1 \leq i \leq n \;.\; (n, \_) \in M_{T_0,T_i}\}$ where M_{T_0,T_i} is an exact top-down mapping between trees T_0 and T_i.*
- *edges: $E = \{(m,m') \in E_0 \mid m,m' \in N\}$.*

Hence, the template of a webpage is computed with respect to a set of webpages (usually webpages in the same website). We formalize the template as a new webpage computed with an ETDM between the initial webpage and all the other webpages.

4 Template extraction

Templates are often composed of a set of pagelets. Two of the most important pagelets in a webpage are the menu and the main content. For instance, in Figure 2 we see two webpages that belong to the "News" portal of BBC. At the top of the webpages we see the main menu containing links to all BBC portals. We can also see a submenu under the big word "News". The left webpage belongs to the "Technology" section, while the right webpage belongs to the "Science & Environment" section. Both share the same menu, submenu, and general structure. In both pages the news are inside the pagelet in the dashed square. Note that this pagelet contains the main content and, thus, it should be indexed with a special treatment. In addition to the main content, there is a common pagelet called "Top Stories" with the most relevant news, and another one called "Features and Analysis".

Our technique inputs a webpage (called key page) and it outputs its template. To infer the template, it analyzes some webpages from the (usually huge) universe of directly or indirectly linked webpages. Therefore, we need to decide what concrete webpages should be analyzed. Our approach is very simple yet powerful:

1. Starting from the key page, it identifies a complete subdigraph in the website topology, and then
2. it extracts the template by calculating an ETDM between the DOM tree of the key page and some of the DOM trees of the webpages in the complete subdigraph.

Both processes are explained in the following sections.

4.1 Finding a complete subdigraph in a website topology

Given a website topology, a complete subdigraph (CS) represents a collection of webpages that are pairwise mutually linked. A n-complete subdigraph (n-CS) is formed by n nodes. Our interest in complete subdigraphs comes from the observation that the webpages linked by the items in a menu usually form a CS. This is a new way of identifying the webpages that contain the menu. At the same time, these webpages are the roots of the sections linked by the menu. The following example illustrates why menus provide very useful information about the interconnection of webpages in a given website.

Example 4.1 *Consider the BBC website. Two of its webpages are shown in Figure 2. In this website all webpages share the same template, and this template has a main menu that is present in all webpages, and a submenu for each item in the main menu. The site map of the BBC website may be represented with the topology shown in Figure 3.*

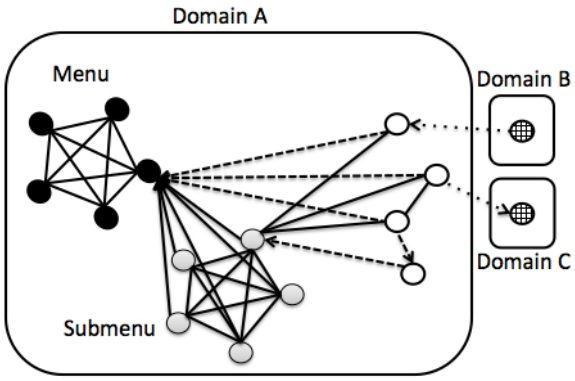


Figure 3: BBC Website topology

In this figure, each node represents a webpage and each edge represents a link between two webpages (we only draw some of the edges for clarity). Solid edges are bidirectional, and dashed and dotted edges are directed. Black nodes are the webpages pointed by the main menu. Because the main menu is present in all webpages, then all nodes are connected to all black nodes. Therefore all black nodes together form a complete graph (i.e., there is an edge between each pair of nodes). Grey nodes are the webpages pointed by a submenu, thus, all grey nodes together also form a complete graph. White nodes are webpages inside one of the categories of the submenu, therefore, all of them have a link to all black and all grey nodes.

Of course, not all webpages in a website implement the same template, and some of them only implement a subset of a template. For this reason, one of the main problems of template extraction is deciding what webpages should be analyzed. Minimizing the number of webpages analyzed is essential to reduce the web crawlers work. In our technique we introduce a new idea to select the webpages that must be analyzed: we identify a menu in the key page and we analyze the webpages pointed out by this menu. Observe that we only need to investigate the webpages linked by the key page, because they will for sure contain a CS that represents the menu.

In order to increase precision, we search for a CS that contains enough webpages that implement the template. This CS can be identified with Algorithm 1.

This algorithm inputs a webpage and the size n of the CS to be computed. We have empirically approximated the optimal value for n, which is 4. This is further discussed in Section 5.2. The algorithm

Algorithm 1 Extract a n-CS from a website

Input: An *initialLink* that points to a webpage and the expected size n of the CS.
Output: A set of links to webpages that together form a n-CS.
If a n-CS cannot be formed, then they form the biggest m-CS with m < n.

begin
$keyPage = loadWebPage(initialLink)$;
$reachableLinks = getLinks(keyPage)$;
$processedLinks = \emptyset$;
$connections = \emptyset$;
$bestCS = \emptyset$;
foreach $link$ **in** $reachableLinks$
$webPage = loadWebPage(link)$;
$existingLinks = getLinks(webPage) \cap reachableLinks$;
$processedLinks = processedLinks \cup \{link\}$;
$connections = connections \cup \{(link \to existingLink) \mid existingLink \in existingLinks\}$;
$CS = \{ls \in \mathscr{P}(processedLinks) \mid link \in ls \wedge \forall l, l' \in ls \,.\, (l \to l'), (l' \to l) \in connections\}$;
$maximalCS = cs \in CS$ such that $\forall cs' \in CS \,.\, |cs| \geq |cs'|$;
if $|maximalCS| = n$ **then return** $maximalCS$;
if $|maximalCS| > |bestCS|$ **then** $bestCS = maximalCS$;
return $bestCS$;
end

uses two trivial functions: $loadWebPage(link)$, which loads and returns the webpage pointed by the input link, and $getLinks(webpage)$, which returns the collection of (non-repeated) links[1] in the input webpage (ignoring self-links). Observe that the main loop iteratively explores the links of the webpage pointed by the *initialLink* (i.e., the key page) until it founds a n-CS. Note also that it only loads those webpages needed to find the n-CS, and it stops when the n-CS has been found. We want to highlight the mathematical expression

$$CS = \{ls \in \mathscr{P}(processedLinks) \mid link \in ls \wedge \forall l, l' \in ls \,.\, (l \to l'), (l' \to l) \in connections\},$$

where $\mathscr{P}(X)$ returns all possible partitions of set X.

It is used to find the set of all CS that can be constructed with the current $link$. Here, *processedLinks* contains the set of links that have been already explored by the algorithm. And *connections* is the set of all links between the webpages pointed by *processedLinks*. Hence, the CS is composed of the subset of *processedLinks* that form a CS with links in *connections*.

Observe that the current link must be part of the CS ($link \in ls$) to ensure that we make progress (not repeating the same search of the previous iteration). Moreover, because the CS is constructed incrementally, the statement

if $|maximalCS| = n$ **then return** $maximalCS$

ensures that whenever a n-CS can be formed, it is returned.

4.2 Template extraction from a complete subdigraph

After we have found a set of webpages linked by the menu of the site (the complete subdigraph), we identify an ETDM between the key page and all webpages in the set. For this, initially, the template is considered to be the key page. Then, we compute an ETDM between the template and one webpage in the set. The result is the new refined template, that is further refined with another ETDM with another webpage, and so on until all webpages have been processed. This process is formalized in Algorithm 2, that uses function *ETDM* to compute the biggest ETDM between two trees.

[1] In our implementation, this function removes those links that point to other domains because they are very unlikely to contain the same template.

Algorithm 2 Extract a template from a set of webpages

Input: A key page $p_k = (N_1, E_1)$ and a set of n webpages P.
Output: A template for p_k with respect to P.

begin
  $template = p_k$;
  foreach (p **in** P)
    if $root(p_k) \triangleq root(p)$
      $template = ETDM(template, p)$;
  return $template$;
end

function $ETDM$(tree $T_1 = (N_1, E_1)$, tree $T_2 = (N_2, E_2)$)
  $r_1 = root(T_1)$;
  $r_2 = root(T_2)$;
  nodes = $\{r_1\}$;
  edges = $\emptyset$;
  foreach $n_1 \in N_1, n_2 \in N_2$. $n_1 \triangleq n_2, (r_1, n_1) \in E_1$ and $(r_2, n_2) \in E_2$
    $(nodes\_st, edges\_st) = ETDM(subtree(n_1), subtree(n_2))$;
    $nodes = nodes \cup nodes\_st$;
    $edges = edges \cup edges\_st \cup \{(r_1, n_1)\}$;
  return $(nodes, edges)$;

As in Definition 3.2, we left the algorithm parametric with respect to the equality relation $\triangleq$. This is done on purpose, because this relation is the only parameter that is subjective and thus, it is a good design decision to leave it open. For instance, a researcher can decide that two DOM nodes are equal if they have the same label and attributes. Another researcher can relax this restriction ignoring some attributes (i.e, the template can be the same, even if there are differences in colors, sizes, or even positions of elements. It usually depends on the particular use of the extracted template). Clearly, $\triangleq$ has a direct influence on the precision and recall of the technique. The more restrictive, the more precision (and less recall).

In our implementation, relation $\triangleq$ is defined with a ponderation that compares two nodes considering their HTML *id*, CSS classes, the number of children, their relative position in the DOM tree, and their HTML attributes. We refer the interested reader to our open and free implementation (`http://www.dsic.upv.es/~jsilva/retrieval/templates`) where relation $\triangleq$ is specified.

5 Implementation

The technique presented in this paper, including all the algorithms, has been implemented as a Firefox's plugin. In this tool, the user can browse on the Internet as usual. Then, when he/she wants to extract the template of a webpage, he/she only needs to press the "Extract Template" button and the tool automatically loads the appropriate linked webpages to form a CS, analyzes them, and extracts the template. The template is then displayed in the browser as any other webpage. For instance, the template extracted for the webpages in Figure 2 contains the whole webpage except for the part inside the dashed box.

5.1 Empirical evaluation

Several experiments were conducted with real, online webpages to provide a measure of the average performance regarding recall, precision, and the widely used F1 measure that combines both (see, e.g., [9] for a discussion on these metrics). Initially, we wanted to use a public standard collection of bench-

marks, but we are not aware of any public dataset for template extraction. In particular, we could not use the standard CleanEval suite [2] of content extraction benchmarks, because it contains a gold standard prepared for content extraction (each part of the webpages is labelled as *main-content* or *non-content*), but it is not prepared for template extraction. Then, we tried to use the same benchmark set as the authors of other template extraction papers. However, due to privacy restrictions, copyright, or unavailability[2] of the benchmarks we could not use a previous dataset. Therefore, we decided to produce a new suite of benchmarks. We have produced a new publicly accessible dataset, with an automatizable gold standard. This is one of the main contributions of our work. Any interested researcher can freely access and download our dataset from:

`http://www.dsic.upv.es/~jsilva/retrieval/templates`

The dataset is composed of a collection of web domains with different layouts and page structures. This allows us to study the performance of the techniques in different contexts (e.g., company websites, news articles, forums, etc.). To measure our technique, we randomly selected an evaluation subset. Table 1 summarizes the results of the performed experiments. First column contains the URLs of the evaluated website domains. For each benchmark, column `DOM nodes` shows the number of nodes of its whole DOM tree; column `Template` shows the number of nodes of the gold standard template; column `Retrieved` shows the number of nodes that were identified by the tool as the template; column `Recall` shows the number of correctly retrieved nodes divided by the number of nodes in the gold standard; column `Precision` shows the number of correctly retrieved nodes divided by the number of retrieved nodes; finally, column `F1` shows the F1 metric that is computed as $(2 * P * R)/(P + R)$ being P the precision and R the recall.

Benchmark	DOM nodes	Template	Retrieved	Recall	Precision	F1
www.felicity.co.uk	300 nodes	232 nodes	232 nodes	100 %	98,72 %	99,36 %
www.dsic.upv.es/~dinsa	241 nodes	74 nodes	74 nodes	100 %	90,24 %	94,87 %
www.engadget.com	1818 nodes	768 nodes	763 nodes	99,35 %	99,22 %	99,28 %
www.bbc.co.uk/news	2991 nodes	604 nodes	552 nodes	91,39 %	67,73 %	77,80 %
www.vidaextra.com	2331 nodes	1137 nodes	18 nodes	1,58 %	100 %	3,12 %
www.ox.ac.uk/staff	948 nodes	538 nodes	104 nodes	19,33 %	92,86 %	32,00 %
clinicaltrials.gov	543 nodes	389 nodes	378 nodes	97,17 %	96,92 %	97,05 %
en.citizendium.org	992 nodes	399 nodes	318 nodes	79,70 %	91,64 %	85,25 %
www.filmaffinity.com	1316 nodes	340 nodes	340 nodes	100 %	98,84 %	99,42 %
www.cnn.com	3860 nodes	192 nodes	148 nodes	77,08 %	98,67 %	86,55 %
www.lashorasperdidas.com	1822 nodes	553 nodes	252 nodes	45,57 %	100 %	62,61 %
labakeryshop.com	1368 nodes	403 nodes	175 nodes	43,42 %	96,15 %	59,83 %
www.dsic.upv.es/~jsilva/wwv2013	197 nodes	163 nodes	163 nodes	100 %	96,45 %	98,19 %
www.thelawyer.com	2708 nodes	949 nodes	742 nodes	78,19 %	76,50 %	77,33 %
www.us-nails.com	250 nodes	184 nodes	184 nodes	100 %	83,64 %	91,09 %
www.informatik.uni-trier.de	3083 nodes	117 nodes	8 nodes	6,84 %	100 %	12,8 %
www.wayfair.co.uk	1950 nodes	1507 nodes	697 nodes	46,25 %	99,57 %	63,16 %
catalog.atsfurniture.com	340 nodes	301 nodes	301 nodes	100 %	99,01 %	99,50 %
www.glassesusa.com	1952 nodes	1708 nodes	1659 nodes	97,13 %	99,70 %	98,40 %
www.mysmokingshop.co.uk	575 nodes	407 nodes	407 nodes	100 %	98,31 %	99,15 %
Average	1479 nodes	548 nodes	376 nodes	74,15 %	94,21 %	76,84 %

Table 1: Results of the experimental evaluation

Experiments reveal an average precision of more than 94%, and an average recall of almost 75% even though two benchmarks produced a recall under 7%. These benchmarks are particularly difficult ones that produce the same problem in previous techniques such as [17]. The problem in these benchmarks is that some webpages pointed by the main menu do not use the template (i.e., some webpages feature the menu, or the necessary links in some form, but they do not implement the template). Therefore, the intersection with these webpages produces an almost empty webpage, and this causes the low recall.

[2] Some authors answered that their benchmarks were not stored for future use, or that they did not save the gold standard.

5.2 Optimizations

Algorithm 1 computes a n-CS in a website. As previously explained, there are several combinations of webpages that form a CS. One could think that the more links the key page has, the better; so we could even think in calculating the maximal CS. Nevertheless, this is not a good idea. Firstly, because computing the maximal CS has an exponential cost. And, secondly, because our experiments reveal that increasing the size of the CS does not necessarily imply a better precision or recall.

In order to prove this, we repeated all our experiments with different sizes (1,2,3,4,5,6,7, and 8) for the CS in order to determine the best value. Results are shown in Table 2.

Size	Recall	Precision	F1	Loads
1	100 %	49,89 %	62,03 %	1
2	77,40 %	88,65 %	76,16 %	3,4
3	78,14 %	93,66 %	79,80 %	5,75
4	74,15 %	94,21 %	76,84 %	7,45
5	73,91 %	95,15 %	77,04 %	9,3
6	72,52 %	95,23 %	76,33 %	14
7	72,48 %	95,31 %	76,35 %	16,15
8	72,44 %	95,34 %	76,35 %	21

Table 2: Determining the ideal size of the complete subdigraph

This table summarizes several experiments. Each row is the average of repeating all the experiments in Table 1 with a different value for n in the n-CS searched by the algorithm. In particular, column `Size` represents the size of the CS that the algorithm tried to find in the websites. In the case that there did not exist a CS of the searched size, then the algorithm used the biggest CS with a size under the specified size (see Algorithm 1). Columns `Recall`, `Precision`, and `F1` has the same meaning as in Table 1. Finally, column `Loads` represents the average number of webpages loaded to extract the template.

Observe that recall is progressively reduced while increasing the size, whereas precision is progressively increased. This phenomenon was expected, because considering more webpages in the complete graph implies an intersection between more webpages, thus reducing the size of the template. We observed in the data that one of the benchmarks produced anomalous results. It positively affected the experiments when using size 3, and negatively affected the experiments when using a size higher than 3. The reason is that the fourth link in the menu pointed to a webpage not using the template. If we skip this benchmark, then almost all results are very similar when using a size higher than 3. We determined that a subdigraph of size 4 is the best option because it keeps almost the best F1 value, while being very efficient (a small number of webpages must be loaded to extract the template). Therefore, the results shown in Table 1 have been computed with a 4-CS. Our implementation can be configured to search for a subdigraph of any size (4,5,6, etc.). By default, it stops when a subdigraph of size 4 has been found.

Another important configuration parameter is related to the domain boundaries of the websites analyzed. It is possible that several webpages of different domains are mutually linked forming a CS. Sometimes this is even usual between the main webpages of different companies in an alliance. They all point to the others, e.g., with a set of logos. Nevertheless, the templates of the companies are often different. In fact, in our experiments, we did not find a shared template between different domains. Therefore, for efficiency reasons, external domains are omitted when computing the CS. In our implementation, the CS represents a set of intra-domain webpages linked by a common menu.

Our implementation and all the experimentation is public. All the information of the experiments, the source code of the benchmarks, the source code of the tool, and other material can be found at:

http://www.dsic.upv.es/~jsilva/retrieval/templates

6 Conclusions

Web templates are an important tool for website developers. By automatically inserting content into web templates, website developers, and content providers of large web portals achieve high levels of productivity, and they produce webpages that are more usable thanks to their uniformity.

This work presents a new technique for template extraction. The technique is useful for website developers because they can automatically extract a clean HTML template of any webpage. This is particularly interesting to reuse components of other webpages. Moreover, the technique can be used by other systems and tools such as indexers or wrappers as a preliminary stage. Extracting the template allows them to identify the structure of the webpage and the topology of the website by analyzing the navigational information of the template. In addition, the template is useful to identify pagelets, repeated advertisement panels, and what is particularly important, the main content.

Our technique uses the menus of a website to identify a set of webpages that share the same template with a high probability. Then, it uses the DOM structure of the webpages to identify the blocks that are common to all of them. These blocks together form the template. To the best of our knowledge, the idea of using the menus to locate the template is new, and it allows us to quickly find a set of webpages from which we can extract the template. This is especially interesting for performance, because loading webpages to be analyzed is expensive, and this part of the process is minimized in our technique. As an average, our technique only loads 7 pages to extract the template.

This technique could be also used for content extraction. Detecting the template of a webpage is very helpful to detect the main content. Firstly, the main content must be formed by DOM nodes that do not belong to the template. Secondly, the main content is usually inside one of the pagelets that are more centered and visible, and with a higher concentration of text.

For future work, we plan to investigate a strategy to further reduce the amount of webpages loaded with our technique. The idea is to directly identify the menu in the key page by measuring the density of links in its DOM tree. The menu has probably one of the higher densities of links in a webpage. Therefore, our technique could benefit from measuring the links–DOM nodes ratio to directly find the menu in the key page, and thus, a complete subdigraph in the website topology.

7 Acknowledgements

This work has been partially supported by the *Generalitat Valenciana* under grant PROMETEO/2011/052. David Insa was partially supported by the Spanish Ministerio de Eduación under FPU grant AP2010-4415. Salvador Tamarit was partially supported by research project POLCA, Programming Large Scale Heterogeneous Infrastructures (610686), funded by the European Union, STREP FP7.

References

[1] Ziv Bar-Yossef & Sridhar Rajagopalan (2002): *Template detection via data mining and its applications*. In: *Proceedings of the 11th International Conference on World Wide Web (WWW'02)*, ACM, New York, NY, USA, pp. 580–591, doi:10.1145/511446.511522. Available at http://doi.acm.org/10.1145/511446.511522.

[2] Marco Baroni, Francis Chantree, Adam Kilgarriff & Serge Sharoff (2008): *Cleaneval: a Competition for Cleaning Web Pages*. In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC'08)*, European Language Resources Association, pp. 638–643. Available at `http://www.lrec-conf.org/proceedings/lrec2008/summaries/162.html`.

[3] Radek Burget & Ivana Rudolfova (2009): *Web Page Element Classification Based on Visual Features*. In: *Proceedings of the 1st Asian Conference on Intelligent Information and Database Systems (ACIIDS'09)*, IEEE Computer Society, Washington, DC, USA, pp. 67–72, doi:10.1109/ACIIDS.2009.71. Available at `http://dx.doi.org/10.1109/ACIIDS.2009.71`.

[4] Eduardo Cardoso, Iam Jabour, Eduardo Laber, Rogério Rodrigues & Pedro Cardoso (2011): *An efficient language-independent method to extract content from news webpages*. In: *Proceedings of the 11th ACM symposium on Document Engineering (DocEng'11)*, ACM, New York, NY, USA, pp. 121–128, doi:10.1145/2034691.2034720. Available at `http://doi.acm.org/10.1145/2034691.2034720`.

[5] Soumen Chakrabarti (2001): *Integrating the Document Object Model with hyperlinks for enhanced topic distillation and information extraction*. In: *Proceedings of the 10th International Conference on World Wide Web (WWW'01)*, ACM, New York, NY, USA, pp. 211–220, doi:10.1145/371920.372054. Available at `http://doi.acm.org/10.1145/371920.372054`.

[6] W3C Consortium (1997): *Document Object Model (DOM)*. Available from URL: `http://www.w3.org/{DOM}/`.

[7] Adriano Ferraresi, Eros Zanchetta, Marco Baroni & Silvia Bernardini (2008): *Introducing and evaluating ukWaC, a very large web-derived corpus of english*. In: *Proceedings of the 4th Web as Corpus Workshop (WAC-4)*, pp. 47–54.

[8] David Gibson, Kunal Punera & Andrew Tomkins (2005): *The volume and evolution of web page templates*. In Allan Ellis & Tatsuya Hagino, editors: *Proceedings of the 14th International Conference on World Wide Web (WWW'05)*, ACM, pp. 830–839, doi:10.1145/1062745.1062763.

[9] Thomas Gottron (2007): *Evaluating content extraction on HTML documents*. In Vic Grout, Denise Oram & Rich Picking, editors: *Proceedings of the 2nd International Conference on Internet Technologies and Applications (ITA'07)*, National Assembly for Wales, pp. 123–132.

[10] Thomas Gottron (2008): *Content Code Blurring: A New Approach to Content Extraction*. In A. Min Tjoa & Roland R. Wagner, editors: *Proceedings of the 19th International Workshop on Database and Expert Systems Applications (DEXA'08)*, IEEE Computer Society, pp. 29–33, doi:10.1109/DEXA.2008.43.

[11] David Insa, Josep Silva & Salvador Tamarit (2013): *Using the words/leafs ratio in the DOM tree for content extraction*. *The Journal of Logic and Algebraic Programming* 82(8), pp. 311–325, doi:10.1016/j.jlap.2013.01.002.

[12] Christian Kohlschütter (2009): *A densitometric analysis of web template content*. In Juan Quemada, Gonzalo León, Yoëlle S. Maarek & Wolfgang Nejdl, editors: *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*, ACM, pp. 1165–1166, doi:10.1145/1526709.1526909.

[13] Christian Kohlschütter, Peter Fankhauser & Wolfgang Nejdl (2010): *Boilerplate detection using shallow text features*. In Brian D. Davison, Torsten Suel, Nick Craswell & Bing Liu, editors: *Proceedings of the 3th International Conference on Web Search and Web Data Mining (WSDM'10)*, ACM, pp. 441–450, doi:10.1145/1718487.1718542.

[14] Christian Kohlschütter & Wolfgang Nejdl (2008): *A densitometric approach to web page segmentation*. In James G. Shanahan, Sihem Amer-Yahia, Ioana Manolescu, Yi Zhang, David A. Evans, Aleksander Kolcz, Key-Sun Choi & Abdur Chowdhury, editors: *Proceedings of the 17th ACM Conference on Information and Knowledge Management (CIKM'08)*, ACM, pp. 1173–1182, doi:10.1145/1458082.1458237.

[15] Davi de Castro Reis, Paulo Braz Golgher, Altigran Soares Silva & Alberto Henrique Frade Laender (2004): *Automatic web news extraction using tree edit distance*. In: *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*, ACM, New York, NY, USA, pp. 502–511, doi:10.1145/988672.988740. Available at `http://doi.acm.org/10.1145/988672.988740`.

[16] Kuo Chung Tai (1979): *The Tree-to-Tree Correction Problem*. *Journal of the ACM* 26(3), pp. 422–433, doi:10.1145/322139.322143. Available at `http://doi.acm.org/10.1145/322139.322143`.

[17] Karane Vieira, Altigran S. da Silva, Nick Pinto, Edleno S. de Moura, João M. B. Cavalcanti & Juliana Freire (2006): *A fast and robust method for web page template detection and removal*. In: *Proceedings of the 15th ACM International Conference on Information and Knowledge Management (CIKM'06)*, ACM, New York, NY, USA, pp. 258–267, doi:10.1145/1183614.1183654. Available at `http://doi.acm.org/10.1145/1183614.1183654`.

[18] Tim Weninger, William Henry Hsu & Jiawei Han (2010): *CETR: Content Extraction via Tag Ratios*. In Michael Rappa, Paul Jones, Juliana Freire & Soumen Chakrabarti, editors: *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*, ACM, pp. 971–980, doi:10.1145/1772690.1772789.

[19] Lan Yi, Bing Liu & Xiaoli Li (2003): *Eliminating noisy information in Web pages for data mining*. In: *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data mining (KDD'03)*, ACM, New York, NY, USA, pp. 296–305, doi:10.1145/956750.956785. Available at `http://doi.acm.org/10.1145/956750.956785`.

Space Consumption Analysis by Abstract Interpretation: Reductivity Properties (High-level Work)

Manuel Montenegro Ricardo Peña Clara Segura

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

montenegro@fdi.ucm.es ricardo@sip.ucm.es csegura@sip.ucm.es

In a previous paper we presented an abstract interpretation-based static analysis for inferring heap and stack memory consumption in a functional language. The language, called *Safe*, is eager and first-order, and its memory management system is based on heap regions instead of the more conventional approach of having a garbage collector.

In this paper we concentrate on an important property of our analysis, namely that the inferred bounds are *reductive* under certain reasonable conditions. This means that by iterating the analysis using as input the prior inferred bound, we can get tighter and tighter bounds, all of them correct. In some cases, even the exact bound is obtained.

The paper includes several examples and case studies illustrating in detail the reductivity property of the inferred bounds.

Submitted to:
PROLE 2014

Improving the Deductive System DES with Persistence by Using SQL DBMS's (Original Work)

Fernando Sáenz-Pérez
Grupo de Programación Declarativa (GPD)
Dept. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
fernan@sip.ucm.es

This work presents how persistent predicates have been included in the in-memory deductive system DES by relying on external SQL database management systems. We introduce how persistence is supported from a user-point of view and the possible applications the system opens up, as the deductive expressive power is projected to relational databases. Also, we describe how it is possible to intermix computations of the deductive engine and the external database, explaining its implementation and some optimizations. Finally, a performance analysis is undertaken, comparing the system with current relational database systems.

1 Introduction

Persistence is one of the key features a database management system (DBMS) must fulfil. Such features are found in the well-known acronym ACID, where in particular D stands for Durability (i.e., persistence of data along different sessions) [14]. This way, updates in the database must be persistent in a non-volatile memory, as secondary storage (typically, the file system that the host operating system provides). Whereas persistence in relational DBMS's are given for granted, deductive databases have been traditionally implemented as in-memory database systems (as, e.g., DLV [8], XSB [15], bddbddb [7], Smodels [9], DES [12], ...) Some logic programming systems also allow persistent predicates, as Ciao Prolog does [3] (but only for the extensional part of the database).

In this work, we present an approach for adding predicate persistence to the deductive system DES (`des.sourceforge.net`) [12] relying on external SQL DBMS's via ODBC bridges. Enabling persistence leads to several advantages: 1) Persistent predicates with transparent handling, also allowing updates. Both the extensional (EDB, i.e., facts) and intensional (IDB, i.e., rules) databases can be persistent. 2) Interactively declare and undeclare predicates as persistent. Applications for this include database migration (cf. Section 3.4). 3) Mix both deductive solving and external SQL solving. On the one hand, the system takes advantage of the external database performance (in particular, table indexing is not yet provided by DES) and scalability. On the other hand, queries that are not supported in an external database (as hypothetical queries or recursive queries in some systems) can be solved by the deductive engine. So, one can use DES as a front-end to an external database and try extended SQL queries that add expressiveness to the external SQL language (cf. Sections 3.2 and 3.3). 4) Database interoperability. As several ODBC connections are allowed at a time, different predicates can be made persistent in different DBMS's, which allows interoperability among external relational engines and the local deductive engine, therefore enabling business intelligence applications (cf. Section 3.3). 5) Face applications with large amounts of data which do not fit in memory. Predicates are no longer limited by available memory (consider, for instance a 32bit OS with scarce memory); instead, persistent predi-

Submitted to:
Prole 2014

cates are using as much secondary storage as needed and provided by the underlying external database. Predicate size limit is therefore moved to the external database.

Nonetheless, a few deductive systems also integrated persistence or database connections, as DLV^{DB} [17], MyYapDB [6], and LDL++ [1]. One point that makes DES different from others is the ability to declare on-the-fly a given predicate as persistent and drop such declaration. This is accomplished by means of assertions, which together with a wide bunch of commands, make this system amenable for rapid experimenting and prototyping. In addition, since predicates can be understood as relations, and DES enjoys SQL, relational algebra (RA) and Datalog as query languages (SQL and RA are translated into Datalog), a persistent predicate can be used in any language and a given query can mix persistent predicates located at different databases. Those systems neither support full-fledged duplicates (including rules as duplicate providers), nor null-related operations, nor top-N queries, nor ordering metapredicates, nor several query languages accessing the same database (including Datalog, SQL, and extended relational algebra) as DES does [12]. Such features are required for supporting the already available expressiveness of current relational database systems. In addition, no system support hypothetical queries and views for decision support applications [10].

Organization of this paper proceeds as follows. Section 2 describes our approach to persistence, including in Subsection 2.6 a description of intermixing query solving as available as the result of embodying external DBMS access into the deductive engine, as well as some optimizations. Section 3 lists some applications for which persistence in a deductive system is well-suited. Next, Section 4 compares performance of this system w.r.t. DBMS's, and the extra work needed to handle persistent data. Finally, Section 5 summarizes some conclusions and points out future work.

2 Enabling Persistence

For a given predicate to be made persistent in an external SQL database, type information must be provided because SQL is strong-typed. As DES allows optional types for predicates (which are compatible with those of SQL) the system can take advantage of known type information for persistence. Note that, although the predicate to be made persistent has no type information, it may depend on others that do. This means that the declared or inferred type information for such a predicate must be consistent with other's types. To this end, a type consistency check is performed whenever a predicate is to be made persistent.

2.1 Declaring Persistence

We propose an assertion as a basic declaration for a persistent predicate, similar to [3]. The general form of a persistence assertion is as follows:

```
:- persistent(PredSpec,Connection)
```

where *PredSpec* is a predicate schema specification and the optional argument *Connection* is an ODBC connection identifier. *PredSpec* can be either the pattern *PredName*/*Arity* or *PredName*(*Schema*), where *Schema* is the predicate schema, specified as: $ArgName_1$:$Type_1$, ..., $ArgName_n$:$Type_n$, where $ArgName_i$ are the argument names and $Type_i$ are their (optional) types for an n-ary predicate ($n > 0$). If a connection name is not provided, the name of the current open database is used, which must be an ODBC connection. An ODBC connection is identified by a name defined at the OS level, and opening a connection in DES means to make it the current database and that any relation defined in a DBMS as

either a view or a table is allowed as any other relation (predicate) in the deductive local database `$des`. A predicate can be made persistent only in one external database.

Any rule belonging to the definition of a predicate `p` which is being made persistent is expected, in general, to involve calls to other predicates (either directly or indirectly). Each callee (such other called predicate) can be:

- An existing relation in the external database.
- A persistent predicate loaded in the local database.
- A persistent predicate not yet loaded in the local database.
- A non-persistent predicate.

For the first two cases, besides making `p` persistent, nothing else is performed when processing its persistence assertion. For the third case, a persistent predicate is automatically restored in the local database, i.e., it is made available to the deductive engine. For the fourth case, each non-persistent predicate is automatically made persistent if types match; otherwise, an error is raised. This is needed for the external database to be aware of a predicate only known by the deductive engine so far, as this database will be eventually involved in computing the meaning of `p`.

2.2 Implementing Persistence

In general, a predicate is defined by extensional rules (i.e., facts) and intensional rules (including both head and body). DES stores facts in a table and defines a view for the intensional rules. For a predicate `p`, a view with the same name as the predicate is created as the union of a table `p_des_table` (storing its extensional rules) and the equivalent SQL query for the remaining intensional rules. This table is created resorting to the type information associated to `p`. So, given that: a predicate `p` is composed of its extensional part P_{ex} and its intensional part P_{in}, each extensional rule in P_{in} is mapped to a tuple in the table `p_des_table`, $\| \texttt{p} \|_{SQL}$ is the meaning of the view `p` in an SQL system, and $\| \texttt{p} \|_{DL}$ is the meaning of the predicate `p` in the DES system, then:

$$\| \texttt{p} \|_{DL} = \| \texttt{p} \|_{SQL}$$

where the view `p` is defined by the SQL query:

`SELECT * FROM p_des_table UNION ALL` *DL_to_SQL(*P_{in}*)*

and *DL_to_SQL(*P_{in}*)* is the function that translates a set of rules P_{in} into an SQL query. To this end, we have resorted to Draxler's Prolog to SQL compiler [5] (PL2SQL from now on), which is able to translate a Prolog goal into an SQL query. Interfacing to this compiler is performed by the predicate `translate(+`*ProjectionTerm*`,+`*PrologGoal*`,-`*SQLQuery*`)`, where its arguments are, respectively, for: specifying the attributes that are to be retrieved from the database, defining the selection restrictions and join conditions, and representing the SQL query as a term. So, a rule composed of a head H and a body B can be translated into an SQL query S with the call `translate(`H`,`B`,`S`)`. Writing this as the function $dx\_translate(R_i)$, which is applied to a rule $R_i \equiv H_i : -B_i$ and returns its translated SQL query, and being $P_{in} = \{R_1, \ldots, R_n\}$, then:

$DL\_to\_SQL(P_{in})$ `=` $dx\_translate(R_1)$ `UNION ALL ...UNION ALL` $dx\_translate(R_n)$

PL2SQL is able to translate goals with conjunctions, disjunctions, negated goals, shared variables, arithmetic expressions in the built-in `is`, and comparison operations, among others. We have extended

$$
\begin{array}{lcl}
fact & ::= & p(c_1,\ldots,c_n) \\
rule & ::= & l \texttt{ :- } l_1,\ldots,l_n \\
l & ::= & p(a_1,\ldots,a_n) \\
l_i & ::= & l \mid \texttt{not } l \mid a_1 \Diamond a_2 \mid v \texttt{ is } e_1 \\
\Diamond & ::= & \texttt{=} \mid \texttt{\textbackslash=} \mid \texttt{<} \mid \texttt{=<} \mid \texttt{>} \mid \texttt{>=} \\
e_i & ::= & a \mid e_1 \blacklozenge e_2 \mid f(e_1,\ldots,e_n) \\
\blacklozenge & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
f & ::= & \texttt{sin} \mid \texttt{cos} \mid \texttt{abs} \mid \ldots
\end{array}
$$

p is a predicate symbol.	c_i are constants, $i \geq 1$.
l_i are literals, $i \geq 1$.	l is a term with depth 1.
v is a variable.	a_i are either variables or constants, $i \geq 1$.
e_i are arithmetic expressions.	*rule* is required to be safe and non recursive.

True type symbols and pipes denote terminals and alternatives, respectively.

Figure 1: Valid Inputs to PL2SQL$^+$

this compiler (PL2SQL$^+$ from now on) in order to deal with: Different, specific-DBMS-vendor code (including identifier delimiters and from-less SQL statements), the translation of facts, the mapping of some missing comparison operators, the inclusion of arithmetic functions to build expressions, and to reject both unsafe [18] and recursive rules. For instance, Access uses brackets as delimiters whereas MySQL uses back quotes. Also, Oracle does not support from-less SQL statements and requires a reference to the table `dual`, in contrast to other systems as PostgreSQL, which do not require it to deliver a one-tuple result (usually for evaluating expressions). The predicate `translate` does not deal with true goals as they would involve a from-less SQL statement. True goals are needed for translating facts, and so, we added support for this. We have included arithmetic functions for the compilation of arithmetic expressions, including trigonometric functions (`sin`, `cos`, ...), and others (`abs`, ...). However, the support of such functions depend on whether the concrete SQL system supports them as well. PL2SQL requires safe rules but it does not provide a check, so that we have included such a check to reject unsafe rules. Recursive rules are not translated because not all DBMS's support recursive SQL statements (further DES releases might deal with specific code for recursive rules for particular DBMS's supporting recursion, as DB2 and SQL Server). Figure 1 summarizes the syntax of valid inputs to PL2SQL$^+$ which are eventually represented as SQL statements. Note that propositional predicates are not supported because relational databases require relations with arity greater than 0.

DES preprocess Datalog rules before they can be eventually executed. Preprocessing includes source-to-source transformations for translating several built-ins, including disjunction, outer joins, relational algebra division, top-N queries and others. Rules sent to the Prolog to SQL compiler are the result of these transformations, so that several built-ins that are not supported by [5] can be processed by DES, as outer joins (left, right and full). As well, there are other built-ins that PL2SQL can deal with but which are not passed by DES up to now (as aggregates and grouping).

Non-valid rules for PL2SQL$^+$ but otherwise valid for DES are kept in the local database for their execution. In such a case, the deductive engine couples its own processing with the processing of the external database in the following way. Let a predicate `p` be defined by a set of rules S that can be externally processed and other set of rules D that cannot. Then, the meaning of `p` is computed as the union of the meanings of both sets of rules:

$\| \mathtt{p} \| = \| S \|_{SQL} \cup \| D \|_{DL}$

Rules in S are therefore not included in P_{in} in the call to `DL_to_SQL` as described above, and they are otherwise stored as regular in-memory Datalog rules and processed by the deductive engine. Therefore, all the deductive computing power is preserved when either the external DBMS lacks some features as, e.g., recursion (e.g., MySQL and MS Access), or a predicate contains some non-valid rules for PL2SQL$^+$.

2.3 An Example

As an example, let's consider the predicate `ancestor`, the DBMS MySQL, and a table `father` already created and populated in this external database.

MySQL:

```
CREATE TABLE father(father VARCHAR(20),child VARCHAR(20));
INSERT INTO father VALUES('tom','amy');
...
```

DES:

```
:-type(mother(mother:string,child:string)).
mother(grace,amy).
...

:-type(parent(parent:string,child:string)).
parent(X,Y) :- father(X,Y) ; mother(X,Y).

:-type(ancestor(ancestor:string,descendant:string)).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Then, if we submit the assertion `:-persistent(ancestor/2)` when the current opened database is MySQL, we get the following excerpt of the DES verbose output:

```
Warning: Recursive rule cannot be transferred to external database
  (kept in local database for its processing):
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
Info: Predicate mother/2 made persistent.
Info: Predicate parent/2 made persistent.
Info: Predicate ancestor/2 made persistent.
```

Recalling Section 2.1, declaring the persistence of `ancestor/2` involves to make persistent both `mother/2` and `parent/2` because, in particular, the first rule of `ancestor/2` includes a call to `parent/2`, and the second call of `parent/2` is to `mother/2`. Even when `parent/2` includes a call to `father/2`, the latter predicate is not made persistent because there exist the table `father/2` in the external database already. Note that if the current database was `$des`, then a warning indicating that the predicate `father/2` had been issued, because this external relation is not visible to the local database unless the external connection is the current database. The resulting views[1] after processing the assertion are:

```
CREATE VIEW mother(mother,child) AS
  SELECT * FROM mother_des_table;

CREATE VIEW parent(parent,child) AS
  (SELECT * FROM parent_des_table) UNION ALL
```

[1] They can be displayed, for instance, with the command `/dbschema $des`.

```
  (SELECT rel1.mother,rel1.child FROM mother AS rel1) UNION ALL
  (SELECT rel1.father,rel1.child FROM father AS rel1);

CREATE VIEW ancestor(ancestor,descendant) AS
  (SELECT * FROM ancestor_des_table) UNION ALL
  (SELECT rel1.parent,rel1.child FROM parent AS rel1);
```

Note that, on the one hand, and as a difference with other systems as DLV^{DB}, these views are not materialized. On the other hand, DES allows to project such intensional rules to the external database by contrast to Ciao, which only project extensional rules.

Processing a call either to `father/2`, or `mother/2` or `parent/2` is computed by the external database. However, a call to `ancestor/2` is processed both by the external database because of its first rule involving a call to `parent`, and by the local deductive engine due to the local rule (the recursive one which cannot be processed by MySQL), as it will be explained in Section 2.6.

All intensional rules (both valid and non-valid inputs to PL2SQL$^+$) of a persistent predicate `p` are externally stored as metadata information in a table named `p_des_metadata` to allow to recover original rules when removing a persistence assertion (cf. Section 2.5). For instance, the contents of this table for `parent` is [2]:

```
parent_des_metadata('parent(X,Y):-father(X,Y).').
parent_des_metadata('parent(X,Y):-mother(X,Y).').
```

Whilst the contents of `mother_des_table` are its extensional rules (the facts `mother(grace,amy)`, ...), the contents of `parent_des_table` and `ancestor_des_table` are empty (unless a fact is asserted in any of the corresponding predicates). Note that, as `father` is a table in the external database, if we *assert* a new tuple *t* for it, it will be only loaded in the local database, instead of externally stored if it was a persistent predicate[3]. In both cases, the query `father(X,Y)` would return the same tuples (either for the table or for the persistent predicate), but upon restoring persistence of `ancestor/2`, the tuple *t* would not be restored for the table `father`.

2.4 Updating Persistent Predicates

Updating a persistent predicate `p` is possible with the commands `/assert` and `/retract`, which allow to insert and delete a rule, respectively, and their counterpart SQL statements `INSERT` and `DELETE`, which allow to insert and delete, respectively, a batch of tuples (either extensionally or intensionally). Implementing the update of the IDB part of a persistent predicate amounts to retrieve the current external view corresponding to the persistent predicate, drop it, and create a new one with the update. The update of the EDB part (insert or delete a tuple) is simply performed to the external table with an appropriate SQL statement (`INSERT INTO ...` or `DELETE FROM ...`). Each update is tuple-by-tuple, even when batch updates via select statements are processed. For each update, if constraint checking is enabled, any strong constraint defined at the deductive level is checked.

Note that the view update problem is not an issue because our approach to insertions and deletions of tuples in a persistent predicate amounts to modify the extensional part of the predicate, which is stored in a table. This is a different approach to DBMS's where a relation defined by a view only consists of an intensional definition, so that trying to update a view involves updating the relations (other views and tables) it depends on, and this can be done is some situations but not in general.

[2] Note that as a result of DES preprocessing, the rule with the disjunction has been translated into two rules.

[3] Of course, *inserting* a tuple in the external table will store it in the DBMS.

2.5 Restoring and Removing a Persistent Predicate

Once a predicate p has been made persistent in a given session, the state of p can be restored in a next session (i.e., after the updates –assertions or retractions– on p)[4]. It is simply done by submitting again the same assertion as used to make p persistent for the first time. Note, however, that if there exists any rule for p in the in-memory database for p already, it will not be removed but stored as persistent in the external database.

Also, a given predicate can be made non-persistent by dropping its assertion, as, e.g.:

```
DES> /drop_assertion :-persistent(p(a:int),mysql)
```

This retrieves all the facts stored in the external database, stores them back in the in-memory database, removes them from the external database, and the original rules, as they were asserted (in its compiled Datalog form) are recovered from the table p_des_metadata. The view and tables for predicate p are dropped.

2.6 Intermixing Query Solving

As already introduced, persistence enables to couple external DBMS processing with deductive engine processing. DES implements a top-down-driven, bottom-up fixpoint computation with tabling [12], which follows the ideas found in [13, 4, 16]. This mechanism is implemented as described in [11]. In particular, the predicate solve_goal solves a goal (built-ins and user-defined predicates). The following clause of this predicate is responsible of using program rules to solve a goal corresponding to a user predicate (where arguments which are not relevant for illustration purposes have been removed):

```
solve_goal(G) :- datalog((G:-B),_Source), solve(B).
```

This predicate selects a program rule matching the goal via backtracking and solves the rule body as a call to the the predicate solve. Such program rules are loaded in the dynamic predicate datalog.

In order to allow external relations to be used as user predicates, this dynamic predicate is overloaded with the following clause, which in turn calls datalog_rdb:

```
datalog(Rule,rdb(Connection)) :-
  datalog_rdb(Rule,rdb(Connection)).

datalog_rdb(R,Source) :-
  datalog_rdb_single_np(R,Source) ; % Single, non-persistent relation
  datalog_rdb_all_np(R,Source)    ; % All the non-persistent relations
  datalog_rdb_single_p(R,Source)  ; % Single, persistent predicate
  datalog_rdb_all_p(R,Source).      % All persistent predicates
```

The predicate datalog_rdb identifies two possible sources: non-persistent and persistent predicates. Also, it identifies whether a particular predicate is called or otherwise all predicates are requested. In the last case, all external relations must be retrieved, and predicates datalog_rdb_all_np and datalog_rdb_all_p implement this via backtracking. The (simplified) implementation of the predicate datalog_rdb_single_p (a single, concrete, persistent predicate) for an external ODBC connection Conn is as follows:

```
datalog_rdb_single_p(R,RuleId,rdb(Conn)) :-
  my_persistent(Connection,TypedSchema),
  functor(TypedSchema,TableName,Arity),
```

[4] Cf. transaction logic [2] to model states in logic programming.

```
  R =.. [Name|Columns],
  length(Columns,Arity),
  schema_to_colnames(TypedSchema,ColNames),
  sql_rdb_datasource(Conn,Name,ColNames,Columns,SQLstr),
  my_odbc_dql_query_fetch_row(Conn,SQLstr,Row),
  Row=..[_AnswerRel|Columns].
```

The predicate `sql_rdb_datasource` builds an SQL statement which returns rows for a relation under a connection matching the input column values (`Columns` is the list of variables and/or constants for the query). As an example, the query `ancestor(A,amy)` for the example in Section 2.3 generates the following SQL statement (notice that the identifier delimiters in this DBMS do not follow standard SQL):

```
SELECT * FROM 'ancestor' WHERE 'descendant'='amy'
```

The predicate `my_odbc_dql_query_fetch_row` returns rows, one-by-one, via backtracking for this SQL statement. Note that, for this simple example, row filtering is performed by the external engine.

Recall that this persistent predicate consists of two program rules:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

The first one was loaded in the external database as the view:

```
CREATE VIEW ancestor(ancestor,descendant) AS
  (SELECT * FROM ancestor_des_table) UNION ALL
  (SELECT rel1.parent,rel1.child FROM parent AS rel1);
```

and the second one was loaded in the local deductive database, as the dynamic clause:

```
datalog((ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y)),source)
```

So, the fixpoint mechanism uses with the call to `datalog` both the non-recursive rule from the external database via `datalog_rdb_single_p`, and the recursive rule via the dynamic clause. Concluding, the predicate `datalog` provides to the deductive query solving not only the rules which are local, but also the rules which are externally stored and processed, retrieved via the predicate `datalog_rdb`, therefore enabling intermixed query solving.

2.7 Fixpoint Optimizations

We list some already implemented optimizations which are key to avoid retrieving the same tuple from the external database several times due to fixpoint iterations. They can be independently enabled and disabled with commands to test their impact on performance.

- **Complete Computations**. Each call during the computation of a stratum is remembered in addition to its outcome (in the answer table). Even when the calls are removed in each fixpoint iteration, most general ones do persist as a collateral data structure to be used for saving computations should any of them is called again during either computing a higher stratum or a subsequent query solving. If a call is marked as a completed computation, it is not even tried if called again. This means the following two points: 1) During the computation of the memo function, calls already computed are not tried to be solved again, and only the entries in the memo table are returned. 2) Moreover, computing the memo function is completely avoided if a subsuming already-computed call can be found. In the first case, that saves solving goals in computing the memo function. In the second case, that completely saves fixpoint computation.

- **Extensional Predicates**. There is no need to iteratively compute extensional predicates and, therefore, no fixpoint computation is needed for them. They are known from the predicate dependency graph simply because they occur in the graph without incoming arcs. For them, a linear fetching is enough to derive their meanings.
- **Non-Recursive Predicates**. Memoization comes at the cost of maintaining a cache which can be wasteful in some cases. A top-level goal involving non-recursive predicates are computed by only caching the top-level goal, avoiding memorizing dependent calls. This allows a fast solving by looking for all the answers of the goal, and finally storing the results in the memo table.

3 Applications

This section lists several applications derived from supporting persistence in DES as it includes some features which are not available in external DBMS's, such as hypothetical queries, extended recursion, and intermixed query solving.

3.1 Database Interoperability

Persistence allows for database interoperability as each persistent predicate is mapped to an ODBC connection and several connections can be opened simultaneously. First scenario is for a persistent predicate `p` in a given connection and opening another connection from another database. Then, both the predicate `p` and the relations defined in the latter connection are visible for the deductive database. This is in contrast to other systems (as, e.g., DLV^{DB}) that need to explicitly state what relations from the external database are visible. Here, no extra effort is needed. Second scenario is for several persistent predicates which are mapped to different connections. As they are visible for the deductive engine, all of them can be part of a query solved by the deductive engine. Recall that any external view will be still processed by the external DBMS.

3.2 Extending DBMS Expressivity

The more expressive SQL and Datalog languages as provided by DES can improve the expressiveness of the external database when acting as a front-end. For instance, let's consider MySQL, which does not support recursive queries up to its current version 5.6. The following predicate can be made persistent in this DBMS even when it is recursive:

```
DES> :-persistent(path(a:int,b:int),mysql)
DES> /assert path(1,2)
DES> /assert path(2,3)
DES> /assert path(X,Y):-path(X,Z),path(Z,Y)
Warning: Recursive rule cannot be transferred to external database (kept
    in local database for its processing):
path(X,Y) :- path(X,Z), path(Z,Y).
DES> path(X,Y)
{ path(1,2), path(1,3), path(2,3) }
```

Here, non-recursive rules are stored in the external database whereas the recursive one is kept in the local database. External rules are processed by MySQL and local rules by the deductive engine. Though the recursive rule is not externally processed, it is externally stored as metadata, therefore ensuring its persistence between sessions.

In addition to Datalog, DES includes support for SQL for its local deductive database. To this end, on the one hand, SQL data definition statements are executed and metadata (as the name and type of table fields) is stored as assertions. On the other hand, SQL queries are translated into Datalog and executed by the deductive engine. The supported SQL dialect includes features which are not found in current DBMS's, as non-linear recursive queries, hypothetical views and queries, and the relational algebra division operator. Therefore, DES is able to compute more queries than a DBMS: For instance, neither MS SQL Server nor IBM DB2 allow cycles in a path without compromising termination. Also, recursive and stratifiable SQL queries do not fully allow `EXCEPT` such in MS SQL Server and IBM DB2. Another limitation is linear recursion: The above rules cannot be expressed in any DBMS as there are several recursive calls. To name another, `UNION ALL` is enforced in those SQL's, so that just `UNION` (discarding duplicates) is not allowed. For instance, the following recursive query is rejected in any current commercial DBMS, but accepted by DES:

```
DES> CREATE TABLE edge(a int, b int);
DES> INSERT INTO edge VALUES (1,2),(2,3),(1,3);
DES> :-persistent(edge/2,mysql).
DES> :-persistent(path(a:int,b:int),mysql).
DES> WITH RECURSIVE path(a, b) AS
  SELECT * FROM edge
  UNION --Discard duplicates (ALL not required)
  SELECT p1.a,p2.b FROM path p1, path p2 WHERE p1.b=p2.a
SELECT * FROM path;
Warning: Recursive rule cannot be transferred to external database
  (kept in local database for its processing):
path_2_1(A,B) :- path(A,C), path(C,B).
answer(path.a:number(integer), path.b:number(integer)) ->
{ answer(1,2), answer(1,3), answer(2,3) }
```

In this example, `edge` becomes a Datalog typed (and populated) relation because it is defined with the DES SQL dialect in the local deductive database, and it has been made persistent, as well as `path` (which is also typed because of the persistence assertion, but not populated). The `WITH` statement allows to declare temporary relations. In this case, the result of the compilation of the SQL query definition of `path` are temporary Datalog rules which are added to the persistent predicate `path` (note that the recursive part is not transferred to the external database):

```
path(A,B) :- distinct(path_2_1(A,B)).
path_2_1(A,B) :- edge(A,B).
path_2_1(A,B) :- path(A,C), path(C,B).
```

and the SQL query `SELECT * FROM path` is compiled to:

```
answer(A,B) :- path(A,B).
```

After executing the goal `answer(A,B)` for solving the SQL query, the temporary Datalog rules are removed. Adding `ALL` to `UNION` to the same query for keeping duplicates makes to include the tuple `answer(1,3)` twice in the result.

3.3 Business Intelligence

Business intelligence refers to systems which provide decision support [19] by using data integration, data warehousing, analytic processing and other techniques. In particular, one of these techniques refer to "what-if" applications. DES also supports a novel SQL feature: Hypothetical SQL queries. Such

queries are useful, for instance, in decision support systems as they allow to submit a query by assuming some knowledge which is not in the database. Such knowledge can be either new data assumed for relations (both tables and views) and also new production rules. For example, and following the above system session, the tuple `(3,1)` is assumed to be in the relation `path`, and then this relation is queried:

```
DES> ASSUME (SELECT 3,1 IN path(a,b)) SELECT * FROM path;
answer(path.a:number(integer),path.b:number(integer)) ->
{ answer(1,1), answer(1,2), answer(1,3), answer(2,1), answer(2,2),
  answer(2,3), answer(3,1), answer(3,2), answer(3,3) }
```

As an example of adding a production rule, let's suppose a relation `flight` and a view `connect` for locations connected by direct flights:

```
DES> CREATE TABLE flight(ori STRING, dest STRING, duration INT);
DES> INSERT INTO flight VALUES ('Madrid','Paris',90),
       ('Paris','Oslo',100), ('Madrid','London',110);
DES> CREATE VIEW connect(ori,dest) AS SELECT ori,dest FROM flight;
DES> :-persistent(connect/2,access) -- This also makes 'flight' persistent
DES> SELECT * FROM connect;
answer(connect.ori:string(real),connect.dest:string(real)) ->
{ answer('Madrid','London'), answer('Paris','Oslo'),
  answer('Madrid','Paris') }
```

Then, if we assume that connections are allowed with transits, we can submit the following hypothetical query (where the assumed SQL statement is recursive):

```
DES> ASSUME
      (SELECT flight.ori,connect.dest
       FROM flight,connect
       WHERE flight.dest = connect.ori)
     IN
       connect(ori,dest)
     SELECT * FROM connect;
answer(connect.ori:string(real),connect.dest:string(real)) ->
{ answer('Madrid','London'),answer('Madrid','Oslo'),
  answer('Madrid','Paris'), answer('Paris','Oslo')}
```

Also, several assumptions for different relations can be defined in the same query.

3.4 Migrating Data

Once a predicate has been made persistent in a given connection, dropping its persistent assertion retrieves all data and schema from the external database into the in-memory Prolog database. A successive persistent assertion for the same predicate in a different connection dumps it to the new external database. These two steps, therefore, implement the migration from one database to another, which can be of different vendors. For instance, let's consider the following session, which dumps data from MS Access to MySQL:

```
DES> :-persistent(p(a:int),access)
DES> /drop_assertion :-persistent(p(a:int),access)
DES> :-persistent(p(a:int),mysql)
```

4 Performance

In this section we analyze how queries involving persistent predicates perform w.r.t. native SQL queries, and the overhead caused by persistence w.r.t. the in-memory (Prolog-implemented) database.

As relational database systems, three widely-used systems have been chosen with a default configuration: The non-active desktop database MS Access (version 2003 with ODBC driver 4.00.6305.00), the mid-range, open-source Oracle MySQL (version 5.5.9 with ODBC driver 5.01.08.00), and the full-edged, commercial IBM DB2 (version 10.1.0 with ODBC driver 10.01.00.872). All times are given in milliseconds and have been run on an Intel Core2 Quad CPU at 2.4GHz and 3GB RAM, running Windows XP 32bit SP3. Each test has been run 10 times, the maximum and the minimum numbers have been discarded, and then the average has been computed. Also, as Access quickly fragments the single file it uses for persistence, and this heavily impacts performance, each running of the benchmarks in this system is preceded by a defragmentation (though, the time for performing this has not been included in the numbers). All optimizations, as listed in Section 2.6, are enabled.

Some results are collected in Table 1. The tests consist of, first, inserting 1,000 tuples in a relation with a numeric field (columns with heading $Insert_n$ and $Insert_p$, for the native and persisted queries, respectively). The Datalog commands are `/assert t(`i`)` and the SQL update queries are `INSERT INTO t VALUES(`i`)` ($1 \leq i \leq 1{,}000$).

Then, 1,000 select queries are issued (columns $Select_n$ and $Select_p$). The i-th select query asks for the i-th value stored in the table, so that all values are requested by independent queries. The Datalog query is `t(`i`)` and the SQL select query is `SELECT a FROM t WHERE a=`i ($1 \leq i \leq 1{,}000$).

Next, a single query which computes the Cartesian product of the table with itself is submitted (columns $Product_n$ and $Product_p$), therefore providing one million tuples in the result set. The Datalog queries are `t(X),t(Y)` and the SQL select queries are `SELECT * FROM t AS t1,t AS t2`.

First line below headings of this table collects the results of the in-memory deductive database DES (Datalog commands and queries), with no persistence. The next three lines in the block with subscripts n (referred to as 'block n' from now on) show the results for the native queries in each DBMS (SQL `INSERT` and `SELECT` queries). The three lines in the block with subscripts p (referred to as 'block p' from now on) show the results for the Datalog commands (`/assert`) and queries (`t(`i`)` and `t(X),t(Y)`) when the relation `t` has been made persistent in each external DBMS.

Then, this table allows, first, to compare the in-memory, state-less system DES w.r.t. the relational, durable DBMS's (ratio values enclosed between parentheses in block n). Second, to examine the overhead of persistence by confronting the results in the line DES and the results in the block p for each DBMS (first ratio value enclosed between parentheses in the table). And, third, to compare the results of DES as a persistent database w.r.t. each DBMS for dealing with the same actions (inserting and retrieving data), by confronting the results in block p and block n for each DBMS (second ratio value enclosed between parentheses in the table).

For the select queries, we focus on retrieving to the main memory the result but without actually

System	$Insert_n$	$Select_n$	$Product_n$	$Insert_p$	$Select_p$	$Product_p$
DES 3.2	359	773	3,627			
Access	439 (1.22)	1,014(1.31)	7,303(2.01)	1,102 (3.07⋄2.51)	2,138(2.77⋄2.11)	17,270(4.76⋄2.36)
MySQL	9,950(27.72)	1,160(1.50)	13,183(3.63)	10,279(28.63⋄1.03)	2,364(3.06⋄2.04)	22,305(6.15⋄1.69)
DB2	1,264 (3.52)	1,018(1.32)	9,057(2.50)	1,869 (5.21⋄1.48)	2,260(2.92⋄2.22)	18,637(5.14⋄2.06)

Table 1: Results for in-memory DES, DBMS's and Persistent Predicates

displaying it in order to elide the display time. For the deductive database, this means that each tuple in the result is computed and stored in the answer table but it is not displayed. For the relational databases, this means that a single ODBC cursor connection is used for a single query and each tuple in its result is retrieved to main memory, but not displayed.

With respect to the native queries, a first observation is that insertions (column $Insert_n$) in the in-memory deductive database are, as expected, faster than for DBMS's. However, Access is very fast as it is more oriented towards a file system (it is not an active database) and its time is comparable to that of DES (Access is only 22% slower). Another observation is that MySQL takes much more time for updates than DB2 (8 times slower) and Access (22.6 times slower), but it performs close to them for the batch of 1,000 select queries. (This behaviour can be also observed in queries to persistent data.) A third observation is that computations for select operations (columns *Select* and *Product*) in the in-memory deductive database are, in general, faster than in persistent (relational) databases. While for 1,000 queries in Datalog there is a speed-up of up to 1.5, in the single query this grows up to 3.6 (MySQL).

Queries to persistent data show two factors: 1) The performance of queries involving persistence w.r.t. their counterpart native SQL queries, and 2) The overhead caused by persistence in the deductive system for the different DBMS's. With respect to factor 1, by comparing native queries to queries to persistent data, we observe that the cost for inserting tuples by using a persistent predicate w.r.t. a native SQL `INSERT` statement ranges from a negligible ratio of 1.03 (MySQL) to 2.51 (Access). Also, the overhead for computing 1,000 queries with a Datalog query on a persistent predicate w.r.t. its counterpart native SQL select statement, is around 2 times for all DBMS's. And for the product, the minimum ratio is 1.69 for MySQL and 2.36 for Access. With respect to factor 2, insertions require a ratio from 3.07 to 5.21 for Access and DB2, respectively, whereas for DB2 a huge ratio of 28.63 is found. Managing individual insert statements via cursor connections is hard in this case. However, the overhead comes from the connection itself as the code to access the different external databases is the same. The select queries perform quite homogeneously with ratios from 2.76 to 3.06, in accordance to factor 1. Last, for the Cartesian product, the ratio ranges from 4.76 to 6.15.

Finally, Table 2 shows the cost for creating and removing persistence for each DBMS. The column *Create* shows the time for creating a persistent predicate where its 1,000 tuples are in the in-memory database. This amounts to store each in-memory tuple in the external database, so that numbers are similar to that of the column *Insert*. Dropping the persistent assertion, as shown in column *Drop*, takes a small time. Recall that this operation also retrieve the 1,000 tuples to the in-memory database. The difference between the cost of creating and dropping the assertion lies in that the former submits 1,000 SQL queries while the latter submits a single SQL query. Thus, the cost of opening and closing cursor connections is therefore noticeable.

DBMS	*Create*	*Drop*
Access	1,256	31
MySQL	10,523	74
DB2	1,926	172

Table 2: Creating and Removing Persistence

5 Conclusions

This paper has shown how persistence is supported by a tabled-based deductive system. This work includes extended language features that might be amenable to try even projected to such external databases. Although this system was targeted at teaching and not to performance, some numbers have been taken to assess its applicability. When comparing the times taken by the queries relating persistent predicates w.r.t. their counterpart native SQL queries, ratios from 1.03 up to 2.51 are got, which overall shows the overhead of using the deductive persistent system w.r.t. the SQL systems. When comparing the times taken by the queries relating persistent predicates w.r.t. their counterpart in-memory queries, higher ratios have been found, from 2.76 up to 6.15, and an extreme case of 28.63 due to the costly insertions through the ODBC bridge. These results suggest that the cost of persistence might be worthwhile depending on the DBMS and the application.

Differences between this system and others can be highlighted, besides those which were already noted in the introduction. For instance, predicates in DLV^{DB} are translated into materialized relations, i.e., a predicate is mapped to a table and the predicate extension is inserted in this table, which opens up the 'view' maintenance problem. Ciao Prolog is only able to make the extensional part of a predicate to persist, disabling the possibility of surrogating the solving of views for intensional rules. MyYapDB (for *unixes) is not understood as implementing persistence, instead, it allows to connect to the external MySQL DBMS, making external relations available to YAP as if they were usual predicates. This is similar to what DES does simply by opening an ODBC connection, which automatically makes visible all the external relations (not only in MySQL but for any other DBMS and OS). LDL++ was retired in favor of DeALS, and currently there are no information about its connection to external databases, though in [1] such a connection was very briefly described for the former.

As for future work, built-ins supported by the compiler [5] but not passed by DES can be included in forthcoming releases. Also, query clustering can be useful (cf. [3]), i.e., identifying those complex subgoals that can be mapped to a single SQL query, therefore improving the results for queries as the Cartesian product, by reducing the number of cursors. Rules with linear recursive queries supported by the external DBMS can be allowed to be projected. Since the deductive engine is not as efficient as others [15], it can be improved or replaced with an existing one but upgraded to deal with extra features (as nulls and duplicates). Finally, the current implementation has been tested for several DBMS's, including Access, SQL Server, MySQL, and DB2. Although the connection to such external databases is via the ODBC bridge which presents a common interface to SQL, some tweaks depending on the particular SQL dialect should be made in order to cope with other DBMS's.

Acknowledgements

This work has been partially supported by the Spanish projects STAMP (TIN2008-06622-C03-01), Prometidos-CM (S2009TIC-1465), GPD (UCM-BSCH-GR35/10-A-910502), and the Department Ingeniería del Software e Inteligencia Artificial at University Complutense of Madrid. Also thanks to the anonymous referees who helped in improving this paper and the system DES.

References

[1] Faiz Arni, KayLiang Ong, Shalom Tsur, Haixun Wang & Carlo Zaniolo (2003): *The Deductive Database System LDL++*. *TPLP* 3(1), pp. 61–94.

[2] Anthony J. Bonner & Michael Kifer (1994): *An Overview of Transaction Logic*. *Theoretical Computer Science* 133.

[3] Jesús Correas, J. M. Gómez, Manuel Carro, Daniel Cabeza & Manuel V. Hermenegildo (2004): *A Generic Persistence Model for (C)LP Systems (and Two Useful Implementations)*. In Bharat Jayaraman, editor: *PADL*, *LNCS* 3057, pp. 104–119.

[4] Suzanne W. Dietrich (1987): *Extension Tables: Memo Relations in Logic Programming*. In: *IEEE Symp. on Logic Programming*, pp. 264–272.

[5] Cristoph Draxler (1992): *A Powerful Prolog to SQL Compiler*. CIS-Bericht-, Univ. München, Centrum für Informations-und Spracverarbeitung.

[6] Michel Ferreira & Ricardo Rocha (2004): *The MyYapDB Deductive Database System*. In: *JELIA*, pp. 710–713. Available at `http://dx.doi.org/10.1007/978-3-540-30227-8_63`.

[7] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin & Christopher Unkel (2005): *Context-sensitive program analysis as database queries*. In: *PODS*, pp. 1–12.

[8] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri & Francesco Scarcello (2006): *The DLV system for knowledge representation and reasoning*. *ACM Transactions on Computational Logic* 7(3), pp. 499–562. Available at `http://doi.acm.org/10.1145/1149114.1149117`.

[9] Ilkka Niemelä, Patrik Simons & Tommi Syrjänen (2000): *Smodels: A System for Answer Set Programming*. In: *Proc. of the 8th Intl. Workshop on Non-Monotonic Reasoning*. Available at `http://arxiv.org/abs/cs.AI/0003033`.

[10] Fernando Sáenz-Pérez (2013): *Implementing Tabled Hypothetical Datalog*. In: *Proceedings of the 25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI'13*.

[11] Fernando Sáenz-Pérez (2013): *Tabling with Support for Relational Features in a Deductive Database*. *Electronic Communications of the EASST* 55, pp. 1 – 16.

[12] Fernando Sáenz-Pérez, Rafael Caballero & Yolanda García-Ruiz (2011): *A Deductive Database with Datalog and SQL Query Languages*. In Hongseok Yang, editor: *APLAS*, *LNCS* 7078, Springer, pp. 66–73. Available at `http://dx.doi.org/10.1007/978-3-642-25318-8_8`.

[13] C.-F. Shih & Suzanne W. Dietrich (1991): *Extension Table Evaluation of Datalog Programs with Negation*. In: *Proc. of the IEEE International Phoenix Conference on Computers and Communications*, AZ, Scottsdale, pp. 792–798.

[14] Avi Silberschatz, Henry F. Korth & S. Sudarshan (2010): *Database System Concepts (Sixth Edition)*. McGraw-Hill New York.

[15] Terrance Swift & David Scott Warren (2012): *XSB: Extending Prolog with Tabled Logic Programming*. *TPLP* 12(1-2), pp. 157–187. Available at `http://dx.doi.org/10.1017/S1471068411000500`.

[16] Hisao Tamaki & Taisuke Sato (1986): *OLDT Resolution with Tabulation*. In: *Third International Conference on Logic Programming*, pp. 84–98.

[17] Giorgio Terracina, Nicola Leone, Vincenzino Lio & Claudio Panetta (2008): *Experimenting with recursive queries in database and logic programming systems*. *Theory and Practice of Logic Programming* 8(2), pp. 129–165.

[18] Jeffrey D. Ullman (1988): *Database and Knowledge-Base Systems, Vols. I (Classical Database Systems) and II (The New Technologies)*. Computer Science Press.

[19] Hugh J. Watson & Barbara H. Wixom (2007): *The Current State of Business Intelligence*. *Computer* 40(9), pp. 96–99.

XQOWL: An Extension of XQuery for OWL Querying and Reasoning (Work in Progress)*

Jesús M. Almendros-Jiménez
Dept. of Informatics
Universidad de Almería
04120 Almería (Spain)
jalmen@ual.es

One of the main aims of the so-called Web of Data is to be able to handle heterogeneous resources where data can be expressed in either XML or RDF. The design of programming languages able to handle both XML and RDF data is a key target in this context. In this paper we present a framework called XQOWL that makes possible to handle XML and RDF/OWL data with XQuery. XQOWL can be considered as an extension of the XQuery language that connects XQuery with SPARQL and OWL reasoners. XQOWL embeds SPARQL queries (via Jena SPARQL engine) into XQuery and enables to make calls to OWL reasoners (HermiT, Pellet and FaCT++) from XQuery. It permits to combine queries against XML and RDF/OWL resources as well as to reason with RDF/OWL data. Therefore input data can be either XML or RDF/OWL and output data can be formatted in XML (also using RDF/OWL XML serialization).

1 Introduction

There are two main formats to publish data on the Web. The first format is *XML*, which is based on a tree-based model and for which the *XPath* and *XQuery* languages for querying, and the *XSLT* language for transformation, have been proposed. The second format is *RDF* which is a graph-based model and for which the *SPARQL* language for querying and transformation has been proposed. Both formats (XML and RDF) can be used for describing data of a certain domain of interest. XML is used for instance in the *Dublin Core* [1], *MPEG-7* [2], among others, while RDF is used in *DBPedia* [3] and *LinkedLifeData* [4], among others. The number of organizations that offers their data from the Web is increasing in the last years. The so-called *Linked open data* initiative [5] aims to interconnect the published Web data.

XML and RDF share the same end but they have different data models and query/transformation languages. Some data can be available in XML format and not in RDF format and vice versa. The *W3C (World Wide Web Consortium)* [6] proposes transformations from XML data to RDF data (called *lifting*), and vice versa (called *lowering*). RDF has XML-based representations (called *serializations*) that makes possible to represent in XML the graph based structure of RDF. However, XML-based languages are not usually used to query/transform serializations of RDF. Rather than SPARQL is used to query RDF whose

*This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-44742-C4-4-R, as well as by the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

[1] http://www.dublincore.org/.
[2] http://mpeg.chiariglione.org/.
[3] http://www.dbpedia.org/.
[4] http://linkedlifedata.com/.
[5] http://linkeddata.org/
[6] http://www.w3.org/.

Submitted to:
PROLE 2014

syntax resembles SQL and abstract from the XML representation of RDF. The same happens when data are available in XML format: queries and transformations are usually expressed in XPath/XQuery/XSLT, instead of transforming XML to RDF, and using SPARQL.

One of the main aims of the so-called Web of Data is to be able to handle heterogeneous resources where data can be expressed in either XML or RDF. The design of programming languages able to handle both XML and RDF data is a key target in this context and some recent proposals have been presented with this end. One of most known is XSPARQL [5] which is an hybrid language which combines XQuery and SPARQL allowing to query XML and RDF. XSPARQL extends the XQuery syntax with new expressions able to traverse a RDF graph and construct the graph of the result of a query on RDF. One of the uses of XSPARQL is the definition of lifting and lowering from XML to RDF and vice versa. But also XSPARQL is able to query XML and RDF data without transforming them, and obtaining the result in any of the formats. They have defined a formal semantics for XSPARQL which is an extension of the XQuery semantics. The SPARQL2XQuery interoperability framework [4] aims to overcome the same problem by considering as query language SPARQL for both formats (XML and RDF), where SPARQL queries are transformed into XQuery queries by matching XML Schemas into RDF metadata. In early approaches, SPARQL queries are embedded into XQuery and XSLT [7] and XPath expressions are embedded into SPARQL queries [6].

OWL is an ontology language working with concepts (i.e., classes) and roles (i.e., object/data properties) as well as with individuals (i.e., instances) which fill concepts and roles. OWL can be considered as an extension of RDF in which a richer vocabulary allows to express new relationships. OWL offers more complex relationships than RDF between entities including means to limit the properties of classes with respect to the number and type, means to infer that items with various properties are members of a particular class and a well-defined model of property inheritance. OWL reasoning [16] is a topic of research of increasing interest in the literature. Most of OWL reasoners (for instance, *HermiT* [12], *Racer* [9], FaCT++ [17], Pellet [15]) are based on tableaux based decision procedures.

In this context, we can distinguish between (1) *reasoning tasks* and (2) *querying tasks* from a given ontology. The most typical (1) *reasoning tasks*, with regard to a given ontology, include: (a) *instance checking*, that is, whether a particular individual is a member of a given concept, (b) *relation checking*, that is, whether two individuals hold a given role, (c) *subsumption*, that is, whether a concept is a subset of another concept, (d) *concept consistency*, that is, consistency of the concept relationships, and (e) a more general case of consistency checking is *ontology consistency* in which the problem is to decide whether a given ontology has a model. However, one can be also interested in (2) *querying tasks* such as: (a) *instance retrieval*, which means to retrieve all the individuals of a given concept, and (b) *property fillers retrieval* which means to retrieve all the individuals which are related to a given individual with respect to a given role.

SPARQL provides mechanisms for querying tasks while OWL reasoners are suitable for reasoning tasks. SPARQL is a query language for RDF/OWL triples whose syntax resembles SQL. OWL reasoners implement a complex deduction procedure including ontology consistency checking that SPARQL is not able to carry out. Therefore SPARQL/OWL reasoners are complementary in the world of OWL.

In this paper we present a framework called XQOWL that makes possible to handle XML and RDF/OWL data with XQuery. XQOWL can be considered as an extension of the XQuery language that connects XQuery with SPARQL and OWL reasoners. XQOWL embeds SPARQL queries (via Jena SPARQL engine) into XQuery and enables to make calls to OWL reasoners (HermiT, Pellet and FaCT++) from XQuery. It permits to combine queries against XML and RDF/OWL resources as well as to reason with RDF/OWL data. Therefore input data can be either XML or RDF/OWL and output data can be formatted in XML (also using RDF/OWL XML serialization).

Thus the framework proposes to embed SPARQL code into XQuery as well as to make calls to OWL reasoners from XQuery. With this aim a *Java API* has been implemented on top of the OWL API [11] and OWL Reasoner API [10] that makes possible to interconnect XQuery with SPARQL and OWL reasoners. The Java API is invoked from XQuery thanks to the use of the *Java Binding* facility available in most of XQuery processors (this is the case, for instance, of *BaseX* [8], *Exist* [14] and *Saxon* [13]). The Java API enables to connect XQuery to HermiT, Pellet and FaCT++ reasoners as well as to Jena SPARQL engine. The Java API returns the results of querying and reasoning in XML format which can be handled from XQuery. It means that querying and reasoning RDF/OWL with XQOWL one can give XML format to results in either XML or RDF/OWL. In particular, lifting and lowering is possible in XQOWL.

Therefore our proposal can be seen as an extension of the proposed approaches for combining SPARQL and XQuery. Our XQOWL framework is mainly focused on the use of XQuery for querying and reasoning with OWL ontologies. It makes possible to write complex queries that combines SPARQL queries with reasoning tasks. As far as we know our proposal is the first to provide such a combination.

The implementation has been tested with the BaseX processor [8] and can be downloaded from our Web site `http://indalog.ual.es/XQOWL`. There the XQOWL API and the examples of the paper are available as well as installation instructions.

Let us remark that here we continue our previous works on combination of XQuery and the Semantic Web. In [1] we have described how to extend the syntax of XQuery in order to query RDF triples. After, in [2] we have presented a (Semantic Web) library for XQuery which makes possible to retrieve the elements of an ontology as well as to use SWRL. Here, we have followed a new direction, by embedding existent query languages (SPARQL) and reasoners into XQuery.

The structure of the paper is as follows. Section 2 will show an example of OWL ontology used in the rest of the paper as running example. Section 3 will describe XQOWL: the Java API as well as examples of use. Finally, Section 4 will conclude and present future work.

2 OWL

In this section we show an example of ontology which will be used in the rest of the paper as running example. Let us suppose an ontology about a social network (see Table 1) in which we define ontology classes: user, user_item, activity; and event, message $\sqsubseteq$ activity (1); and wall, album $\sqsubseteq$ user_item (2). In addition, we can define (object) properties as follows: created_by which is a property whose domain is the class activity and the range is user (3), and has two sub-properties: added_by, sent_by (4) (used for events and messages, respectively).

We have also belongs_to which is a functional property (5) whose domain is user_item and range is user (6); friend_of which is a irreflexive (7) and symmetric (8) property whose domain and range is user (9); invited_to which is a property whose domain is user and range is event (10); recommended_friend_of which is a property whose domain and range is user (11), and is the composition of friend_of and friend_of (12); replies_to which is an irreflexive property (13) whose domain and range is message (14); written_in which is a functional property (15) whose domain is message and range is wall (16); attends_to which is a property whose domain is user and range is event (17) and is the inverse of the property confirmed_by (18); i_like_it which is a property whose domain is user and range is activity (19), which is the inverse of the property liked_by (20).

Besides, there are some (data) properties: the content of a message (21), the date (22) and name (23)

Ontology	
(1) event, message $\sqsubseteq$ activity	(2) wall, album $\sqsubseteq$ user_item
(3) $\forall$ created_by.activity $\sqsubseteq$ user	(4) added_by, sent_by $\sqsubseteq$ created_by
(5) $\top \sqsubseteq \leq 1$. belongs_to	(6) $\forall$ belongs_to.user_item $\sqsubseteq$ user
(7) $\exists$ friend_of.Self $\sqsubseteq \bot$	(8) friend_of^{-} $\sqsubseteq$ friend_of
(9) $\forall$ friend_of.user $\sqsubseteq$ user	(10) $\forall$ invited_to.user $\sqsubseteq$ event
(11) $\forall$ recommended_friend_of.user $\sqsubseteq$ user	(12) friend_of $\cdot$ friend_of $\sqsubseteq$ recommended_friend_of
(13) $\exists$ replies_to.Self $\sqsubseteq \bot$	(14) $\forall$ replies_to.message $\sqsubseteq$ message
(15) $\top \sqsubseteq \leq 1$.written_in	(16) $\forall$ written_in.message $\sqsubseteq$ wall
(17) $\forall$ attends_to.user $\sqsubseteq$ event	(18) attends_to^{-} $\equiv$ confirmed_by
(19) $\forall$ i_like_it.user $\sqsubseteq$ activity	(20) i_like_it^{-} $\equiv$ liked_by
(21) $\forall$ content.message $\sqsubseteq$ String	(22) $\forall$ date.event $\sqsubseteq$ DateTime
(23) $\forall$ name.event $\sqsubseteq$ String	(24) $\forall$ nick.user $\sqsubseteq$ String
(25) $\forall$ password.user $\sqsubseteq$ String	(26) event$\sqcap\exists$confirmed_by.user $\sqsubseteq$ popular
(27) activity$\sqcap$ $\exists$liked_by.user $\sqsubseteq$ popular	(28) activity $\sqsubseteq \leq 1$ created_by.user
(29) message $\sqcap$ event $\equiv \bot$	

Table 1: Social Network Ontology (in Description Logic Syntax)

Ontology Instance
user(jesus), nick(jesus,jalmen), password(jesus,passjesus), friend_of(jesus,luis)
user(luis), nick(luis,Iamluis), password(luis,luis0000)
user(vicente), nick(vicente,vicente), password(vicente,vicvicvic), friend_of(vicente,luis), i_like_it(vicente,message2), invited_to(vicente,event1), attends_to(vicente,event1)
event(event1), added_by(event1,luis), name(event1,"Next conference"), date(event1,21/10/2014)
event(event2)
message(message1), sent_by(message1,jesus), content(message1,"I have sent the paper")
message(message2), sent_by(message2,luis), content(message2,"good luck!"), replies_to(message2,message1)
wall(wall_jesus), belongs_to(wall_jesus,jesus)
wall(wall_luis), belongs_to(wall_luis,luis)
wall(wall_vicente), belongs_to(wall_vicente,vicente)

Table 2: Individuals and object/data properties of the ontology

of an event, and the nick (24) and password (25) of a user. Finally, we have defined the concepts popular which are events confirmed_by some user and activities liked_by some user ((26) and (27)) and we have defined constraints: activities are created_by at most one user (28) and message and event are disjoint classes (29). Let us now suppose the set of individuals and object/data property instances of Table 2.

From OWL reasoning we can deduce new information. For instance, the individual *message1* is an

Java API
public OWLReasoner getOWLReasonerHermiT(OWLOntology ontology)
public OWLReasoner getOWLReasonerPellet(OWLOntology ontology)
public OWLReasoner getOWLReasonerFact(OWLOntology ontology)
public String OWLSPARQL(String filei,String queryStr)
public <T extends OWLAxiom> String OWLQuerySetAxiom(Set<T> axioms)
public <T extends OWLEntity> String[] OWLQuerySetEntity(Set<T> elems)
public <T extends OWLEntity> String[] OWLReasonerNodeEntity(Node <T> elem)
public <T extends OWLEntity> String[] OWLReasonerNodeSetEntity(NodeSet<T> elems)

Table 3: Java API of XQOWL

activity, because message is a subclass of activity, and the individual *event1* is also an activity because event is a subclass of activity. The individual *wall_jesus* is an user_item because wall is a subclass of user_item. These inferences are obtained from the subclass relation. In addition, object properties give us more information. For instance, the individuals *message1*, *message2* and *event1* have been created_by *jesus*, *luis* and *luis*, respectively, since the properties sent_by and added_by are sub-properties of created_by. In addition, the individual *luis* is a friend_of *jesus* because friend_of is symmetric.

More interesting is that the individual *vicente* is a recommended_friend_of *jesus*, because *jesus* is a friend_of *luis*, and *luis* is a friend_of *vicente*, which is deduced from the definition of recommended_friend_of, which is the composition of friend_of and friend_of. Besides, the individual *event1* is confirmed_by *vicente*, because *vicente* attends_to *event1* and the properties confirmed_by and attends_to are inverses. Finally, there are popular concepts: *event1* and *message2*; the first one has been confirmed_by *vicente* and the second one is liked_by *vicente*.

The previous ontology is consistent. The ontology might introduce elements that make the ontology inconsistent. We might add a user being friend_of of him(er) self. Even more, we can define that certain events and messages are created_by (either added_by or sent_by) more than one user. Also a message can reply to itself. However, there are elements that do not affect ontology consistency. For instance, *event2* has not been created_by users. The ontology only requires to have at most one creator. Also, messages have not been written_in a wall.

3 XQOWL

XQOWL allows to embed SPARQL queries into XQuery. It also makes possible to make calls to OWL reasoners. With this aim a Java API has been developed.

3.1 The Java API

Now, we show the main elements of the Java API developed for connecting XQuery and SPARQL and OWL reasoners. Basically, the Java API has been developed on top of the OWL API and the OWL Reasoner API and makes possible to retrieve results from SPARQL and OWL reasoners. The elements of the library are shown in Table 3.

The first three elements of the library: *getOWLReasonerHermiT*, *getOWLReasonerPellet* and *getOWLReasonerFact* make possible to instantiate HermiT, Pellet and FaCT++ reasoners. For instance, the code of *getOWLReasonerHermiT* is as follows:

```
public OWLReasoner getOWLReasonerHermiT(OWLOntology ontology){
    org.semanticweb.HermiT.Reasoner reasoner = new Reasoner(ontology);
    reasoner.precomputeInferences(InferenceType.CLASS_HIERARCHY,
                  InferenceType.CLASS_ASSERTIONS,
                  InferenceType.DATA_PROPERTY_ASSERTIONS,
                  InferenceType.DATA_PROPERTY_HIERARCHY,
                  InferenceType.DISJOINT_CLASSES,
                  InferenceType.DIFFERENT_INDIVIDUALS,
                  InferenceType.OBJECT_PROPERTY_ASSERTIONS,
                  InferenceType.OBJECT_PROPERTY_HIERARCHY,
                  InferenceType.SAME_INDIVIDUAL);
        return reasoner;
  };
```

The fourth element of the library *OWLSPARQL* makes possible to instantiate SPARQL Jena engine. The input of this method is an ontology included in a file and a string representing the SPARQL query. The output is a file (name) including the result of the query. The code of *OWLSPARQL* is as follows:

```
public String OWLSPARQL(String filei,String queryStr)
throws FileNotFoundException{
  OntModel model = ModelFactory.createOntologyModel();
  model.read(filei);
  com.hp.hpl.jena.query.Query query = QueryFactory.create(queryStr);
  ResultSet result =
      (ResultSet) SparqlDLExecutionFactory.create(query,model).execSelect();
  String fileName = "./tmp/"+result.hashCode()+"result.owl";
  File f = new File(fileName);
  FileOutputStream file = new FileOutputStream(f);
  ResultSetFormatter.outputAsXML(file,(com.hp.hpl.jena.query.ResultSet) result);
  try { file.close(); } catch (IOException e) {e.printStackTrace();}
     return fileName;
  };
```

We can see in the code that the result of the query is obtained in XML format and stored in a file. The rest of elements (i.e, *OWLQuerySetAxiom*, *OWLQuerySetEntity*, *OWLReasonerNodeSetEntity* and *OWLReasonerNodeEntity*) of the Java API make possible to handle the results of calls to SPARQL and OWL reasoners.

OWL Reasoners implement Java interfaces of the OWL API for storing OWL elements. The main Java interfaces are *OWLAxiom* and *OWLEntity*. *OWLAxiom* is a Java interface which is a super-interface of all the types of OWL axioms: *OWLSubClassOfAxiom*, *OWLSubDataPropertyOfAxiom*, *OWLSubObjectPropertyOfAxiom*, etc. *OWLEntity* is a Java interface which is a super-interface of all types of OWL elements: *OWLClass*, *OWLDataProperty*, *OWLDatatype*, etc.

The XQOWL API includes the method *OWLQuerySetAxiom* that returns a file name where a set of axioms are included. It also includes *OWLQuerySetEntity* that returns in an array the URI's of a set of entities. Moreover, *OWLReasonerNodeEntity* returns in an array the URI's of a node. Finally, *OWLReasonerNodeSetEntity* returns in an array the URIs of a set of nodes. For instance, the code of *OWLQuerySetEntity* is as follows:

```
public <T extends OWLEntity> String[] OWLQuerySetEntity(Set<T> elems)
    {
      String[] result = new String[elems.size()];
      Iterator<T> it = elems.iterator();
      for(int i=0;i<elems.size();i++){
        result[i]=it.next().toStringID();
              };
    return result;
    };
```

3.2 XQOWL: SPARQL

XQOWL is an extension of the XQuery language. Firstly, XQOWL allows to write XQuery queries in which calls to SPARQL queries are achieved and the results of SPARQL queries in XML format (see [3]) can be handled by XQuery. In XQOWL, XQuery variables can be bounded to results of SPARQL queries and vice versa, XQuery bounded variables can be used in SPARQL expressions. Therefore, in XQOWL both XQuery and SPARQL queries can share variables.

Example 3.1 *For instance, the following query returns the individuals of concepts* user *and* event *in the social network:*

```
declare namespace spql="http://www.w3.org/2005/sparql-results#";
declare namespace xqo = "java:xqowl.XQOWL";

let $model := "socialnetwork.owl"
for $class in ("sn:user","sn:event")
return
let $queryStr := concat(
  "PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
   PREFIX sn: <http://www.semanticweb.org/socialnetwork.owl#>
   SELECT ?Ind
   WHERE { ?Ind rdf:type ", $class," }")
return
let $xqo := xqo:new()
let $res:= xqo:OWLSPARQL($xqo,$model,$queryStr)
return
doc($res)/spql:sparql/spql:results/spql:result/spql:binding/spql:uri/text()
```

Let us observe that the name of the classes (i.e., sn:user *and* sn:event*) is defined by an XQuery variable (i.e.,* `$class`*) in a* `for` *expression, which is passed as parameter of the SPARQL expression. In addition, the result is obtained in an XQuery variable (i.e.* `$res`*). Here OWLSPARQL of the XQOWL API is used to call the SPARQL Jena engine, which returns a file name (a temporal file) in which the result is found. Now,* `$res` *can be used from XQuery to obtain the URIs of the elements:*

$$\texttt{doc}(\$\texttt{res})/\texttt{spql:sparql}/\texttt{spql:results}/\texttt{spql:result}/\texttt{spql:binding}/\texttt{spql:uri}/\texttt{text}()$$

In this case, we obtain the following plain text:

```
http://www.semanticweb.org/socialnetwork.owl#vicente
http://www.semanticweb.org/socialnetwork.owl#jesus
http://www.semanticweb.org/socialnetwork.owl#luis
http://www.semanticweb.org/socialnetwork.owl#event2
http://www.semanticweb.org/socialnetwork.owl#event1
```

Example 3.2 *Another example of using XQOWL and SPARQL is the code of lowering from the document:*

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://relations.org">
  <foaf:Person xmlns:foaf="http://xmlns.com/foaf/0.1/" rdf:about="#b1">
    <foaf:name>Alice</foaf:name>
    <foaf:knows>
      <foaf:Person rdf:about="#b4"/>
    </foaf:knows>
    <foaf:knows>
      <foaf:Person rdf:about="#b6"/>
    </foaf:knows>
  </foaf:Person>
  <foaf:Person xmlns:foaf="http://xmlns.com/foaf/0.1/" rdf:about="#b4">
```

```
    <foaf:name>Bob</foaf:name>
    <foaf:knows>
      <foaf:Person rdf:about="#b6"/>
    </foaf:knows>
  </foaf:Person>
  <foaf:Person xmlns:foaf="http://xmlns.com/foaf/0.1/" rdf:about="#b6">
    <foaf:name>Charles</foaf:name>
  </foaf:Person>
</rdf:RDF>
```

to the document:

```
<relations>
<person name="Alice">
<knows> Bob </knows>
<knows> Charles </knows>
</person>
<person name="Bob">
<knows> Charles </knows>
</person>
<person name="Charles" />
</relations>
```

This example has been taken from [5][7] in which they show the lowering example in XSPARQL. In our case the code of the lowering example is as follows:

```
declare namespace spql="http://www.w3.org/2005/sparql-results#";
declare namespace xqo = "java:xqowl.XQOWL";
declare variable $model := "relations.rdf";

let $query1 :=
    "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
    PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    SELECT ?Person ?Name
    WHERE {
    ?Person foaf:name ?Name
    } ORDER BY ?Name"
let $xqo := xqo:new(),
$result := xqo:OWLSPARQL($xqo,$model,$query1)
return
for $Binding in doc($result)/spql:sparql/spql:results/spql:result
let $Name := $Binding/spql:binding[@name="Name"]/spql:literal/text(),
    $Person := $Binding/spql:binding[@name="Person"]/spql:uri/text(),
    $PersonName := functx:fragment-from-uri($Person)
return
<person name="{$Name}">{
let  $query2 :=
  concat(
  "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
  PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  PREFIX rel:  <http://relations.org#>
  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  SELECT ?FName
  WHERE {
              _:",$PersonName," foaf:knows ?Friend .
              _:",$PersonName," foaf:name ", "'",$Name,"' .
              ?Friend foaf:name ?FName
}")
let $result2 := xqo:OWLSPARQL($xqo,$model,$query2)
```

[7] XSPARQL works with blank nodes, and there the RDF document includes `nodeID` tag for each RDF item. In XQOWL we cannot deal with blank nodes at all, and therefore a preprocessing of the RDF document is required: `nodeID` tags are replaced by `about`.

```
return
for $FName in doc($result2)/spql:sparql/spql:results/spql:result/spql:binding/
    spql:literal/text()
return
<knows>{$FName}</knows>
}
</person>
}
</relations>
```

In this example, two SPARQL queries are nested and share variables. The result of the first SPARQL query (i.e., `$PersonName` *and* `$Name`*) is used in the second SPARQL query.*

3.3 XQOWL: OWL Reasoners

XQOWL can be also used for querying and reasoning with OWL. With this aim the OWL API and OWL Reasoner API have been integrated in XQuery. Also for this integration, the XQOWL API is required. For using OWL Reasoners from XQOWL there are some calls to be made from XQuery code. Firstly, we have to instantiate the ontology manager by using *createOWLOntologyManager*; secondly, the ontology has to be loaded by using *loadOntologyFromOntologyDocument*; thirdly, in order to handle OWL elements we have to instantiate the data factory by using *getOWLDataFactory*; finally, in order to select a reasoner *getOWLReasonerHermiT*, *getOWLReasonerPellet* and *getOWLReasonerFact* are used.

Example 3.3 *For instance, we can query the object properties of the ontology using the OWL API as follows:*

```
let $xqo := xqo:new(),
    $man := api:createOWLOntologyManager(),
    $fileName := file:new($file),
    $ont := om:loadOntologyFromOntologyDocument($man,$fileName)
return
doc(xqo:OWLQuerySetAxiom($xqo,o:getAxioms($ont)))/rdf:RDF/owl:ObjectProperty
```

obtaining the following result:

```
<ObjectProperty...rdf:about="...#added_by">
  <rdfs:subPropertyOf rdf:resource="...#created_by"/>
  <rdfs:domain rdf:resource="...#event"/>
  <rdfs:range rdf:resource="...#user"/>
</ObjectProperty>
<ObjectProperty ... rdf:about="...#attends_to">
  <inverseOf rdf:resource="...#confirmed_by"/>
  <rdfs:range rdf:resource="...#event"/>
  <rdfs:domain rdf:resource="...#user"/>
</ObjectProperty>
...
```

Example 3.4 *Another example of query using the OWL API is the following which requests class axioms related to* wall *and* event*:*

```
let $xqo := xqo:new(),
  $man := api:createOWLOntologyManager(),
  $fileName := file:new($file),
  $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
  $fact := om:getOWLDataFactory($man)
return
for $class in ("wall","event")
let $iri := iri:create(concat($base,$class)),
  $class := df:getOWLClass($fact,$iri)
```

```
return
doc(xqo:OWLQuerySetAxiom($xqo,o:getAxioms($ont,$class)))/rdf:RDF/owl:Class
```

in which a `for` *expression is used to define the names of the classes to be retrieved, obtaining the following result:*

```
<Class ... rdf:about="...#user_item"/>
<Class ... rdf:about="...#wall">
  <rdfs:subClassOf rdf:resource="...#user_item"/>
</Class>
<Class ... rdf:about="...#activity"/>
<Class ... rdf:about="...#event">
  <rdfs:subClassOf rdf:resource="...#activity"/>
  <disjointWith rdf:resource="...#message"/>
</Class>
<Class ... rdf:about="...#message"/>
```

Now we can see examples about how to use XQOWL for reasoning with an ontology. With this aim, we can use the OWL Reasoner API (as well as the XQOWL API). The XQOWL API allows easily to use HermiT, Pellet and FaCT++ reasoners.

Example 3.5 *For instance, let us suppose we want to check the consistence of the ontology by the HermiT reasoner. The code is as follows:*

```
let $xqo := xqo:new(),
  $man := api:createOWLOntologyManager(),
  $fileName := file:new($file),
  $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
  $fact := om:getOWLDataFactory($man),
  $reasoner := xqo:getOWLReasonerHermiT($xqo,$ont),
  $boolean := r:isConsistent($reasoner),
  $dispose := r:dispose($reasoner)
return $boolean
```

which returns `true`*. Here the HermiT reasoner is instantiated by using getOWLReasonerHermiT. In addition, the OWL Reasoner API method isConsistent is used to check ontology consistence. Each time the work of the reasoner is done, a call to dispose is required.*

Example 3.6 *Let us suppose now we want to retrieve instances of concepts* `activity` *and* `user`*. Now, we can write the following query using the HermiT reasoner:*

```
for $classes in ("activity","user")
let $xqo := xqo:new(),
  $man := api:createOWLOntologyManager(),
  $fileName := file:new($file),
  $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
  $fact := om:getOWLDataFactory($man),
  $iri := iri:create(concat($base,$classes)),
  $reasoner := xqo:getOWLReasonerHermiT($xqo,$ont),
  $class := df:getOWLClass($fact,$iri),
  $result:= r:getInstances($reasoner,$class,false()),
  $dispose := r:dispose($reasoner)
return
<concept class="{$classes}">
{ for $instances in xqo:OWLReasonerNodeSetEntity($xqo,$result)
    return <instance>{substring-after($instances,'#')}</instance>}
</concept>
```

obtaining the following result in XML format:

```
<concept class="activity">
  <instance>message1</instance>
  <instance>message2</instance>
  <instance>event1</instance>
  <instance>event2</instance>
</concept>
<concept class="user">
  <instance>jesus</instance>
  <instance>vicente</instance>
  <instance>luis</instance>
</concept>
```

Here getInstances of the OWL Reasoner API is used to retrieve the instances of a given ontology class. In addition, a call to create of the OWL API, which creates the IRI of the class, and a call to getClass of the OWL API, which retrieves the class, are required. The OWL Reasoner is able to deduce that `message1` *and* `message2` *belongs to concept* `activity` *since they belongs to concept* `message` *and* `message` *is a subconcept of* `activity`*. The same can be said for events.*

Example 3.7 *Let us suppose now we want to retrieve the subconcepts of* `activity` *using the Pellet reasoner. The code is as follows:*

```
let $xqo := xqo:new(),
  $man := api:createOWLOntologyManager(),
  $fileName := file:new($file),
  $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
  $fact := om:getOWLDataFactory($man),
  $iri := iri:create(concat($base,"activity")),
  $reasoner := xqo:getOWLReasonerPellet($xqo,$ont),
  $class := df:getOWLClass($fact,$iri),
  $result:= r:getSubClasses($reasoner,$class,false()),
  $dispose := r:dispose($reasoner)
return
  for $subclass in xqo:OWLReasonerNodeSetEntity($xqo,$result)
    return <subclass>{substring-after($subclass,'#')} </subclass>
```

and the result in XML format is as follows:

```
<subclass>popular_message</subclass>
<subclass>event</subclass>
<subclass>Nothing</subclass>
<subclass>popular_event</subclass>
<subclass>message</subclass>
```

Here getSubClasses of the OWL Reasoner API is used.

Example 3.8 *Finally, let us suppose we want to retrieve the recommended friends of* `jesus`*. Now, the query is as follows:*

```
let $xqo := xqo:new(),
  $man := api:createOWLOntologyManager(),
  $fileName := file:new($file),
  $ont := om:loadOntologyFromOntologyDocument($man,$fileName),
  $fact := om:getOWLDataFactory($man),
  $iri := iri:create(concat($base,"recommended_friend_of")),
  $iri2 := iri:create(concat($base,"jesus")),
  $reasoner := xqo:getOWLReasonerPellet($xqo,$ont),
  $property := df:getOWLObjectProperty($fact,$iri),
  $ind := df:getOWLNamedIndividual($fact,$iri2),
  $result:= r:getObjectPropertyValues($reasoner,$ind,$property),
  $dispose := r:dispose($reasoner)
return
```

```
for $rfriend in xqo:OWLReasonerNodeSetEntity($xqo,$result)
return
<recommended_friend>
{substring-after($rfriend,'#')}
</recommended_friend>
```

and the answer as follows:

```
<recommended_friend>jesus</recommended_friend>
<recommended_friend>vicente</recommended_friend>
```

Here the OWL Reasoner API is used to deduce the friends of friends of `jesus`*. Due to symmetry of* `friend` *relationship, a person is a recommended friend of itself.*

4 Conclusions and Future Work

In this paper we have presented an extension of XQuery called XQOWL to query XML and RDF/OWL documents, as well as to reason with RDF/OWL resources. We have described the XQOWL API that allows to make calls from XQuery to SPARQL and OWL Reasoners. Also we have shown examples of use of XQOWL. The main advantage of the approach is to be able to handle both types of documents through the sharing of variables between XQuery and SPARQL/OWL Reasoners. The implementation has been tested with the BaseX processor [8] and can be downloaded from our Web site `http://indalog.ual.es/XQOWL`. As future work, we would like to extend our work as follows. Firstly, we would like to extend our Java API. More concretely, with the SWRL API in order to execute rules from XQuery, and to be able to provide explanations about ontology inconsistence. Secondly, we would like to use our framework in ontology transformations (refactoring, coercion, splitting, amalgamation) and matching.

References

[1] Jesús Manuel Almendros-Jiménez (2009): *Extending XQuery for Semantic Web Reasoning*. In: *Applications of Declarative Programming and Knowledge Management - 18th International Conference, INAP 2009*, *LNCS* 6547, Springer, pp. 117–134.

[2] Jesús Manuel Almendros-Jiménez (2011): *Querying and Reasoning with RDF(S)/OWL in XQuery*. In: *AP-Web 2011 Proceedings*, *LNCS* 6612, Springer, pp. 450–459.

[3] Dave Beckett & Jeen Broekstra (2013): *SPARQL Query Results XML Format (Second Edition)*. `http://http://www.w3.org/TR/rdf-sparql-XMLres/`.

[4] Nikos Bikakis, Chrisa Tsinaraki, Ioannis Stavrakantonakis, Nektarios Gioldasis & Stavros Christodoulakis (2013): *The SPARQL2XQuery interoperability framework*. *World Wide Web*, pp. 1–88.

[5] Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes & Axel Polleres (2012): *Mapping between RDF and XML with XSPARQL*. *Journal on Data Semantics* 1(3), pp. 147–185.

[6] Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schöpf, Hannes Staffler et al. (2009): *Bringing the XML and semantic web worlds closer: transforming XML into RDF and embedding XPath into SPARQL*. In: *Enterprise Information Systems*, Springer, pp. 31–45.

[7] Sven Groppe, Jinghua Groppe, Volker Linnemann, Dirk Kukulenz, Nils Hoeller & Christoph Reinke (2008): *Embedding SPARQL into XQUERY/XSLT*. In: *Proceedings of the 2008 ACM symposium on Applied computing*, ACM, pp. 2271–2278.

[8] Christian Grun (2014): *BaseX. The XML Database*. `http://basex.org`.

[9] V. Haarslev, R. Möller & S. Wandelt (2008): *The revival of structural subsumption in tableau-based description logic reasoners*. In: *Proceedings of the 2008 International Workshop on Description Logics (DL2008), CEUR-WS*, pp. 701–706.

[10] Matthew Horridge (2009): *OWL Reasoner API*. `http://owlapi.sourceforge.net/javadoc/org/semanticweb/owlapi/reasoner/OWLReasoner.html`.

[11] Matthew Horridge & Sean Bechhofer (2011): *The OWL API: A Java API for OWL ontologies*. *Semantic Web* 2(1), pp. 11–21.

[12] Ian Horrocks, Boris Motik & Zhe Wang (2012): *The HermiT OWL Reasoner*. In: *Proceedings of the 1st International Workshop on OWL Reasoner Evaluation (ORE-2012), Manchester, UK*.

[13] Michael Kay (2008): *Ten Reasons Why Saxon XQuery is Fast*. *IEEE Data Eng. Bull.* 31(4), pp. 65–74.

[14] Wolfgang Meier (2003): *eXist: An open source native XML database*. In: *Web, Web-Services, and Database Systems*, Springer, pp. 169–183.

[15] Evren Sirin, Bijan Parsia, Bernardo C. Grau, Aditya Kalyanpur & Yarden Katz (2007): *Pellet: A practical OWL-DL reasoner*. *Web Semantics: Science, Services and Agents on the World Wide Web* 5(2), pp. 51–53, doi:10.1016/j.websem.2007.03.004. Available at `http://dx.doi.org/10.1016/j.websem.2007.03.004`.

[16] Steffen Staab & Rudi Studer (2010): *Handbook on ontologies*. Springer.

[17] D. Tsarkov & I. Horrocks (2006): *FaCT++ Description Logic Reasoner: System Description*. In: *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, Springer, LNAI 4130, pp. 292–297.

Logical Foundations for More Expressive Declarative Temporal Logic Programming Languages (Extended Abstract)

Jose Gaintzarain
University of the Basque Country
Bilbao, Spain
EUITI de Bilbao
jose.gaintzarain@ehu.es

Paqui Lucio
University of the Basque Country
San Sebastián, Spain
Facultad de Informática
paqui.lucio@ehu.es

We provide an extended abstract of the article with the same title and authors published in the journal *ACM Transactions on Computational Logic, Vol. 14, No. 4, Article 28, November 2013*, DOI: http://dx.doi.org/10.1145/2528931.

Temporal logic is extensively used in the specification, refinement, development, and verification of software and hardware systems. Indeed, temporal logic constitutes the foundation of many formal methods and techniques whose central purpose is to improve the reliability of computer systems, in particular to verify their correctness. The term temporal logic is often used to denote the broad class of logical systems that are aimed to the representation of temporal information. Propositional Linear-time Temporal Logic (PLTL) is one of such logical systems. In PLTL, the intended model for time is the standard model of natural numbers and, consequently, this logic is appropriate for reasoning about discrete, linear sequences of states. Different contributions in the literature on temporal logic show its usefulness in computer science and other related areas. For a recent and extensive monograph on PLTL techniques and tools, we refer the reader to [6], where sample applications along with references to specific works that use this temporal formalism to represent dynamic entities in a wide variety of fields –such as Program Specification, System Verification, Robotics, Reactive Systems, Databases, Control Systems, Agent-based Systems, etc– can be found. The syntax of PLTL extends the syntax of classical propositional logic by allowing the use of temporal connectives. Different temporal connectives can be considered in order to obtain the full expressiveness of PLTL. One option is to consider the temporal connectives $\circ$ ("next") and $\mathscr{U}$ ("until") as primitive temporal connectives. Formulas of the form $\circ\psi$ and $\varphi \mathscr{U} \psi$ are interpreted, respectivelly, as "the next state makes ψ true" and "φ is true from now until ψ eventually becomes true". Additional useful temporal connectives can be defined as derived connectives, e.g. $\Diamond$ ("eventually"), $\Box$ ("always") and $\mathscr{R}$ ("release").

Temporal Logic Programming (TLP) deals with the direct execution of temporal logic formulas. Hence TLP provides a single framework in which dynamic systems can be specified, developed, validated, and verified by means of executable specifications that make possible to prototype, debug, and improve systems before their final use. In TLP, the direct execution of a formula corresponds to building a model for that formula.

It is well known that one of the features of temporal logic is the ability to express eventualities and invariants. An eventuality is a formula that asserts that something does eventually hold. For example, to fulfill the formula $\varphi \mathscr{U} \psi$, the formula ψ must eventually be satisfied. Invariants state that a property will always be true (from some moment onwards). Syntactically, eventualities are easily detectable but invariants can be expressed in intricate ways by means of loops. Consequently, we say that invariants can

Submitted to:
PROLE 2014

be "hidden". Since a "hidden" invariant can prevent the fulfillment of an eventuality, the way in which the issue of eventualities and "hidden" invariants is dealt with becomes an outstanding characteristic of TLP languages. The use of the customary inductive definitions of the temporal connectives as the only mechanism for detecting the existence of an invariant that prevents the fulfillment of an eventuality, leads to incompleteness. The reason is that such inductive definitions make possible to indefinitely postpone the fulfillment of an eventuality and, consequently, they make possible to indefinitely postpone the contradiction between an eventuality that states that a property ψ will eventually hold and an invariant that states that ψ will never hold. Therefore, more elaborated mechanisms are needed.

TLP, in a broad sense, means programming in any language based on temporal logic. In classical Logic Programming, the underlying execution procedure is based on (classical) clausal resolution ([14, 15]). The extension of this approach to Temporal Logic Programming faces three main challenges: the undecidability of first-order temporal logic [16, 24, 23], the difficulty for dealing with eventualities and invariants and the complexity (even for the propositional fragment [22]). Consequently, different proposals that can be classified into two groups have arisen. One of the groups is formed by the languages that are based on the *imperative future* approach (e.g. [18, 2, 17]). In these languages programs are formulas –written in temporal logic– that state which literals must be true in the next state. So the execution consists in explicitly building the model for the program, state by state. The other group is formed by the languages that are based on the *declarative* approach. The declarative languages extend classical Logic Programming for reasoning about time. However, some of the declarative languages are not purely based on temporal logic due to their extra-logical features for handling eventualities (e.g. [13, 12, 4, 8, 21]). The declarative languages that are purely based on temporal logic extend classical Logic Programming by including temporal connectives in the atoms and by also extending classical resolution ([1, 3, 25, 19, 9, 11]). The languages that belong to the imperative future approach either restrict the use of eventualities (e.g. [18, 17]) or explicitly consider the finite-model property[1], as it is the case in the propositional fragment of MetateM ([2]). The finite-model property is used in order to fix an upper bound of forward chaining steps that are required to fulfill an eventuality. If a model is not obtained bellow this upper bound, then the attempt is given up and the procedure backtracks. The declarative languages that are purely based on temporal logic either directly avoid eventualities ([1, 3, 25, 19, 11]) or do not provide completeness result and still some restrictions on the use of eventualities apply ([9]). If the clausal temporal resolution method presented in [5, 7] were considered as a basis for a declarative temporal logic programming language, its execution would require invariant generation. In the same way, the sequent-based logical foundation for declarative temporal logic programming provided in [20] includes an invariant-based rule. In general, it seems that the troublesome solving (in the resolution sense) of the eventualities (whose fulfillment can be prevented by "hidden" invariants) has been blocking the steps toward more expressive resolution-based declarative TLP languages.

In this article, we present a declarative propositional temporal logic programming language called TeDiLog that is a combination of the temporal and disjunctive paradigms in logic programming. TeDiLog is, syntactically, a sublanguage of the Propositional Linear-time Temporal Logic (PLTL). The TLP language TeDiLog imposes no restrictions on the use of eventualities. To the best of our knowledge, TeDiLog is the first declarative TLP language that achieves this high degree of expressiveness. Hence, TeDiLog is strictly more expressive than the propositional fragments of the above mentioned purely declarative proposals: Templog [1, 3], Chronolog [25, 19], Disjunctive Chronolog [11] and Gabbay's Temporal Prolog [9]. Additionally, TeDiLog is as expressive as propositional MetateM [2]. From the operational point of view, MetateM follows the imperative future approach, i.e. it is not based on reso-

[1] Also known as small model property.

lution. For deciding whether an eventuality is satisfied, the MetateM procedure performs backtracking whenever the upper bound of the size of the (hypothetical) model is exceeded. However, the resolution mechanism of TeDiLog directly manages unsatisfiable eventualities and requires neither backtracking nor the explicit calculation of such an upper bound. We establish the logical foundations of our proposal by formally defining operational and logical semantics for TeDiLog and by proving their equivalence. The operational semantics of TeDiLog relies on a restriction of the invariant-free temporal resolution procedure for PLTL that was introduced in [10]. Consequently, we deal with eventualities without requiring invariant generation. We define a fixpoint semantics that captures the reverse (bottom-up) operational mechanism and prove its equivalence with the logical semantics. We also provide illustrative examples and comparison with other proposals.

References

[1] Martín Abadi & Zohar Manna (1989): *Temporal logic programming*. Journal of Symbolic Computation 8(3), pp. 277–295, doi:10.1016/S0747-7171(89)80070-7.

[2] Howard Barringer, Michael Fisher, Dov M. Gabbay, Graham Gough & Richard Owens (1989): *METATEM: A Framework for Programming in Temporal Logic*. In: Proceedings of the REX (Research and Education in Concurrent Systems) Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, LNCS 430, Springer, pp. 94–129, doi:10.1007/3-540-52559-9_62.

[3] Marianne Baudinet (1989): *Temporal Logic Programming is Complete and Expressive*. In: Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL), ACM Press, pp. 267–280, doi:10.1145/75277.75301.

[4] Christoph Brzoska (1991): *Temporal Logic Programming and its Relation to Constraint Logic Programming*. In: Proceedings of the International Symposium on Logic Programming (ISLP), MIT Press, pp. 661–677.

[5] Michael Fisher (1991): *A Resolution Method for Temporal Logic*. In: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI), Morgan Kaufmann, pp. 99–104.

[6] Michael Fisher (2011): *An Introduction to Practical Formal Methods Using Temporal Logic*. John Wiley & Sons, Ltd, doi:10.1002/9781119991472.

[7] Michael Fisher, Clare Dixon & Martin Peim (2001): *Clausal temporal resolution*. ACM Transactions on Computational Logic 2(1), pp. 12–56, doi:10.1145/371282.371311.

[8] Thom W. Frühwirth (1996): *Temporal Annotated Constraint Logic Programming*. Journal of Symbolic Computation 22(5/6), pp. 555–583, doi:10.1006/jsco.1996.0066.

[9] Dov M. Gabbay (1987): *Modal And Temporal Logic Programming*. In: Temporal Logics And Their Application, Academic Press, pp. 197–237.

[10] Jose Gaintzarain, Montserrat Hermo, Paqui Lucio, Marisa Navarro & Fernando Orejas (2013): *Invariant-Free Clausal Temporal Resolution.* Journal of Automated Reasoning 50(1), pp. 1–49, doi:10.1007/s10817-011-9241-2.

[11] Manolis Gergatsoulis, Panos Rondogiannis & Themis Panayiotopoulos (2000): *Temporal Disjunctive Logic Programming*. New Generation Computing 19(1), pp. 87–102, doi:10.1007/BF03037535.

[12] Tomas Hrycej (1993): *A Temporal Extension of Prolog*. Journal of Logic Programming 15(1 & 2), pp. 113–145, doi:10.1016/0743-1066(93)90016-A.

[13] Shinji Kono, T. Aoyagi, Masahiro Fujita & Hidehiko Tanaka (1985): *Implementation of Temporal Logic Programming Language Tokio*. In: Proceedings of the 4th Conference on Logic Programming (LP), LNCS 221, Springer, pp. 138–147, doi:10.1007/3-540-16479-0_14.

[14] John W. Lloyd (1984): *Foundations of Logic Programming, 1st Edition*. Springer, doi:10.1007/978-3-642-96826-6.

[15] Jorge Lobo, Jack Minker & Arcot Rajasekar (1992): *Foundations of disjunctive logic programming*. MIT Press.

[16] Stephan Merz (1992): *Decidability and incompleteness results for first-order temporal logics of linear time*. *Journal of Applied Non-Classical Logics* 2(2), pp. 139–156, doi:10.1080/11663081.1992.10510779.

[17] Stephan Merz (1995): *Efficiently Executable Temporal Logic Programs*. In: *Proceedings of the IJCAI'93 Workshop on Executable Modal and Temporal Logics*, *LNCS* 897, Springer, pp. 69–85, doi:10.1007/3-540-58976-7_5.

[18] Ben C. Moszkowski (1986): *Executing temporal logic programs*. Cambridge University Press, doi:10.1007/3-540-15670-4_6.

[19] Mehmet A. Orgun (1995): *Foundations of linear-time logic programming*. *International Journal of Computer Mathematics* 58(3-4), pp. 199–219, doi:10.1080/00207169508804444.

[20] Regimantas Pliuskevicius (1992): *Logical Foundation for Logic Programming Based on First Order Linear Temporal Logic*. In: *Proceedings of the First (1990) and Second (1991) Russian Conference on Logic Programming (RCLP)*, *LNCS* 592, Springer, pp. 391–406, doi:10.1007/3-540-55460-2_29.

[21] Takashi Sakuragawa (1986): *Temporal Prolog*. Technical Report, Kyoto University. Available at `http://repository.kulib.kyoto-u.ac.jp/dspace/bitstream/2433/99379/1/0586-16.pdf`.

[22] A. Prasad Sistla & Edmund M. Clarke (1985): *The Complexity of Propositional Linear Temporal Logics*. *Journal of the ACM* 32(3), pp. 733–749, doi:10.1145/3828.3837.

[23] Andrzej Szalas (1995): *Temporal Logic of Programs: A Standard Approach*. In: *Time and Logic. A Computational Approach*, UCL Press Ltd., pp. 1–50.

[24] Andrzej Szalas & Leszek Holenderski (1988): *Incompleteness of First-Order Temporal Logic with Until*. *Theoretical Computer Science* 57, pp. 317–325, doi:10.1016/0304-3975(88)90045-X.

[25] William W. Wadge (1988): *Tense logic programming: a respectable alternative*. In: *Proceedings of the International Symposium on Lucid and Intensional Programming*, pp. 26–32.

Towards an XQuery-based implementation of Fuzzy XPath (Work in Progress)*

Jesús M. Almendros-Jiménez
Dept. of Informatics
Universidad de Almería
04120 Almería (Spain)
jalmen@ual.es

Alejandro Luna
Ginés Moreno
Dept. of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)
Alejandro.Luna@alu.uclm.es
Gines.Moreno@uclm.es

We have recently designed a fuzzy extension of the XPath language which provides ranked answers to flexible queries taking profit of fuzzy variants of *and*, *or* and *avg* operators for XPath conditions, as well as two structural constraints, called *down* and *deep*, for which a certain degree of relevance is associated. In this work, we describe how to implement the proposed fuzzy XPath with the XQuery language. Basically, we have defined an XQuery library able to fuzzily handle XPath expressions in such a way that our proposed fuzzy XPath can be encoded as XQuery expressions. The advantages of our approach is that any XQuery processor can handle a fuzzy version of XPath by using the library we have implemented.

1 Introduction

The *XPath* language [6] has been proposed as a standard for XML querying and it is based on the description of the path in the XML tree to be retrieved. XPath allows to specify the name of nodes (i.e., tags) and attributes to be present in the XML tree together with boolean conditions about the content of nodes and attributes. XPath querying mechanism is based on a Boolean logic: the nodes retrieved from an XPath expression are those matching the path of the XML tree, according to Boolean conditions.

Information retrieval requires the design of query languages able to adapt to user's preferences and providing ranked sets of answers. The *degree of satisfaction* of the user with respect to an answer can be measured in several ways. XPath lacks mechanisms for giving priority to queries and ranking answers. In a XPath-based query, the main criteria to provide a certain degree of satisfaction are the *hierarchical deepness* and *document order*. Moreover, conditions on XPath expressions are usually of varying importance for a user, that is, the user gives a *higher degree of importance to certain requirements* when satisfying his(er) wishes.

With this aim we have recently designed a fuzzy extension of XPath whose main aim is to provide mechanisms to assign priority to queries and to rank answers. Priorities are given by using fuzzy extensions of Boolean operators, while rankings are defined with regard to the location of a tag in the XML tree. Firstly, we have proposed the incorporation to XPath of two structural constraints called *down* and *deep* for which a certain degree of relevance can be associated. So, whereas *down* provides a ranked set of answers depending on the path they are found from "top to down" in the XML document, *deep* provides a ranked set of answers depending on the path they are found from "left to right" in the XML document. Both structural constraints can be used together, assigning degree of importance with respect to the distance to the root XML element. Secondly, we provide fuzzy variants of *and* and *or* for XPath

*This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grants TIN2013-45732-C4-2-P and TIN2013-44742-C4-4-R, as well as by the Andalusian Regional Government (Spain) under Project P10-TIC-6114.

Submitted to:
PROLE 2014

conditions. We have enriched the arsenal of operators of XPath with fuzzy variants of *and* and *or*. Particularly, we have considered three versions of *and*: *and*+, *and*, *and*- (and the same for *or* : *or*+, *or*, *or*-) which make more flexible the composition of fuzzy conditions. Three versions for each operator that come for free from our adaptation of fuzzy logic to the XPath paradigm. One of the most known elements of fuzzy logic is the introduction of fuzzy versions of classical boolean operators. *Product*, *Łukasiewicz* and *Gödel* fuzzy logics are considered as the most prominent logics and give a suitable semantics to fuzzy operators. We claim that in our work the fuzzy versions provide a mechanism to force (and debilitate) conditions in the sense that stronger (and weaker) user preferences can be modeled with the use of stronger (and weaker) fuzzy conditions. The combination of fuzzy operators in queries permits to specify a ranked set of fuzzy conditions according to user's requirements. Finally, we have equipped XPath with an additional operator that is also traditional in fuzzy logic (apart from *min*, *max*, etc.): the average operator *avg*. This operator offers the possibility to explicitly give weight to fuzzy conditions. Rating such conditions by *avg*, solutions increase its weight in a proportional way. However, from the point view of the user's preferences, it forces the user to quantify his(er) wishes which, in some occasions, can be difficult to measure. For this reason, fuzzy versions of *and* and *or* are better choices in some circumstances.

In this work, we describe how to implement our fuzzy variant of the XPath language with the XQuery language. Basically, we have defined an XQuery library able to fuzzily handle XPath expressions in such a way that our proposed fuzzy XPath can be encoded as XQuery expressions. The XQuery library include functions for the *deep* and *down* operators as well as the fuzzy operators *and*+, *and*-, *and*, *or*+, *or*-, *or* and *avg*. Using this library the user can replace Boolean operators by fuzzy versions in XPath expressions, as well as (s)he can call to *deep* and *down* operators to obtain ranked sets of answers. The answers are shown with a *Retrieval State Value (RSV)* representing the degree of satisfaction. They can be also ordered with respect to the RSV in XQuery making use of *descending* expression, as well as filtered with regard to a *threshold*.

Our approach is not intended to be focused on handling XML documents with fuzzy information. The input documents in our proposal are XML documents with crisp information, but the answers of a query offers fuzzy information, that is, a RSV of each answer. Therefore our approach is focused on the handling of standard XML documents in which we can retrieve information ranked by a certain degree of satisfaction. In other words, our library can be used from any XQuery processor to query any XML document with crisp information.

Although the input of a query is a crisp XML document, the library assign internally and, in a transparent way to the user, a RSV to each of node of interest in the document. The RSVs assigned to each node of interest are used to compute the RSV of the answer. It makes the implementation a non-trivial task. Starting from a crisp XML document as input, our implementation annotates at run-time a RSV to each node of the query result. It also involves to dynamically annotate RSVs of nodes in subqueries. Additionally, *where* and *return* expressions of XQuery become XQuery functions in order to handle fuzzy conditions and RSVs, respectively.

The structure of the paper is as follows. Section 2 will describe the fuzzy version of XPath. Section 3 will show some examples of fuzzy XPath. Section 4 will present the implementation in XQuery. Section 5 will show the same of examples of Section 3, written in XQuery. Finally, Section 6 will conclude and present future work.

Figure 1: Fuzzy XPath Grammar

xpath :=	['['deep-down']']path
path :=	literal | text() | node | @att | node/path | node//path
node :=	QName | QName[cond]
cond :=	xpath op xpath | xpath num-op number
deep :=	**DEEP**=number
down :=	**DOWN**=number
deep-down :=	deep | down | deep '**;**' down
num-op :=	**>** | **=** | **<** | **<>**
fuzzy-op :=	**and** | **and+** | **and-** | **or** | **or+** | **or-** | **avg** | **avg**{number,number}
op :=	num-op | fuzzy-op

2 A Flexible XPath Language

Our proposal for flexible XPath is defined by the grammar of Figure 1. Basically, the extension of XPath is as follows:

- A given XPath expression can be adorned with «[DEEP = r_1; DOWN = r_2]» which means that the *deepness* of elements is penalized by r_1 and that the *order* of elements is penalized by r_2, and such penalization is proportional to the distance (i.e., the length of the branch and the weight of the tree, respectively). In particular, «[DEEP = 1; DOWN = r_2]» can be used for penalizing only w.r.t. document order. *deep* works for //, which in XPath retrieves all the descendant nodes, that is, the deepness in the XML tree is only computed when descendant nodes are explored, while *down* works for both / and // (i.e., direct and non-direct descendant nodes). Let us remark that *deep* and *down* can be used anywhere, and many times in an XPath expression.
- We consider three versions for each one of the conjunction and disjunction operators (also called connectives or aggregators) which are based in the so-called *Product*, *Gödel* and *Łukasiewicz* fuzzy logics. The *Gödel* and *Łukasiewicz* logic based fuzzy symbols [1] are represented in our application by *and+*, *and-*, *or-* and *or+*, in contrast with product logic operators *and* and *or* (see Figure 2). Adjectives like *pessimistic*, *realistic* and *optimistic* are sometimes applied to the *Łukasiewicz*, *Product* and *Gödel* fuzzy logics since operators satisfy that, for any pair of real numbers x and y in $[0,1]$: $0 \leq \&_{\mathtt{L}}(x,y) \leq \&_{\mathtt{P}}(x,y) \leq \&_{\mathtt{G}}(x,y) \leq 1$ and the contrary for the disjunction operations (as used in MALP): $0 \leq |_{\mathtt{G}}(x,y) \leq |_{\mathtt{P}}(x,y) \leq |_{\mathtt{L}}(x,y) \leq 1$. So, note that it is more difficult to satisfy a condition based on a pessimistic conjuntor/disjunctor (i.e, *and-/or-* inspired by the *Łukasiewicz* and *Gödel* logics, respectively) than with *Product* logic based operators (i.e, *and/or*), while the optimistic versions of such connectives (i.e., *and+/or+*) are less restrictive, obtaining a greater set of answers. This is a consequence of the following chain of inequalities: $0 \leq and-(x,y) \leq and(x,y) \leq and+(x,y) \leq or-(x,y) \leq or(x,y) \leq or+(x,y) \leq 1$. Therefore users should refine queries by choosing operators in the previous sequence from left to right (or from right to left), till finding solutions satisfying in a stronger (or weaker) way the requirements.
- Finally, the *avg* operator is defined too in a *weighted* way. Assuming two given *RSV*'s r_1 and r_2, *avg* is defined as $(r_1 + r_2)/2$, and $avg\{a,b\}$ is defined as $(a * r_1 + b * r_2)/a + b$.

[1] The fuzzy logic community frequently uses the terms *t-norm* and *t-conorm* for expressing generalized versions of conjunctions and disjunctions.

Figure 2: Fuzzy Logical Operators

$\&_{\mathtt{P}}(x,y)$	=	$x*y$	$\|_{\mathtt{P}}(x,y)$	=	$x+y-x*y$	*Product: and/or*
$\&_{\mathtt{G}}(x,y)$	=	$\min(x,y)$	$\|_{\mathtt{G}}(x,y)$	=	$\max(x,y)$	*Gödel: and+/or-*
$\&_{\mathtt{L}}(x,y)$	=	$\max(x+y-1,0)$	$\|_{\mathtt{L}}(x,y)$	=	$\min(x+y,1)$	*Łukasiewicz: and-/or+*

Figure 3: Input XML document collecting Hotel's information

```
<hotels>
<hotel name="Melia">
    <close_to>Gran Via
        <close_to>Callao</close_to>
        <close_to>Plaza de Espana</close_to>
    </close_to>
    <services>
        <pool></pool>
        <metro>150</metro>
    </services>
    <price>100</price>
</hotel>
<hotel name="NH">
    <close_to>Sol
        <close_to>Gran Via</close_to>
        <close_to>Callao</close_to>
    </close_to>
    <services>
        <metro>300</metro>
    </services>
    <price>150</price>
</hotel>
<hotel name="Hilton">
    <close_to>Moncloa
        <close_to>Gran Via</close_to>
        <close_to>Sol</close_to>
    </close_to>
    <services>
        <metro>150</metro>
    </services>
    <price>50</price>
</hotel>
<hotel name="Tryp">
    <close_to>Cibeles
        <close_to>Alcala
            <close_to>Gran Via</close_to>
        </close_to>
        <close_to>Retiro</close_to>
    </close_to>
    <services>
        <pool></pool>
        <metro>10</metro>
    </services>
    <price>575</price>
</hotel>
<hotel name="Sheraton">
    <close_to>Recoletos
        <close_to>Cibeles</close_to>
        <close_to>Gran Via
            <close_to>Sol</close_to>
        </close_to>
    </close_to>
    <close_to>Sol</close_to>
    <services>
        <pool></pool>
        <metro>300</metro>
    </services>
    <price>475</price>
</hotel>
</hotels>
```

In general, an extended XPath expression defines, w.r.t. an XML document, a sequence of subtrees of the XML document where each subtree has an associated RSV. XPath conditions, which are defined as fuzzy operators applied to XPath expressions, compute a new RSV from the RSVs of the involved XPath expressions, which at the same time, provides an RSV to the node.

2.1 Examples of Fuzzy XPath

In order to illustrate the language, let us see some examples of flexible queries in XPath. We will take as input document the one shown in Figure 3. The example shows a sequence of hotels where each one is described by *name* and *price*, proximity to streets (*close_to*) and provided services (*pool* and *metro* -together with distance-). In the example, we assume that document order has the following semantics. The tag *close_to* specifies the proximity to a given street. However, the order of *close_to* tags is relevant, and the top streets are closer than the streets at the bottom. In other words, the case:

```
hotel_H
        close_to street_A
        close_to street_B
```

implies that hotel H is near to both streets A and B, but closer to A than to B. The nesting of *close_to* has also a relevant meaning. While a given street A can be close to the hotel H, the streets close to A are not necessarily close to the hotel H. In other words, in the case:

```
hotel_H
        close_to street_A
                    close_to street_B
```

the street B is near to street A, and street A is close to hotel H, which implies that street B is also close to hotel H, but no so close as street A. H can be situated at the end of street A, and B can cross A at the beginning. We can say, in this case, that B is an *adjacent* street to H, while A is *close* to H. This means that when looking for a hotel close to a given street, the highest priority should be assigned to streets close to the hotel, while adjacent streets should be relegated to lower priority. The example has been modeled in order to illustrate the use of structural constraints and fuzzy operators. Particularly, when the user tries to find hotels very close to a given street it should be provided a high *down* value and a low *deep* value, whereas in the case the user tries to find hotels in the neighborhood of an street should provide high *deep* and low *down*.

In our first example, we focus on the use of *down*. Let us now suppose that the user is interested to find a hotel close to Sol street. This might be his(er) first tentative looking for a hotel. Using crisp XPath (s)he would formulate:

$$<< /hotels/hotel[close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

However, it gives the user the set of hotels close to Sol without distinguishing the degree of proximity. The fuzzy version of XPath permits to specify a degradation of answers, in such a way that the user reformulates the query as:

$$<< /hotels/hotel[[DOWN = 0.9]close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

The query specifies that *close_to* tag is degraded by 0.9 from top to down. In other words, when Sol is found close to a hotel, the position in which it occurs gives a different satisfaction value. In this case, we will obtain:

```
<result>
  <result rsv="1.0">NH</result>
  <result rsv="0.9">Sheraton</result>
</result>
```

Fortunately, we have found a hotel (NH) which is very close to Sol, and one (Sheraton) which is a little bit farther from Sol. Let us remark the previous example and the other examples of the Section show the results in order of satisfaction degree.

Let us now suppose that we are looking for a hotel close to Callao. In this case, we can try to make the same question:

$$<< /hotels/hotel[[DOWN = 0.9]close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

However, the result is empty. Therefore we can try to relax the query by changing '/' by '//':

$$<< /hotels/hotel[[DOWN = 0.9]//close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

Now, we will find answers, however, we will not be able to distinguish the proximity of the hotels. Our fuzzy version of XPath permits to specify how the solutions are degraded but not only taking into account the order but also the deepness. In other words, there would be useful to give different weights to be a close street, and to be an adjacent street. Therefore we can use the query:

$$<< /hotels/hotel[[DEEP = 0.5; DOWN = 0.9]//close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

obtaining the following results:

```
<result>
  <result rsv="0.5">Melia</result>
  <result rsv="0.45">NH</result>
</result>
```

Thus Melia is near to Callao, and NH is a little bit farther than Melia.

The use of *deep* combined with *down* could be considered as the best choice. However, *deep* can be used alone when the user only wants to penalize adjacency. If we like to search hotels near to Gran Via street, degrading adjacent streets with a factor of 0.5, we can consider the following query (and we obtain the following result):

$$<< //hotel[[DEEP = 0.5]//close\_to/text() = \text{“}Gran\ Via\text{”}]/@name >>$$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.5">NH</result>
  <result rsv="0.5">Hilton</result>
  <result rsv="0.5">Sheraton</result>
  <result rsv="0.25">Tryp</result>
</result>
```

We can see that *Melia* is close to *Gran Via*, while *NH*, *Hilton* and *Sheraton* are situated in adjacent streets of *Gran Via*. *Tryp* is the farthest hotel. Let us now suppose that the user is interested in a hotel combining two services like pool and metro. Instead of using classical *and/or* connectives for mixing both features, we can obtain more flexible estimations on *RSV* values by using the *avg* operator as follows:

$$<< //hotel[services/pool\ avg\ services/metro]/@name >>$$

thus obtaining the following results:

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="1.0">Tryp</result>
  <result rsv="1.0">Sheraton</result>
  <result rsv="0.5">NH</result>
  <result rsv="0.5">Hilton</result>
</result>
```

By using the *avg* fuzzy operator, the user finds that *Melia*, *Tryp* and *Sheraton* have pool and metro, while *NH* and *Hilton* lack on one of them.

Let us now suppose that the importance of the metro is the double of the importance of the pool. In this case, the user can formulate the query as follows:

$$<< //hotel[services/pool\ avg\{1,2\}\ services/metro]/@name >>$$

obtaining the following results:

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="1.0">Tryp</result>
  <result rsv="1.0">Sheraton</result>
  <result rsv="0.666667">NH</result>
  <result rsv="0.666667">Hilton</result>
</result>
```

We can see in the results that NH and Hilton increase the degree of satisfaction w.r.t. the previous query given that they have metro station.

Let us now suppose the user is looking now for hotels giving more importance to the fact that the price of the hotel is lower than 150 euros than to the proximity to Sol street. The user can formulate the query as follows, obtaining the results below:

$$<< //hotel[[DEEP = 0.8]//close\_to/text() = \text{“}Sol\text{”}\ avg\{1,2\}\ //price/text() < 150]/@name >>$$

```
<result>
  <result rsv="0.933333">Hilton</result>
  <result rsv="0.666667">Melia</result>
  <result rsv="0.333333">NH</result>
  <result rsv="0.333333">Sheraton</result>
</result>
```

In the following queries we express the following requirement: hotels near to Gran Via, near to a metro station, having pool, with greater preference (3 to 2) to pool than metro. We will use *and+*, *and* and *and-* which provide distinct levels of exigency, which are demonstrated in the results.

$$<< //hotel[([DEEP = 0.5]//close\_to/text() = \text{”}Gran\ Via\text{”})\ and+(//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.5">Sheraton</result>
  <result rsv="0.4">Hilton</result>
  <result rsv="0.25">Tryp</result>
</result>
```

$$<< //hotel[([DEEP = 0.5]//close\_to/text() = \text{”}Gran\ Via\text{”})\ and(//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.3">Sheraton</result>
  <result rsv="0.25">Tryp</result>
  <result rsv="0.2">Hilton</result>
</result>
```

$$<< //hotel[([DEEP = 0.5]//close\_to/text() = \text{”}Gran\ Via\text{”})\ and-(//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$$

```
<result>
  <result rsv="1.0">Melia</result>
  <result rsv="0.25">Tryp</result>
  <result rsv="0.1">Sheraton</result>
</result>
```

So, in the first case (the least demanding and optimistic) we obtain four hotels (Melia, Sheraton, Hilton and Tryp), as well as in the second case (a little bit more exigent) while third table (the strongest one) lists three candidates (Melia, Tryp and Sheraton). Sheraton and Hilton are degraded using *and* and *and-*.

3 XQuery Library for Fuzzy XPath

We can summarize the elements of the implementation as follows:

3.1 Elements of the Library

1. The *deep* and *down* operators become XQuery functions that take as arguments a context node, an XPath expression and the value (a real number in [0,1]) assigned to deep and down, respectively. For combining deep and down an XQuery function is defined having as argument two real values in [0,1]:

```
declare function f:deep($node,$xpath,$deep)
declare function f:down($node,$xpath,$down)
declare function f:deep_down($node,$xpath,$deep,$down)
```

2. Fuzzy versions of Boolean operators *and*, *or* have been defined as XQuery functions, each one for each fuzzy logic we have considered (i.e., *Product*, *Łukasiewicz* and *Gödel*):

```
declare function f:andP($left,$right)
declare function f:orP($left,$right)
declare function f:andG($left,$right)
declare function f:orG($left,$right)
declare function f:andL($left,$right)
declare function f:orL($left,$right)
```

3. Operators *avg* and *avg{a,b}* have been defined as XQuery functions:

```
declare function f:avg($left,$right)
declare function f:avg_ab($left,$right,$a,$b)
```

4. Fuzzy versions of XQuery expressions *where* and *return* have been defined. In order to make transparent to the user the incorporation of RSVs, we have defined a new version of the *return* expression, called *returnF*, which transparently carries out the computation of the RSVs of the answers. Similarly, since XQuery works with a Boolean logic, the introduction of fuzzy versions of the operators, force us to define a new version of the *where* expression, called *whereF*, which transparently carries out the computation of the RSVs from fuzzy conditions. *ReturnF* has as parameters the context node and an XPath expression. *WhereF* has as parameters the context node and a fuzzy condition.

```
declare function f:whereF($node,$fuzzycond)
declare function f:returnF($node,$xpath)
```

5. Fuzzy versions of comparison operators for XPath expressions have been defined as XQuery functions. Similarly to *whereF*, comparison operators have been adapted to handle the RSVs:

```
declare function f:equalF($left,$right)
declare function f:lessF($left,$right)
declare function f:greaterF($left,$right)
```

3.2 Implementation of the Library

In order to implement our library in XQuery we have used the *XQuery Module* available in the *BaseX* processor [17]. In particular, we make use of the function *eval* that makes possible the manipulation of

XPath expressions. This function is also available for *Exist* [20] and *Saxon* [18] processors. For instance, *down* is defined as follows:

```
declare function f:down($nodes,$query, $down){
  let $docDown := document{f:down_aux($nodes/*,$down,(),())}
  let $docQ := xquery:eval(concat('$x',$query), map { '$x' :=$docDown})
  let $docL := xquery:eval(concat('$x',$query), map { '$x' :=$nodes})
  return f:putListRSV($docL,f:getListRSV($docQ))
};
```

deep is defined as follows:

```
declare function f:deep($doc as node()*, $query, $deep as xs:double){
  let $docDeep := document{f:deep_aux($doc/*,$deep,1)}
  let $docQ := xquery:eval(concat('$x',$query), map { '$x' :=$docDeep})
  let $docL := xquery:eval(concat('$x',$query), map { '$x' :=$doc})
  return f:putListRSV($docL,f:getListRSV($docQ))
};
```

and *deep_down* is defined as follows:

```
declare function f:deep_down($nodes as node()*,$query, $deep as xs:double, $down as xs:
    double){
  let $docDown := document{f:down_aux($nodes/*,$down,(),())}
  let $docDeep := document{f:deep_aux($docDown/*,$deep,1)}
  let $docQ := xquery:eval(concat('$x',$query), map { '$x' :=$docDeep})
  let $docL := xquery:eval(concat('$x',$query), map { '$x' :=$nodes})
  return f:putListRSV($docL,f:getListRSV($docQ))
};
```

Each fuzzy operator has been defined as a function, for instance, *and* (*Product* logic), *or+* (*Gödel* logic), *avg*, and *avg{a,b}* are defined as follows:

```
declare function f:andP($cond1,$cond2)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return $tv1*$tv2
};

declare function f:orG($cond1,$cond2)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return
    if ($tv1 > $tv2) then $tv1
    else $tv2
};

declare function f:avg($cond1,$cond2)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return (xs:double($tv1)+xs:double($tv2)) div (2)
};

declare function f:avg_ab($cond1,$cond2, $a, $b)
{
  let $tv1 := f:truthValue($cond1)
  let $tv2 := f:truthValue($cond2)
  return (xs:double($tv1)*$a+xs:double($tv2)*$b) div ($a+$b)
};
```

4 Examples of Fuzzy XPath in XQuery

Now, we show how the previous fuzzy XPath queries can be written in XQuery. Let us now suppose the following fuzzy XPath query:

$$<< /hotels/hotel[[DOWN = 0.9]close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

We can now write the same query in XQuery as follows:

```
for $x in doc('hotels.xml')/hotels/hotel
let $y := f:whereF($x,f:equalF(f:down($x,'/close_to',0.9),'Sol'))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

We can see that fuzzy XPath expressions are written as XQuery expressions. This is the same kind of transformation from crisp XPath to XQuery. For instance:

$$<< /hotels/hotel[close\_to/text() = \text{“}Sol\text{”}]/@name >>$$

can be translated into:

```
for $x in doc("hotels.xml")/hotels/hotel
where $x/close_to/text()="Sol"
return $x/@name
```

In the fuzzy case, *"="* is transformed into *equalF*, and *where* as well as *return* become XQuery functions, with an extra argument to represent the context node. The query makes use of the function *down* of the library to compute the RSVs associated to *close_to*. In addition, the attribute *rsv*, which has been (internally) added to the output document, can be handled to show the answer in a sorted way, and even to define a threshold.

Let us now consider the following query, that uses *deep* and *down*:

$$<< /hotels/hotel[[DEEP = 0.5; DOWN = 0.9]//close\_to/text() = \text{“}Callao\text{”}]/@name >>$$

We can now write the same query in XQuery using the function *deep_down*:

```
for $x in doc('hotels.xml')/hotels/hotel
let $y :=
  f:whereF($x, f:equalF(f:deep_down($x,'//close_to',0.5,0.9),'Callao'))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

Let us now suppose the following fuzzy XPath expression that makes use of the *avg* operator.

$$<< //hotel[services/pool\ avg\ services/metro]/@name >>$$

Here, we use the function *avg* of the library, having as parameters both sides of the fuzzy condition:

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x, f:avg($x/services/pool,$x/services/metro))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

The same can be said for the following query, using *avg{a,b}* having as parameters a and b.

$$<< //hotel[services/pool\ avg\{1,2\}\ services/metro]/@name >>$$

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x,f:avg_ab($x/services/pool, $x/services/metro,1,2))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

Let us now suppose the following queries that combine *deep* and *avg*, and *deep* and *and*+, respectively:

$$<< //hotel[[DEEP = 0.8]//close\_to/text() = \text{“}Sol\text{”} avg\{1,2\}\ //price/text() < 150]/@name >>$$

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x, f:avg_ab(f:equalF(f:deep($x,'//close_to',0.8),'Sol'),
      $x//price/text()<150,1,2))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

$$<< //hotel[([DEEP = 0.5]//close\_to/text() = \text{”}Gran\ Via\text{”}) and+(//pool\ avg\{3,2\}\ //metro/text() < 200)]/@name >>$$

```
for $x in doc('hotels.xml')//hotel
let $y := f:whereF($x,f:andG(f:equalF(f:deep($x,'//close_to',0.5),'GranVia'),
      f:avg_ab($x//pool,$x//metro<200,3,2)))
let $z := f:returnF($y,'/@name')
order by $y/@rsv descending
return $z
```

4.1 Benchmarks

Now, we would like to show the benchmarks we have obtained using our library. We have tested our library using data sets of different sizes. We have used as data sets traces of execution of MALP programs developed under our FLOPER tool. The FLOPER tool generates traces in XML format, with a high degree of tag nesting when a recursive program is executed. These data sets facilitate the testing of our structural based operators *deep* and *down*.

In Figure 4 we can see the results, where we indicate the number of nodes examined in each tree, as well as the depth of the tree. We have compared the execution times for two XPath expressions in crisp and fuzzy versions. The first query is Q1:

$$<< //node/goal >>$$

and the second query is Q2:

$$<< //node[goal[contains(text(),\text{”}p(\text{”})]\ and\ substitution[contains(text(),\text{”}g(\text{”})]]//goal >>$$

We have used the BaseX Query processor in a Intel Core 2 Duo 2.66 GHz Mac OS machine.

4.2 Related Work

Fuzzy logic plays a key role in information retrieval and the need for providing fuzzy/flexible mechanisms to XML querying has recently motivated the investigation of extensions of the XQuery/XPath language. We can distinguish those in which the main goal is the introduction of fuzzy information in data (similarity, proximity, vagueness, etc) [25, 26, 8, 7, 15, 27, 23, 24] and the proposals in which the main goal is the handling of crisp information by fuzzy concepts [16, 9, 10, 13, 12, 19, 11]. Our work focuses on the second line of research.

Fuzzy versions of XQuery have been previously studied in some works. The closest to our approach is [16], in which preferences can be described by queries in order to retrieve discriminated answers by user's preferences. FLOWR expressions are extended to cover with fuzzy values and answers. The main aim of their work is to extend XQuery with definition of fuzzy terms: good, cheap, high, young, etc.,

Figure 4: Benchmarks

Query	*16Kb*	*700Kb*	*4.8Mb*	*15.4Mb*
Examined nodes in Q1	*28*	*148*	*298*	*448*
Examined nodes in Q2	*25*	*145*	*295*	*445*
Tree Depth	*21*	*101*	*201*	*301*
Q1	7.09 ms	25.47 ms	123.66 ms	461.6 ms
down in Q1	12.52 ms	107.1 ms	481.24 ms	2853.36 ms
deep in Q1	10.08 ms	74.17 ms	510.74 ms	1953.31 ms
deep and down in Q1	69.97 ms	102.0 ms	685.87 ms	7315.59 ms
Q2	5.77 ms	57.96 ms	172.03 ms	529.18 ms
avg in Q2	36.59 ms	1266.99 ms	9729.49 ms	60426.28 ms

defined as fuzzy predicates that can be imposed in XPath expressions. They extend XQuery datatypes with *xs:truth* and incorporate *xml:truth* as attribute to represent degree of satisfaction. Nevertheless, they lack an implementation, and therefore we cannot compare our proposal with theirs, although we believe that a similar technique we have proposed here can be used. In [25], they also extend the syntax of XQuery, in particular, the expression *where* to cover with priority and thresholding. Their approach is focused on querying fuzzy XML data, and therefore their proposal is different from ours. They have developed an implementation using Java on top of the Exist [20] XQuery processor. A fuzzy query is transformed into standard XQuery to be executed. Fuzzy data querying is also the main aim of the work of [26], in which they propose a fuzzy XML Schema and algebraic operators to handle fuzzy data over an schema. They provide transformations from the algebraic operators to XQuery (and XPath) expressions. Again, their approach is different from ours, since they work with fuzzy XML data as input.

Fuzzy versions of XPath have been previously studied in some works. The closest works to our proposal are [9, 10] in which authors introduce in XPath flexible matching by means of fuzzy constraints called *close* and *similar* for node content, together with *below* and *near* for path structure. In addition, they have studied the *deep-similar* notion for tree matching, and fuzzy versions for *not*, *and* and *or* operators. In order to provide ranked answers they assign a *RSV* to each item. Our work is similar to the proposed by [9, 10]. The *below* operator of [9, 10] is equivalent to our proposed *down*: both extract elements that are direct descendants of the current node, and the penalization is proportional to the distance. The *near* operator of [9, 10], which is defined as a generalization of *below*, ranks answers depending on the distance to the required node, in any XPath axis. Our proposed *deep* ranks answers depending of the distance to the current node, but the considered nodes can be direct and non direct descendants. Therefore our proposed *deep* combined with *down* is a particular case of *near*. To have the same expressivity power as *near* we could incorporate to our framework a new operator to rank answers from bottom to up. With respect to *similar* and *close* operators proposed in [9, 10], our framework lacks similarity relations and rather focuses on structural (i.e. path-based) flexibility.

In [13], the authors propose to give a satisfaction degree to XPath expressions based on associating weights to XPath steps. Relaxing XPath expressions when the path does not match the XML schema is the main goal of this work. They have studied how to compute the best k answers. In this line, in [12, 14] XPath relaxation is studied given some rules for query rewriting: axis relaxation, step deletion and step cloning, among others. The proposed *deep-similar* notion of [9, 10] can be also considered a relaxation technique of XML tree equality. Our work has some similarities with these proposals: *deep* and *down*, and also the use of *avg* operator, are mechanisms for relaxing queries and giving priority to paths and

answers. We have also studied in [4] how to introduce axis relaxation, step deletion and step cloning in our approach, but the proposed implementation does not still include these mechanisms. It is considered as future work.

Finally, let us remark that we have previously developed [3, 2, 4, 5] an implementation[2] of our fuzzy XPath using the *FLOPER "Fuzzy LOgic Programming Environment for Research"* tool[3] which is based on Multi-Adjoint Logic Programming (MALP) [21, 22]. There we made use of the fuzzy logic nature of FLOPER to implement fuzzy XPath by using fuzzy logic rules. Here the implementation has to adapt a Boolean logic based language (i.e., XQuery) to obtain the same behavior as in MALP. The implementation in XQuery can be download from `http://dectau.uclm.es/fuzzyXPath/`.

5 Conclusions and Future Work

In this paper we have presented an implementation of a fuzzy version of XPath by using an XQuery library. Fuzzy XPath incorporates mechanisms to rank answers depending on the location of the item in the XML tree of input, as well as to give priority to queries. The output of a query contains a RSV in each item according to the user's preferences. We have described the elements of the XQuery library that make possible to express fuzzy queries against crisp XML data. As future work, we plan the following steps. Firstly, to incorporate new mechanisms of searching and ranking to queries. We have previously studied [4, 5] how to penalize answers when a given XPath expression is incorrect, and tags have to be jumped, switched and added. We believe that these mechanisms can be implemented also in XQuery. Secondly, we would like to extend our work to other fuzzy logic mechanism (vagueness, similarity, etc). Finally, we would like to improve the performance of our implementation, for instance, in the use of thresholding. Up to now, thresholding is achieved on the output of the query, and dynamic thresholding would improve the performance. Dynamic thresholding has been already used in our MALP-based implementation [1].

6 Bibliography

References

[1] Jesús M. Almendros-Jiménez, Alejandro Luna-Tedesqui & Ginés Moreno (2014): *Dynamic Filtering of Ranked Answers When Evaluating Fuzzy XPath Queries*. In: *Rough Sets and Current Trends in Soft Computing*, *Lecture Notes in Computer Science* 8536, Springer International Publishing, pp. 319–330. Available at `http://dx.doi.org/10.1007/978-3-319-08644-6_33`.

[2] Jesús Manuel Almendros-Jiménez, Alejandro Luna & Ginés Moreno (2012): *Fuzzy Logic Programming for Implementing a Flexible XPath-based Query Language*. *Electr. Notes Theor. Comput. Sci.* 282, pp. 3–18. Available at `http://dx.doi.org/10.1016/j.entcs.2011.12.002`.

[3] J.M. Almendros-Jiménez, A. Luna & G. Moreno (2011): *A Flexible XPath-based Query Language Implemented with Fuzzy Logic Programming*. In: *Proc. of 5th International Symposium on Rules: Research Based, Industry Focused, RuleML'11*, Springer Verlag, LNCS 6826, pp. 186–193.

[4] J.M. Almendros-Jiménez, A. Luna & G. Moreno (2012): *A XPath Debugger Based on Fuzzy Chance Degrees*. In P. Herrero et al., editor: *On the Move to Meaningful Internet Systems: Proceedings OTM 2012 Workshops, Rome, Italy, September 10-14*, Springer Verlag, LNCS 7567, pp. 669–672. Available at `http://dx.doi.org/10.1007/978-3-642-33618-8_91`.

[2] `http://dectau.uclm.es/fuzzyXPath/`

[3] `http://dectau.uclm.es/floper`.

[5] J.M. Almendros-Jiménez, A. Luna & G. Moreno (2013): *Annotating Fuzzy Chance Degrees when Debugging XPath Queries*. In: *Proc. of the 12th International Work-Conference on Artificial Neural Networks, IWANN'13*, Springer Verlag, LNCS 7903, Part II, pp. 300–311.

[6] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie & J. Siméon (2007): *XML path language (XPath) 2.0*. *W3C*.

[7] P. Buche, J. Dibie-Barthélemy, O. Haemmerlé & G. Hignette (2006): *Fuzzy semantic tagging and flexible querying of XML documents extracted from the Web*. *Journal of Intelligent Information Systems* 26(1), pp. 25–40.

[8] P. Buche, J. Dibie-Barthélemy & F. Wattez (2006): *Approximate querying of XML fuzzy data*. *Flexible Query Answering Systems*, pp. 26–38.

[9] A. Campi, E. Damiani, S. Guinea, S. Marrara, G. Pasi & P. Spoletini (2009): *A fuzzy extension of the XPath query language*. *Journal of Intelligent Information Systems* 33(3), pp. 285–305.

[10] E. Damiani, S. Marrara & G. Pasi (2007): *FuzzyXPath: Using fuzzy logic and IR features to approximately query XML documents*. *Foundations of Fuzzy Logic and Soft Computing*, pp. 199–208.

[11] E. Damiani, S. Marrara & G. Pasi (2008): *A flexible extension of XPath to improve XML querying*. In: *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, pp. 849–850.

[12] B. Fazzinga, S. Flesca & F. Furfaro (2010): *On the expressiveness of generalization rules for XPath query relaxation*. In: *Proceedings of the Fourteenth International Database Engineering & Applications Symposium*, ACM, pp. 157–168.

[13] B. Fazzinga, S. Flesca & A. Pugliese (2009): *Top-k Answers to Fuzzy XPath Queries*. In: *Database and Expert Systems Applications*, Springer, pp. 822–829.

[14] Bettina Fazzinga, Sergio Flesca & Filippo Furfaro (2011): *Xpath query relaxation through rewriting rules*. *Knowledge and Data Engineering, IEEE Transactions on* 23(10), pp. 1583–1600.

[15] A. Gaurav & R. Alhajj (2006): *Incorporating fuzziness in XML and mapping fuzzy relational data into fuzzy XML*. In: *Proceedings of the 2006 ACM symposium on Applied computing*, ACM, pp. 456–460.

[16] Marlene Goncalves & Leonid Tineo (2010): *Fuzzy XQuery*. In: *Soft Computing in XML Data Management*, Springer, pp. 133–163.

[17] Christian Grün (2014): *BaseX. The XML Database*. `http://basex.org`.

[18] Michael Kay (2008): *Ten Reasons Why Saxon XQuery is Fast*. *IEEE Data Eng. Bull.* 31(4), pp. 65–74.

[19] H.G. Li, S.A. Aghili, D. Agrawal & A. El Abbadi (2006): *FLUX: fuzzy content and structure matching of XML range queries*. In: *Proceedings of the 15th international conference on World Wide Web*, ACM, pp. 1081–1082.

[20] Wolfgang Meier (2003): *eXist: An open source native XML database*. In: *Web, Web-Services, and Database Systems*, Springer, pp. 169–183.

[21] P.J. Morcillo & G. Moreno (2008): *Programming with Fuzzy Logic Rules by using the FLOPER Tool*. In Nick Bassiliades et al., editor: *Proc of the 2nd. Rule Representation, Interchange and Reasoning on the Web, International Symposium, RuleML'08*, Springer Verlag, LNCS 3521, pp. 119–126.

[22] P.J. Morcillo, G. Moreno, J. Penabad & C. Vázquez (2010): *A Practical Management of Fuzzy Truth Degrees using FLOPER*. In M. Dean et al., editor: *Proc. of 4nd Intl Symposium on Rule Interchange and Applications, RuleML'10*, Springer Verlag, LNCS 6403, pp. 20–34.

[23] B. Oliboni & G. Pozzani (2008): *Representing fuzzy information by using XML schema*. In: *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on*, IEEE, pp. 683–687.

[24] B. Oliboni & G. Pozzani (2010): *An XML Schema for Managing Fuzzy Documents*. *Soft Computing in XML Data Management*, pp. 3–34.

[25] P Ueng & S Skrbic (2012): *Implementing XQuery fuzzy extensions using a native XML database*. In: *Computational Intelligence and Informatics (CINTI), 2012 IEEE 13th International Symposium on*, IEEE, pp. 305–309.

[26] Ekin Üstünkaya, Adnan Yazici & Roy George (2007): *Fuzzy data representation and querying in xml database*. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 15(supp01), pp. 43–57.

[27] L. Yan, ZM Ma & J. Liu (2009): *Fuzzy data modeling based on XML schema*. In: *Proceedings of the 2009 ACM symposium on Applied Computing*, ACM, pp. 1563–1567.

Modelling Hybrid Systems on a Concurrent Constraint Paradigm (Work in Progress)

Damián Adalid María del Mar Gallardo
Dept. Lenguajes y Ciencias de la Computación
E.T.S.I. Informática
University of Málaga*
damian@lcc.uma.es gallardo@lcc.uma.es

Complex reactive systems and, in particular, hybrid systems that combine discrete and continuous dynamics, require quality modelling languages to be either described or analyzed. Concurrent constraint programming (ccp) is a highly expressive declarative programming paradigm, characterized by the use of a common *constraint store* to communicate and synchronize agents. Thus, in this paradigm, data are stated in the form of constraints, in contrast to the usual variable/value mechanism of imperative languages. The goal of this paper is to explore the expressive capacity of the ccp paradigm to describe hybrid systems. In particular, we have defined HY-tccp as an extension of the *timed concurrent constraint language* tccp. The new language completely subsumes tccp, and includes new agents, and a new notion of store, able to describe the continuous dynamics of hybrid systems. In this paper, we present the semantics of HY-tccp as an extension of that of tccp, and some examples that show the expressiveness of the new language.

1 Introduction

Modelling and analysis of reactive systems, including those dealing with real-time, is an active research line in the formal method community. Reactive systems usually display essential time properties [2], since they react continuously to their environment and might be subjected to temporal aspects or, at least, to a necessary order of execution [17]. Several different approaches have been specifically developed to describe reactive systems with the final objective of constructing reliable systems at the end of the development and validation phases.

For instance, classic process algebras, such as CSP [16] and CCS [20], are probably one of the most known families of related approaches for modelling concurrent systems. They provide compositional mechanisms to describe the interaction between independent processes, also called agents. Timed process algebras [23, 21, 8, 6] are more powerful algebras which make it possible to describe and analyze time-related features of concurrent systems.

Alternatively, the synchronous reactive programming paradigm [13] has led to the so-called synchronous languages, which are particularly suitable for modelling reactive systems. *SIGNAL* [19], *Statecharts* [14], *ESTEREL* [3] and *LUSTRE* [12] are well-known synchronous languages. Apart from their particular characteristics, all of them have in common the notion of logical time. A synchronous program reacts to its environment in a sequence of ticks, and computations within a tick are assumed to be in zero-time, *i.e.* instantaneous. This allows them to have a deterministic semantics which is ideal for verification and formal analysis.

*This work has been supported by Andalusian Excellence Project P11-TIC7659 and Spanish Ministry of Economy and Competitiveness project TIN2012-35669

Submitted to:
PROLE 2014

Concurrent constraint programming (ccp) [28, 26] has been proposed as a declarative concurrent computational model. Within this standard, relationships between variables are stated in the form of constraints. Constraints differ from the primitives of the common imperative programming languages [4]. They do not specify the steps to execute, but rather the properties of the solution. Languages following ccp are simple to define and, thus, bring to the foreground issues as time problems. It has also been extended in order to introduce synchronism and time passing. These new features have branched off into two different approaches. On the one hand, tccp [5], which essentially allows for weak-preemption and non-determinism, is highly suitable for modelling large concurrent timed systems. On the other hand, *tcc* [27] and *Timed Default cc* [25] allow for strong-preemption and determinism which make them better for real-time systems, such as kernels.

All the aforementioned paradigms focus on the discrete behaviour of systems, abstracting from the continuous components present in most real systems. However, taking into account the continuous behaviour allows for more faithful modelling and the analysis of interesting properties of systems. Following this idea, *hybrid cc* [11] has been defined as an extension of *Default cc* to support the modelling of the continuous component of systems. Language *hybrid cc* is *synchronous and deterministic* and includes continuous variables that evolve following a linear dynamics.

In this paper, we present HY-tccp, a hybrid extension of tccp. HY-tccp is a non-deterministic and synchronous language that incorporates continuous variables that follow a dynamics determined by a differential equation. In consequence, the language inherits and enriches the declarative nature of tccp, thus allowing the modelling of systems that display a discrete/continuous behaviour. The paper shows that the extension of a declarative paradigm with continuous dynamics is not only possible, but it also leads to a powerful language with which it is possible to describe and formally analyze complex systems. Initially, we have only considered the modelling of multi-rated [7] hybrid systems, *i.e.*, systems whose continuous variables follow a constant dymanics. However, our proposal is to use HY-tccp to describe more complex dynamics such as those defined by rectangular sets.

The paper is organized as follows. In Section 2, we briefly introduce the essential aspects of tccp. In addition, we also introduce hybrid automata, which constitute the commonly used formalism for describing hybrid systems [22]. In Section 3, we describe the new characteristics that have been added to tccp in order to describe hybrid systems, and formally define the semantics of the new language. Section 4 contains two examples to highlight the expressive power of HY-tccp. Finally, Section 5 concludes the paper and outlines the future work.

2 Background

This section describes the preliminaries of the paper. In Subsection 2.1, we present tccp, the starting point language of HY-tccp. Secondly, in Subsection 2.2, we introduce hybrid automata that constitute the key formalism for describing hybrid systems.

2.1 The tccp language

The *Timed Concurrent Constraint Programming* language (tccp for short) was defined as an extension of the *Concurrent Constraint Programming* language ccp [24]. In the ccp paradigm, the notion of *store as valuation* is replaced by the notion of *store as constraint*. The computational model is based on a global store where constraints are accumulated, and on a set of agents that interact with the store. The model is parametric w.r.t. a cylindric constraint system $\mathscr{C}$ that is defined as follows.

Definition 2.1. Let $\langle \mathscr{C}, \leq, \sqcup, true, false \rangle$ be a complete algebraic lattice where $\sqcup$ is the lub operation, and *true*, *false* are the least and the greatest elements of $\mathscr{C}$, respectively. Assume that *Var* is a denumerable set of variables, and for each $x \in Var$, there exists a function $\exists_x : \mathscr{C} \to \mathscr{C}$ such that, for each $u, v \in \mathscr{C}$:

1. $u \vdash \exists_x u$
2. $u \vdash v$ then $\exists_x u \vdash \exists_x v$
3. $\exists_x(u \sqcup \exists_x v) = \exists_x u \sqcup \exists_x v$
4. $\exists_x(\exists_y u) = \exists_y(\exists_x u)$

Then $\langle \mathscr{C}, \leq, \sqcup, true, false, Var, \exists \rangle$ is a *cylindric constraint system*.

We will use the entailment relation $\vdash$ instead of its inverse relation $\leq$. Formally, given $u, v \in \mathscr{C}, u \leq v \iff v \vdash u$.

A *set of diagonal elements* for a cylindric constraint system is a family $\{\delta_{xy} \in \mathscr{C} | x, y \in Var\}$ such that

1. $true \vdash \delta_{xx}$
2. If $y \neq x, z$ then $\delta_{xz} = \exists_y(\delta_{xy} \sqcup \delta_{yz})$.
3. If $x \neq y$ then $\delta_{xy} \sqcup \exists_x(v \sqcup \delta_{xy}) \vdash v$.

Diagonal elements allow us to hide variables that represent local variables, as well as to implement parameter passing between predicates. Thus, quantifier $\exists_x$ and diagonal elements δ_{xy} allow us to properly deal with variables in constraint systems.

In tccp, a new conditional agent (now c then A else B) is introduced (w.r.t. ccp) which makes it possible to model behaviours where the absence of information can cause the execution of a specific action. Intuitively, the execution of a tccp program evolves by asking and telling information to the store. Let us briefly recall the syntax of the language:

$$A ::= \mathsf{stop} \mid \mathsf{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A \mid A||A \mid \exists x A \mid \mathsf{p}(x)$$

where c, c_i are *finite constraints* (i.e., atomic propositions) of $\mathscr{C}$. A tccp *process* P is an object of the form $D.A$, where D is a set of procedure declarations of the form $\mathsf{p}(x) :: -A$, and A is an agent.

Intuitively, the stop agent finishes the execution of the program, $\mathsf{tell}(c)$ adds the constraint c to the store, whereas the choice agent ($\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$) consults the store and non-deterministically executes the agent A_i in the following time instant, provided the store satisfies the condition c_i; otherwise the agent suspends. The conditional agent (now c then A else B) can detect *negative information* in the sense that, if the store satisfies c, then the agent A is executed; otherwise (even if $\neg c$ does not hold), B is executed. $A1||A2$ executes the two agents $A1$ and $A2$ in parallel. The $\exists x\, A$ agent is used to hide the information regarding x, i.e., it makes x local to the agent A. Finally, $\mathsf{p}(x)$ is the procedure call agent.

The semantics of tccp is given by transition relation $\longrightarrow$ defined in rules **R1-R10** in Figure 1. In the semantics, it may be observed that: (1) store increases monotonically as an effect of agent tell (rule **R1**); (2) the only agents which consume a time unit are tell, choice and call agents (rules **R1**, **R2** and **R10**); and (3) agents transit in a completely synchronized manner (rule **R7**).

2.2 Introduction to hybrid systems

Real-life critical systems usually have complex behaviours which cannot be completely captured by discrete modelling and formal techniques verification. Many systems evolve following a continuous dynamics which may instantaneously change due to some external or internal events. For example, a cooling system that has two discrete states (*on* or *off*), and a continuous component which is the temperature of the room, that makes it evolve from one state to the other.

R1	$\langle \mathsf{tell}(c), st\rangle_t \longrightarrow \langle \mathsf{stop}, st \sqcup_{t+1} c\rangle_{t+1}$	
R2	$\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, st\rangle_t \longrightarrow \langle A_j, st\rangle_{t+1}$	if $0 \leq j \leq n$ and $st \vdash_t c_j$
R3	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}$	if $st \vdash_t c$
R4	$\dfrac{\langle B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}$	if $st \nvdash_t c$
R5	$\dfrac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}}$	if $st \vdash_t c$
R6	$\dfrac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B, st\rangle_{t+1}}$	if $st \nvdash_t c$
R7	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1} \text{ and } \langle B, st\rangle_t \longrightarrow \langle B', st''\rangle_{t+1}}{\langle A \Vert B, st\rangle_t \longrightarrow \langle A' \Vert B', st' \sqcup st''\rangle_{t+1}}$	
R8	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1} \text{ and } \langle B, st\rangle_t \not\longrightarrow}{\langle A \Vert B, st\rangle_t \longrightarrow \langle A' \Vert B, st'\rangle_{t+1}}$	
R9	$\dfrac{\langle A, st_1 \sqcup \exists x\, st_2\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \exists^{st_1} x A, st_2\rangle_t \longrightarrow \langle \exists^{st'} x A', st_2 \sqcup \exists x\, st'\rangle_{t+1}}$	
R10	$\langle p(x), st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}$	if $p(x) :- A \in D$

Figure 1: Operational semantics of tccp

These types of systems are referred to as *hybrid systems*, and are characterized by having a combination of discrete and continuous behaviours. Modelling a hybrid system requires using both discrete and continuous variables. The evolution of discrete variables is assumed to be instantaneous, while continuous variables evolve over time following differential equations. Describing the behaviour of continuous variables requires enriching finite automata with time related aspects. The new formalism is referred to as *hybrid automata* [15].

Definition 2.2 (Hybrid automaton)**.** A *hybrid automaton H* is a tuple $\langle Loc, T, \Sigma, X, Init, Inv, Flow, Jump\rangle$ where:

- Loc is a finite set $\{loc_1, \cdots, loc_n\}$ of discrete states (locations).
- $T \subseteq Loc \times Loc$ is a finite set of transitions.

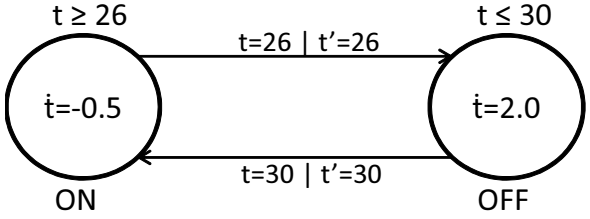


Figure 2: Hybrid automaton for the cooling system

- Σ is the set of event names, with a labelling function $lab : T \to \Sigma$.
- $X = \{x_1, \cdots, x_m\}$ is a finite set of real-valued variables. Associated to X, we consider two new sets. Set $\dot{X} = \{\dot{x_1}, \cdots, \dot{x_m}\}$ represents the first derivatives of the elements of X. In addition, set $X' = \{x'_1, \cdots, x'_m\}$ represents updates of variables when a discrete transition takes place. In this section, we assume that discrete variables are continuous variables whose derivative is zero in all locations. Thus, X contains all continuous and discrete variables, but the update of discrete variables only occurs during transitions.
- *Init*, *Inv* and $Flow$ are functions that assign predicates to each discrete state $loc \in Loc$. Thus, $Init(loc)$ defines the initial values for the continuous variables. $Inv(loc)$ gives the conditions that must be true while the system stays at location loc. $Flow(loc)$ contains the differential equations that define the evolution of the continuous variables in X.
- Function *Jump* assigns each discrete transition $t \in T$ to the guards that assure that the discrete step can be carried out, and to the updates of variables before making the transition to the new location.

*Example 1: Figure 2 shows a hybrid automaton for the cooling example. The automaton has two locations (***ON, OFF***), and a continuous variable (***t***) storing the room temperature. When the automaton is at location* **ON** *(the cooler is on), temperature decreases at rate* $-\mathbf{0.5}$*. However, when the location is* **OFF** *(the cooler is off), temperature increases at rate* **2***. Transitions between locations represent the on/off switch of the cooler. Transitions are guarded with conditions that model when it is possible for them to be fired (for instance, transition* **ON-OFF** *may be carried out when temperature is* **26***).*

A hybrid automaton behaves like a *timed transition system* (TTS), where each step is labelled with a real value that represents its duration, or by d that represents that a discrete transition has been executed. Let $[X \to \mathbb{R}]$ be the set of maps from X to $\mathbb{R}$. Automaton states, which we call hybrid states from now on, are pairs of the form (loc, v), where $loc \in Loc$ is a location of the automaton, and $v \in [X \to \mathbb{R}]$ denotes the current values of continuous variables.

In order to define the trajectories determined by a hybrid automaton, we need some definitions. If p is a predicate over X, then $[[p]]$ denotes all functions $v \in [X \to \mathbb{R}]$ that satisfy p. Similarly, we may define the meaning of a predicate over X', over $X \cup X'$ or over $X \cup \dot{X}$. Finally, the same notion may be extended to sets of predicates such as $Init, Inv$ or $Jump$.

Definition 2.3 (Trajectories)**.** Given $H = \langle Loc, T, \Sigma, X, Init, Inv, Flow, Jump \rangle$ a hybrid automaton, two types of transitions may be considered:

- Discrete transitions: given $t = (loc, loc') \in T, (loc, v) \to_d (loc', v')$, iff $v, v' \in [X \to \mathbb{R}]$, and $(v, v') \in [[Jump(t)]]$.
- Continuous transitions: for each $\delta \in \mathbb{R}^{>0}$, we have $(loc, v) \to_\delta (loc, v')$ iff there exists a differentiable function $f : [0, \delta] \to \mathbb{R}^m$, $\dot{f} : [0, \delta] \to \mathbb{R}^m$ being its first derivative, such that:
  - $f(0) = v, f(\delta) = v'$.
  - $\forall r \in (0, \delta), f(r) \in [[Inv(loc)]]$ and $(f(r), \dot{f}(r)) \in [[Flow(loc)]]$

Thus, a **trajectory** is a (possible infinite) sequence of *hybrid states* such as $(loc_0, v_0) \to_{\lambda_0} \cdots$, where $v_0 \in [[Inv(loc_0)]]$, and $\lambda_i \in \mathbb{R} \cup \{d\}$.

In a given trajectory, each new state $(loc, v) \in Loc \times [X \to \mathbb{R}]$ is obtained by executing a discrete or a continuous transition. There is no priority on transitions. Therefore, if it is possible to evolve through one (or more than one) discrete transition, and through a continuous transition, the system is free to select

any transition.

Example 2: Considering the cooler system in Figure 2, the following trajectory represents a possible evolution of the automaton starting at hybrid state $(\mathbf{ON}, 27)$:

$$(\mathbf{ON},27) \rightarrow_1 (\mathbf{ON},26.5) \rightarrow_1 (\mathbf{ON},26) \rightarrow_d (\mathbf{OFF},26) \rightarrow_{0.5} (\mathbf{OFF},27) \rightarrow_{1.5} (\mathbf{OFF},30) \rightarrow_d \cdots$$

3 Extension of tccp

In this section, we present an extension of tccp that is able to represent hybrid automata. The new language, HY-tccp, subsumes tccp, and includes new agents to model the continuous behaviour.

The intuitive idea of the proposal may be summarized as follows. Firstly, HY-tccp allows the use of discrete constraints as in tccp, but also admits the definition of continuous variables which induce new constraints. The store of tccp is extended with a new component to record the values of continuous variables. We denote the new extended store as $st + st^c$. Store st behaves as in tccp, that is, it contains discrete constraints which are added monotonically. In contrast, st^c is not monotonic: it saves the value (and the flow) of continuous variables which may change over time. All transitions defined by the standard tccp semantics turn into *discrete* transitions in HY-tccp, whose execution is instantaneous, i.e., do not consume time. The continuous passing of time is included in the language by means of new branches askC for the choice agent which may be non-deterministically selected.

We now describe the extension of tccp in detail.

3.1 Elements of HY-tccp

We describe each new element in depth. Firstly, we formalize the adding of the continuous store to the discrete one. Secondly, we present the new operators that allow continuous behaviour in our language. Finally, we show and explain the new rules, which in combination with the existing ones from tccp define HY-tccp.

3.1.1 Extended Store

Let Var^c be the set of continuous variables. In each instant, continuous store st^c contains the current value of each continuous variable together with its flow, that is, $st^c \subseteq Var^c \times \mathbb{R} \times \mathbb{R}$. Given a continuous store st^c, $(x,v,f) \in st^c$ means that the value and flow of x in st^c are v and f, respectively. For consistency reasons, each variable $x \in Var^c$ may appear at most once in a 3-tuple (x,v,f) of continuous stores, otherwise store is $false$. We write $st^c - x$ the continuous store resulting of removing the 3-tuple associated to x in st^c, if it exists, that is, $st^c - x = \{(y,v,f) | (y,v,f) \in st^c, y \neq x\}$. In addition, we write $st^c \sqcup (x,v,f)$ to denote the store obtained by adding 3-tuple (x,v,f) to st^c, that is, $st^c \sqcup (x,v,f) = st^c \cup \{(x,v,f)\}$. Observe that if st^c contains a 3-tuple for x, $st^c \sqcup (x,v,f)$ is inconsistent, i. e., $st^c \sqcup (x,v,f) = false$. Similarly, $st^c \sqcup st^{c\prime}$ denote the continuous store resulting of adding all 3-tuples of $st^{c\prime}$ to st^c.

We define the *time projection* of a continuous store st^c on time instant $t > 0$ as $st^c_t = \{(x, v+t.f, f) | (x,v,f) \in st^c\}$.

Given constraint a, we write $st^c \vdash a$ when store $\{x = v | (x,v,-) \in st^c\} \vdash a$.

3.1.2 Agents change and extended choice

We introduce two new agents to manage continuous variables. Agent change is used to model the updating of values and/or flow of continuous variables. These changes are instantaneous and occur when hybrid automata carry out discrete transitions. Agent askC models the continuous transitions. It permits the hybrid system to stay at a given location while the location invariant is preserved. Continuous variables evolve following their flow, while systems are at locations. Agent choice of tccp choice has been extended to allow the non-deterministic selection between discrete and continuous transitions, as occurs in hybrid automata. The syntaxis of the new agent is $\sum_{i=1}^{n}(\sum_{j=1}^{m_i} \mathsf{ask}(c_i^j) \rightarrow A_i^j + \mathsf{askC}(inv_i))$. This agent typically represents an automaton with n locations. For each $1 \leq i \leq n$, choice $\sum_{j=1}^{m_i} \mathsf{ask}(c_i^j) \rightarrow A_i^j + \mathsf{askC}(inv_i)$ represents the behaviour of automata at the i-th location. Condition inv_i represents the location invariant. Automata may stay at this location while store satisfies this invariant. Branches $\sum_{j=1}^{m_i} \mathsf{ask}(c_i^j) \rightarrow A_i^j$ represent the discrete transitions that automata may execute from the i-th location. As hybrid automata are at an exact location in each instant, we assume that the new choice agents satisfy the following:

1. for each $1 \leq i \neq i' \leq n$, $inv_i \wedge inv_{i'} = false$, that is, two invariant conditions may not be simultaneously true.
2. for each pair of constraints $c_i^a, c_{i'}^b$ with $i \neq i'$, $c_i^a \wedge c_{i'}^b = false$, that is, constraints of branches of different locations are inconsistent.

3.1.3 Semantics of HY-tccp

The semantic rules shown in Figure 3 provide the behaviour of the new language elements together with their integration with the standard tccp agents. The rules define two transition relations: discrete transition $\longrightarrow_d$ for the discrete transitions of the system, and $\longrightarrow_t$ with $t \in \mathbb{R}$ for continuous transitions of duration t. In some cases, to encompass both the discrete and continuous transitions under the same rule, we make use of $\lambda \in \mathbb{R} \cup \{d\}$. In the figure, rules **R1-R10** correspond to the standard semantics of tccp (with some adaptations), while rules **Ri', Ri"** give meaning to the new agents, and to the composition of continuous and discrete transitions.

R1 shows the behaviour of the tell agent. It stores the new discrete constraint in st. **R1'** introduces the new agent change which adds to st^c the new continuous variable (if it has not been added before) as well as its new (or first) flow and initial value.

Rules **R2** and **R2'** define the non-deterministic choice, composed of ask and askC agents. Observe that to simplify the rules, these agents have been reordered with respect to the description given in the previous section. Rule **R2** represents the discrete transition of choice to agent A_i, providing that $st + st^c$ (*i.e.* the whole store) entails the guarded condition c_i. Rule **R2'** shows the new allowed behaviour, that lets the system evolve continuously while it accomplishes one of the invariants. Subindex t is the duration of the transition.

Rules **R3-R6** redefine the behaviour of agent now c then A else B. This agent evaluates guard c and transits choosing branch A or B according to whether c is true. The agent manages negative information in the sense that branch B is selected when the store cannot deduce c, which does not necessarily mean that the store satisfies $\neg c$. Observe that the evaluation of c takes one time unit, only if the branch selected cannot evolve, otherwise it is instantaneous. In addition, in the case the branch (A or B) selected can evolve, it can make both a discrete or a continuous transition.

R1 $\langle \mathsf{tell}(c), st\ +st^c\rangle \longrightarrow_d \langle \mathsf{stop}, st \sqcup c + st^c\rangle$

R1' $\langle \mathsf{change}(x,v,f), st\ +st^c\rangle \longrightarrow_d \langle \mathsf{stop}, st + (st^c - x) \sqcup (x,v,f)\rangle$

R2 $\langle \sum_{i=1}^{n}(\mathsf{ask}(c_i) \to A_i) + \sum_{j=1}^{m}\mathsf{askC}(inv_j), st\ +st^c\rangle \longrightarrow_d \langle A_i, st\ +st^c\rangle$ if $\exists 1 \leq i \leq n$, and $st\ +st^c \vdash c_i$

R2' $\langle \sum_{i=1}^{n}(\mathsf{ask}(c_i) \to A_i) + \sum_{j=1}^{m}\mathsf{askC}(inv_j), st\ +st^c\rangle \longrightarrow_t \langle \sum_{i=1}^{n}(\mathsf{ask}(c_i) \to A_i) + \sum_{j=1}^{m}\mathsf{askC}(inv_j), st\ +st_t^c\rangle$ if $\exists 1 \leq j \leq m$, and $\forall 0 < t' \leq t$, $st\ +st_t^c \vdash inv_j$

R3 $\dfrac{\langle A, st+st^c\rangle \longrightarrow_\lambda \langle A', st'+st^{c\prime}\rangle, \lambda \in \mathbb{R}\cup\{d\}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st+st^c\rangle \longrightarrow_\lambda \langle A', st'+st^{c\prime}\rangle}$ if $st+st^c \vdash c$

R4 $\dfrac{\langle B, st+st^c\rangle \longrightarrow_\lambda \langle B', st'+st^{c\prime}\rangle, \lambda \in \mathbb{R}\cup\{d\}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st+st^c\rangle \longrightarrow_\lambda \langle B', st'+st^{c\prime}\rangle}$ if $st+st^c \nvdash c$

R5 $\dfrac{\langle A, st+st^c\rangle \not\longrightarrow_\lambda, \lambda \in \mathbb{R}\cup\{d\}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st+st^c\rangle \longrightarrow_d \langle A, st+st^c\rangle}$ if $st+st^c \vdash c$

R6 $\dfrac{\langle B, st+st^c\rangle \not\longrightarrow_\lambda, \lambda \in \mathbb{R}\cup\{d\}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st+st^c\rangle \longrightarrow_d \langle B, st+st^c\rangle}$ if $st+st^c \nvdash c$

R7 $\dfrac{\langle A, st+st^c\rangle \longrightarrow_d \langle A', st'+st^{c\prime}\rangle \quad \langle B, st+st^c\rangle \longrightarrow_d \langle B', st''+st^{c\prime\prime}\rangle}{\langle A||B, st+st^c\rangle \longrightarrow_d \langle A'||B', st'\sqcup st''+st^{c\prime}\sqcup st^{c\prime\prime}\rangle}$

R7' $\dfrac{\langle A, st+st^c\rangle \longrightarrow_t \langle A, st+st^{c\prime}\rangle \quad \langle B, st+st^c\rangle \longrightarrow_t \langle B, st+st^{c\prime\prime}\rangle}{\langle A||B, st+st^c\rangle \longrightarrow_t \langle A||B, st+st^{c\prime}\sqcup st^{c\prime\prime}\rangle}$

R8 $\dfrac{\langle A, st+st^c\rangle \longrightarrow_d \langle A', st'+st^{c\prime}\rangle, \langle B, st+st^c\rangle \not\longrightarrow_\lambda}{\langle A||B, st+st^c\rangle \longrightarrow_d \langle A'||B, st'+st^{c\prime}\rangle}$

R8' $\dfrac{\langle A, st+st^c\rangle \longrightarrow_d \langle A', st'+st^{c\prime}\rangle \quad \langle B, st+st^c\rangle \longrightarrow_t \langle B'', st''+st^{c\prime\prime}\rangle}{\langle A||B, st+st^c\rangle \longrightarrow_d \langle A'||B, st'+st^{c\prime}\rangle}$

R8'' $\dfrac{\langle A, st+st^c\rangle \longrightarrow_t \langle A, st+st^{c\prime}\rangle \quad \langle B, st+st^c\rangle \not\longrightarrow_\lambda, \lambda \in \mathbb{R}\cup\{d\}}{\langle A||B, st+st^c\rangle \longrightarrow_t \langle A||B, st+st^{c\prime}\rangle}$

R9 $\dfrac{\langle A, st_1 \sqcup \exists x\, st_2 + st_1^c \sqcup \exists x\, st_2^c\rangle \longrightarrow_d \langle A', st'+st^{c\prime}\rangle}{\langle \exists^{st_1+st_1^c} x A, st_2+st_2^c\rangle \longrightarrow_d \langle \exists^{st'+st^{c\prime}} x A', st_2 \sqcup \exists x\, st' + st_2^c \sqcup \exists x\, st^{c\prime}\rangle}$

R10 $\langle p(x), st+st^c\rangle \longrightarrow_d \langle A, st+st^c\rangle$ if $p(x) : -A \in D$

Figure 3: Operational semantics of HY-tccp

Rules **R7-R7'** model the synchronized execution of agents by means of discrete or continuous transitions. Observe that, in rule **R7'**, the duration of the continuous transitions of parallel agents must coincide. **R8-R8'-R8"** define the evolution of the system when the agents in execution may transit following different continuous/discrete transitions. **R8** represents the case when an agent makes a discrete transition while the other agents are blocked. **R8"** represents this same situation but with the difference that the transition is continuous. Rule **R8'** is a bit more complicated. It models the discrete and instantaneous evolution of an agent when another agent wants to transit continuously. In this case, the discrete transition is executed before the continuous one. Observe that these rules are non-deterministic, and thus, if an agent may make a discrete and a continuous transition, one of them is non-deterministically selected.

Locality is defined in rule **R9**. Agent $\exists x\mathsf{A}$ states that variable x is local to A. This means that depending on whether the information on x is provided by A or by the external environment, that information is hidden. Finally, rule **R10** treats the case of a procedure call when the actual parameter equals the formal parameter. It remains the same after the continuous extension has been added.

4 Examples

In order to show the expressivity of the HY-tccp language, we have modelled two examples. Both have discrete and continous components that define hybrid environments. We present the problems, build a hybrid automaton related to each one and model them on HY-tccp as well.

4.1 Cooling system

We model a simple room cooling system. It consists of a cooler that has a sensor to control its state according to the temperature of the room. When the sensor detects that the temperature is in the upper threshold, it switches the cooler on. From then on the temperature decreases at a given rate. When it reaches the lower threshold, which is the minimum allowed temperature, the cooler turns off again. Therefore, the temperature rises at another given rate.
There are two different discrete states. They are defined by the signals *on* and *off*, which indicate if the cooler is active or turned off. There is also a continuous component, the temperature, defined as t. Since t is continuous, it has an associated differential equation that defines its flow over time. Once the system has been described, we can transform it into an hybrid automaton in order to express both discrete and continuous behaviour. Figure 2 shows the definition of our model as a hybrid automaton. The circles are the locations. The arrows are the edges. Over each location there is a predicate that represents the invariant. Inside each location there is a description of the continuous evolution of the variables when the system is in there. The edges are labelled with the information of the transition. These labels contain the update of the variables and the guard predicate of the transition. This system has two locations and two edges. Each location corresponds to a discrete state.

This problem can be modelled in HY-tccp as shown in Figure 4. There is only one process, called **cooler**. It has on its body a list of ask and askC statements. They define the behaviour of the system according to the temperature and the state of the cooler. Note how streams are used to simulate the evolution of the system from *on* to *off* and viceversa. The askC operator lets the system evolve over time.

Observe that each branch that appears in the HY-tccp code corresponds to a different automaton

```
cooler(State,t) :- ∃ State', State"(
        askC(State=[off|-] ∧ t≤30)
          +
        ask(State=[off|State'] ∧ t=30) →
                change(t, 30, -0.5) ||
                tell(State'=[on|State"]) ||
                cooler(State',t)
          +
        ask(State =[off|-] ∧ t>30) → stop
          +
        askC(State=[on|-] ∧ t≥26)
          +
        ask(State=[on|State'] ∧ t=26) →
                change(t, 26, 2.0) ||
                tell(State'=[off|State"]) ||
                cooler(State',t)
          +
        ask(State =[on|-] ∧ t<26) → stop)

init:- ∃ State,t(cooler(State,t) || tell(State=[off|-]) || change(t, 29, 2.0))
```

Figure 4: HY-tccp model for a cooling system

state. For each location, *on* and *off*, there is a branch that represents the continuous transitions (when the continuous variables evolve), another branch that models the discrete transition from the location, and finally, a branch which represents the deadlock case, when the values of the continuous variables do not allow the system either to stay at the location or to transit.

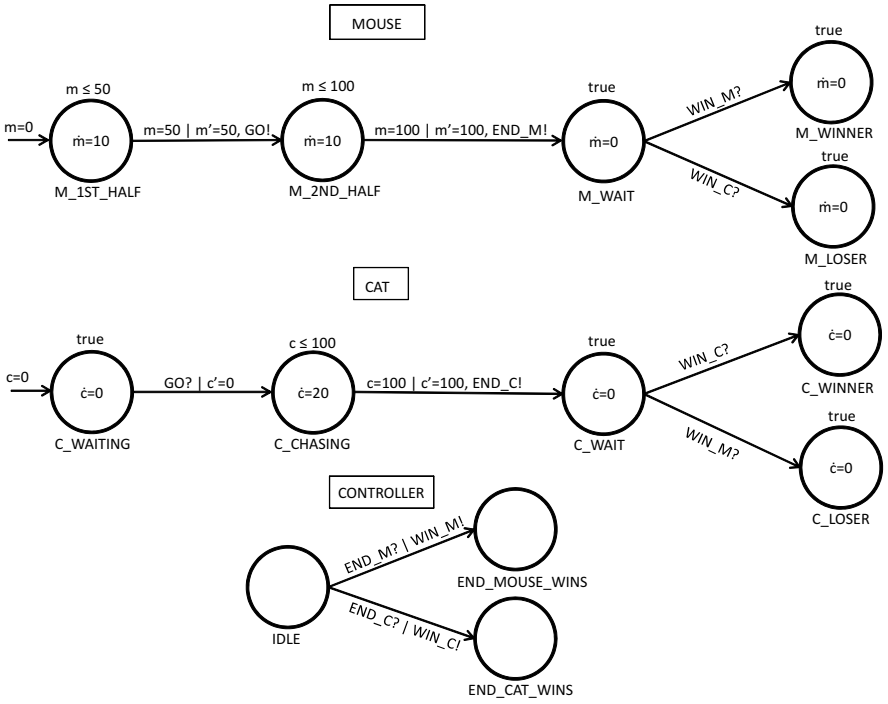


Figure 5: Hybrid automata for the cat and mouse problem

```
mouse() :- ∃ m(
      change(m, 0, 10) ||
      askC(m ≤ 50)
        +
      ask(m=50) → tell(GO) ||
                  askC(m≤ 100)
                    +
                  ask(m=100) → tell(END_M) ||
                               ask(WIN_M)→ claimPrize(...)
                                 +
                               ask(WIN_C)→ stop)

cat() :- ∃ c(
      ask(GO) → change(c, 0, 20) ||
                  askC(c≤ 100)
                    +
                  ask(c=100)→ tell(END_C) ||
                               ask(WIN_C)→ claimPrize(...)
                                 +
                               ask(WIN_M)→ stop)

controller() :-
      ask(END_M) → tell(WIN_M)
        +
      ask(END_C) → tell(WIN_C)

init:- (mouse() || cat() || controller())
```

Figure 6: HY-tccp model for the cat and mouse problem

4.2 Cat and mouse

In the next example we consider the cat and mouse problem (extracted from [10]). A mouse starts at the point of origin, running at a speed of 10 metres/second towards a hole that is 100 metres away. After it has run 50 metres, the cat starts to chase it, at a speed of 20 metres/second (starting from the point of origin as well). The positions of the cat and the mouse are modelled by two continuous variables, called *c* and *m* respectively. The cat wins if it catches the mouse before it reaches the hole, otherwise it loses. There is a controller which determines in a non-deterministic way who the winner is.
As before, we include a hybrid automaton perspective to clarify the behaviour of the system. Figure 5 shows the definition of our model as a hybrid automaton. Now there are three automata, one per process. The first is the mouse, which has three non-return locations, and later has two different termination states, depending on whether it is the winner or not. The cat is similar, but it has to wait in the first location to start its run until the mouse sends its signal. The last one, which has only discrete behaviour, has to decide the winner whenever it receives the end signals of both processes.

This problem can also be modelled in HY-tccp as shown in Figure 6. There are three processes: **mouse**, **cat** and **controller**. At first the cat is waiting for the mouse to inform it that it is at the halfway point (50 meters run). Then the cat starts its hunt. At the end, they send a message to the controller, which decides the winner. In Appendix A we show a possible trace of execution on this program.

5 Conclusions

In this paper we have proposed a language to intuitively model hybrid systems. We have done this by extending a concurrent constraint programming language called tccp. Through the examples given, we have shown the expressiveness of the new language with regard to hybrid automata. In most cases the translation from the HY-tccp code is intuitive. Nevertheless, it has not been defined any translation rule yet. Our language, given that it is based on tccp, provides a formalism inspired by timed process algebra instead of synchronous languages. This approach opts for weak-preemption and non-determinism instead, covering the requirements for large concurrent timed systems, and therefore hybrid systems as well, which is our main goal.

In constrast to ccp, tccp directly introduces a timed interpretation of the operators tell and ask. In our case, after the extension of tccp, time passes only when we are in any askC branch of a non-deterministic choice. Here there are two concepts of time: as a mechanism to arrange synchronization and time itself passing. By extending tccp and by means of the enhacement of the time frame (adding continuous time), we get closer to the behaviour of the hybrid automata. They are non-deterministic and have both a discrete and a continuous component, as our models do. Since our language has a tccp basis, in contrast with *hybrid cc*, it has the advantage that transfers of positive information after each step are not needed, thus implicitly keeping the monotonic approach on *st*. Regarding negative information, we have the problem of distinguishing whether a constraint is not true either by being false or because it is absent. It has to be kept in mind during the modelling of the system. To solve this is one of our future goals.

When defining HY-tccp we have made restrictions on the guards of ask and askC agents to be consistent with the definition of hybrid automata. Basically, the restrictions impede that a hybrid automaton modelled in HY-tccp is simultaneously at two different locations. However, we have to study whether relaxing these limitations would allow us to describe more general hybrid systems.

For future work we plan to model more complex cases as, for example, water resources management. To do this, we are working on the development of a framework to perform the modelling and simulation of hybrid systems written in HY-tccp. We are also interested in the definition of a translation rules system from HY-tccp to hybrid automaton and vice versa. Once we have finished these steps, we expect to make use of temporal logic to verify properties on these models as in [1, 9]. Another feature we would like to explore is the adjustment of the language to make it compatible with rectangular automata [18], which are more complex than the ones presented here.

References

[1] M. Alpuente, M. M. Gallardo, E. Pimentel & A. Villanueva (2006): *Verifying Real-Time Properties of tccp Programs*. Journal of Universal Computer Science 12.

[2] Rajeev Alur & Thomas A. Henzinger (1996): *Reactive Modules*. In: Formal Methods in System Design, IEEE Computer Society Press, pp. 207–218.

[3] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier & Place Sophie Laltte (1992): *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*.

[4] Programs James Bieman, James M. Bieman, David A. Gustafson, Albert L. Baker, Austin C. Melton & Paul N. Clites (1988): *A Standard Representation of Imperative Language*. The Journal of Systems and Software 8, pp. 13–37.

[5] M.C. Meo F.S. de Boer, M. Gabbrielli (2000): *A Time Concurrent Constraint Language*. Information and Computation 161, pp. 45–83.

[6] Patrice Bremond-Gregoire & Insup Lee (1997): *A Process Algebra of Communicating Shared Resources with Dense Time and Priorities*. Technical Report, Theoretical Computer Science.

[7] C. Daws & S. Yovine (1995): *Two Examples of Verification of Multirate Timed Automata with Kronos*. In: *Proceedings of the 16th IEEE Real-Time Systems Symposium*, RTSS '95, IEEE Computer Society, Washington, DC, USA, pp. 66–75.

[8] Harald Fecher (2001): *A Real-time Process Algebra with Open Intervals and Maximal Progress*.

[9] M.-M. Gallardo & L. Panizo (2013): *Extending Model Checkers for Hybrid System Verification: the case study of SPIN*. *Software Testing, Verification and Reliability*.

[10] V. Gupta, R. Jagadeesan & V. A. Saraswat (1998): *Computing with Continuous Change*.

[11] Vineet Gupta, Radha Jagadeesan, Vijay Saraswat & Daniel G Bobrow (1995): *Programming in Hybrid Constraint Languages*. In: *Hybrid Systems II, volume 999 of LNCS*, Springer Verlag, pp. 226–251.

[12] N. Halbwachs, P. Caspi, P. Raymond & D. Pilaud (1991): *The synchronous dataflow programming language LUSTRE*. In: *Proceedings of the IEEE*, pp. 1305–1320.

[13] Nicolas Halbwachs (1993): *Synchronous Programming of Reactive Systems*.

[14] David Harel (1987): *Statecharts: A Visual Formalism for Complex Systems*. *Sci. Comput. Program.* 8(3), pp. 231–274.

[15] T. A. Henzinger (1996): *The theory of hybrid automata*. In: *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, IEEE Computer Society, Washington, DC, USA, pp. 278–292.

[16] C. A. R. Hoare (1978): *Communicating Sequential Processes*.

[17] David R. Jefferson (1985): *Virtual time*. *ACM Transactions on Programming Languages and Systems* 7, pp. 404–425.

[18] Peter W. Kopke (1996): *The Theory of Rectangular Hybrid Automata*. Technical Report, Ithaca, NY, USA.

[19] Paul Le Guernic, Thierry Gautier, Michel Le Borgne & Claude Le Maire (1991): *Programming Real-Time Applications with Signal*. *Proceedings of the IEEE* 79(9), pp. 1321–1336.

[20] R. Milner (1982): *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[21] Faron Moller & Chris Tofts (1990): *A temporal calculus of communicating systems*. In J.C.M. Baeten & J.W. Klop, editors: *CONCUR '90 Theories of Concurrency: Unification and Extension*, *Lecture Notes in Computer Science* 458, Springer Berlin Heidelberg, pp. 401–415.

[22] Jean-Franois Raskin (2005): *An Introduction to Hybrid Automata*. In Dimitrios Hristu-Varsakelis & WilliamS. Levine, editors: *Handbook of Networked and Embedded Control Systems*, Control Engineering, Birkhuser Boston, pp. 491–517.

[23] G. M. Reed & A. W. Roscoe (1988): *A Timed Model for Communicating Sequential Processes*. In: *Theoretical Computer Science*, pp. 314–323.

[24] V. A. Saraswat (1993): *Concurrent Constraint Programming Languages*. *The MIT press* MCMXCVII.

[25] Vijay Saraswat, Radha Jagadeesan & Vineet Gupta (1996): *Timed Default Concurrent Constraint Programming*. *Journal of Symbolic Computation* 22.

[26] Vijay A. Saraswat & et al. (1990): *Semantic foundations of concurrent constraint programming*.

[27] Vijay A. Saraswat, Radha Jagadeesan & Vineet Gupta (1994): *Foundations of Timed Concurrent Constraint Programming*. In: *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, IEEE Computer Press, pp. 71–80.

[28] Vijay Anand Saraswat (1989): *Concurrent Constraint Programming Languages*. Ph.D. thesis, Pittsburgh, PA, USA.

A Appendix

The aim of this appendix is to show how the language works. To do so, we have a step-by-step possible execution trace of the problem given in Subsection 4.2, *i.e.*, the cat and mouse problem. With the purpose of simplifying, we have not taken into account the first steps, mainly related to procedure calls and local declarations. Each row represents, respectively, the active agents in that moment, the state of the store (including the continuous store) and the rules that lead to the next step. Observe that when an active agent begins with a choice, *i.e.*, agent ask, branches have been omitted for clarity. More than one rule appearing in the last column means that there are intermediate states that we have not included.

Active agents	**Store**	**Rules**
$\mathsf{change}(m,0,10)\|\|\mathsf{askC}(m<=50)+\mathsf{ask}(m=50)$ $\|\|\mathsf{ask}(GO)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{\emptyset+\emptyset\}$	$\longrightarrow_d$ **R1'**
$\mathsf{askC}(m<=50)+\mathsf{ask}(m=50)$ $\|\|\mathsf{ask}(GO)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{\emptyset+(m,0,10)\}$	$\longrightarrow_2$ **R8",R2'**
$\mathsf{askC}(m<=50)+\mathsf{ask}(m=50)$ $\|\|\mathsf{ask}(GO)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{\emptyset+(m,20,10)\}$	$\longrightarrow_3$ **R8",R2'**
$\mathsf{askC}(m<=50)+\mathsf{ask}(m=50)$ $\|\|\mathsf{ask}(GO)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{\emptyset+(m,50,10)\}$	$\longrightarrow_d$ **R8,R2**
$\mathsf{tell}(GO)\|\|\mathsf{askC}(m<=100)+\mathsf{ask}(m=100)$ $\|\|\mathsf{ask}(GO)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{\emptyset+(m,50,10)\}$	$\longrightarrow_d$ **R8',R1**
$\mathsf{askC}(m<=100)+\mathsf{ask}(m=100)$ $\|\|\mathsf{ask}(GO)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{GO+(m,50,10)\}$	$\longrightarrow_d$ **R8',R2**
$\mathsf{askC}(m<=100)+\mathsf{ask}(m=100)$ $\|\|\mathsf{change}(c,0,20)\|\|\mathsf{askC}(c<=100)+\mathsf{ask}(c=100)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{GO+(m,50,10)\}$	$\longrightarrow_d$ **R8',R1'**
$\mathsf{askC}(m<=100)+\mathsf{ask}(m=100)$ $\|\|\mathsf{askC}(c<=100)+\mathsf{ask}(c=100)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{GO+(m,50,10),(c,0,20)\}$	$\longrightarrow_5$ **R7'**
$\mathsf{askC}(m<=100)+\mathsf{ask}(m=100)$ $\|\|\mathsf{askC}(c<=100)+\mathsf{ask}(c=100)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{GO+(m,100,10),(c,100,20)\}$	$\longrightarrow_d$ **R8**
$\mathsf{tell}(END\_M)\|\|\mathsf{ask}(WIN\_M)+\mathsf{ask}(WIN\_C)$ $\|\|\mathsf{tell}(END\_C)\|\|\mathsf{ask}(WIN\_C)+\mathsf{ask}(WIN\_M)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{GO+(m,100,10),(c,100,20)\}$	$\longrightarrow_d$ **R8,R1**
$\mathsf{ask}(WIN\_M)+\mathsf{ask}(WIN\_C)$ $\|\|\mathsf{ask}(WIN\_C)+\mathsf{ask}(WIN\_M)$ $\|\|\mathsf{ask}(END\_M)+\mathsf{ask}(END\_C)$	$\{\begin{smallmatrix}GO,END\_M,\\END\_C\end{smallmatrix}+(m,100,10),(c,100,20)\}$	$\longrightarrow_d$ **R2**
$\mathsf{ask}(WIN\_M)+\mathsf{ask}(WIN\_C)$ $\|\|\mathsf{ask}(WIN\_C)+\mathsf{ask}(WIN\_M)$ $\|\|\mathsf{tell}(WIN\_M)$	$\{\begin{smallmatrix}GO,END\_M,\\END\_C\end{smallmatrix}+(m,100,10),(c,100,20)\}$	$\longrightarrow_d$ **R1**
$\mathsf{ask}(WIN\_M)+\mathsf{ask}(WIN\_C)$ $\|\|\mathsf{ask}(WIN\_C)+\mathsf{ask}(WIN\_M)$ stop	$\{\begin{smallmatrix}GO,END\_M,\\END\_C,WIN\_M\end{smallmatrix}+(m,100,10),(c,100,20)\}$	$\longrightarrow_d$ **R2**

Continued on next page

Continued from previous page

Active agents	**Store**	**Rules**
Mouse wins!	$\{ \begin{matrix} WIN\_, GO, END\_M, \\ END\_C, WIN\_M \end{matrix} + (m,100,10),(c,100,20)\}$	

Enhancing Control over Jason Agents (Work in Progress)

Álvaro Fernández-Díaz*
Babel group, LSIIS, Facultad de Informática,
Universidad Politécnica de Madrid, Spain
avalor@babel.ls.fi.upm.es

Clara Benac-Earle
Babel group, LSIIS, Facultad de Informática,
Universidad Politécnica de Madrid, Spain
cbenac@babel.ls.fi.upm.es

Lars-Åke Fredlund
Babel group, LSIIS, Facultad de Informática,
Universidad Politécnica de Madrid, Spain
lfredlund@babel.ls.fi.upm.es

Agent-oriented programming languages like Jason (and its interpreter) facilitate programming multiagent systems because they provide the necessary infrastructure. However, several of these agent-oriented languages are not entirely self-contained. Programmers may have to write code in the low-level language used to implement the agent-oriented programming platform to implement some desired agent behaviour. For example, Jason programmers write Java code to modify the treatment of unexpected goals in the Jason reasoning cycle, or to fix the intention execution order.

In this paper, we discuss the problems such a dependence on underlying implementation languages presents to an agent language, and in the case of Jason we propose a solution based on extending the language with new syntactic constructs for more precise control of the execution flow of Jason agents. These new control mechanisms have been implemented in our Jason interpreter eJason.

1 Introduction

Agent-oriented programming languages facilitate programming multiagent systems because, in contrast to other programming languages, they are centered around the concept of intelligent agents and, therefore, provide the infrastructure necessary for programming such agents. One example is the Jason programming language [3]. In particular, Jason is based on the Belief-Desire-Intention (BDI) model [4]. The BDI model is based upon the model of human behaviour, specifically on the role that the intentions play in order to determine such behaviour. The consideration that certain software system adheres to the BDI model implies that some of the computational entities (namely the agents) composing such system possess a so-called *mental state*.

The utilisation of a BDI approach implies that the software agents will carry out two main activities: *deliberation* and *means-ends reasoning*. Deliberation refers to the process by which an agent decides which of its desires it will pursue (i.e. which desires become intentions). The means-ends reasoning is a procedure followed by an agent in order to determine by which means (i.e. available actions) it will achieve an end (i.e. an intention). Quite frequently, the means-ends reasoning process is implemented by planning techniques. A planner is a component of the agent that receives as input an intention or goal, a set of actions available to the agent, and the set of agent's beliefs. This planner then outputs an execution flow that the agent should follow in order to hopefully achieve the goal given as input.

*This work is partially supported by research project STRONGSOFT (TIN2012-39391-C04-02) from the Spanish Ministerio de Economía y Competitividad

Submitted to:
PROLE 2014

Traditionally, the control mechanisms implemented by agent-oriented programming languages in order to determine the agent's deliberation and means-ends reasoning are complex to use and often require further coding using different programming languages, like Java in Jason or a specific meta-language in the case of 3APL [9]. In the case of Jason, this added control mechanisms are necessary to deal with common situations that, if not handled, may cause undesirable outcomes. One such potential problem is that a Jason agent drops all the goals that cannot be immediately handled (due to the lack of applicable plans). Another potential problem is related to the necessity of specifying an execution order for the different intentions (i.e. the different execution flows available) of an agent since the order in which different intentions are executed is not always irrelevant. To deal with these potentially problematic situations, Jason programmers must write their own Java code.

To improve dealing with the aforementioned contingencies, in this paper we propose a series of mechanisms based on the introduction of syntactic constructs, as well as a modification of the semantics of the Jason interpreter, that tackle those problematic situations. In order to test their effectiveness, these mechanisms have been implemented in eJason [8], our implementation of Jason in the Erlang programming language, and tested in some small examples. Our approach has the following advantages: (i) gives more control over the agent to the programmer, facilitating the programming of agents (ii) the mechanisms proposed have been incorporated to the Jason language itself in a declarative, elegant manner, and (iii) the implementation of these mechanisms into the eJason interpreter is straightforward.

The rest of the paper is organized as follows. A brief introduction to Jason is given in Sect. 2. The problems identified in Jason and our proposed solutions are described in Sect. 3. Some approaches to similar problems implemented in different agent-oriented programming languages are enumerated in Sect. 4. Finally, some concluding remarks are discussed in Sect. 5

2 Background: Jason

Jason was first introduced in [1, 2] as an interpreter (implemented in JAVA) for an extension of the programming language AGENTSPEAK(L). This language extension has since been known as the Jason programming language [3]. The programming language AGENTSPEAK(L) was introduced in [11] but remained an abstract language, being Jason its first practical implementation. Jason is therefore inspired by the popular Belief-Desire-Intention (BDI) architecture [4, 5]. The agent's mental state is defined, in Jason, by the first-class citizens of the language, i.e. beliefs, goals (desires) and plans (intentions).

The agent's *beliefs* represent the information that the agent possesses about the world (i.e. about its environment or even about the agent itself). It is important to notice that there is no guarantee about the accuracy of the information represented by an agent's beliefs. The agent's *achievement goals* are the situations or states of affairs that the agent might like to bring about. The agent's *test goals* pursue the extraction of information from the agent's belief base. The agent's intentions are the desires that an agent has committed itself to accomplish which, in Jason, corresponds to the means (i.e. the execution flow, determined by the agent's *plans*) selected with the aim of fulfilling such desires. The intentions drive the behaviour of an agent, as they determine which actions such agent takes. In Jason, an intention is represented as a stack of plans such that the plan on top of the stack is the one being currently executed. Each plan in an intention stack deals with a subgoal of the plan immediately below in that same stack (i.e. as part of the execution of this latter plan, a new goal was introduced).

The mental state of an agent, together with the Jason interpreter, defines the behaviour of each agent. The Jason interpreter is embodied by an iterative procedure known as the *reasoning cycle*. In a nutshell, along each iteration of its reasoning cycle, an agent perceives its environment (including the processing

of incoming messages from other agents), deliberates over the possible course of action to take in order to fulfill its desires and performs some actions that modify its environment. A brief description of the building blocks of the mental state of an agent, as well as the Jason reasoning cycle, is provided below.

The mental state of a Jason agent is composed by: the agent's belief base, that contains all the agent beliefs; the set of events, which includes the observable events for the agent, i.e. changes in the belief base (addition/deletion of beliefs) as well as the agent's goals; and the plan base, that contains a set of pre-defined plans (implemented by the programmer) that provide recipes to fulfill the agent's desires. These plans are divided in three sections: the *triggering event*, which provide a pattern that identifies the events that can be handled with such plan (i.e. the events for which the plan is relevant); the *context*, that provides a series of conditions that determine whether a relevant plan is applicable; and the *body*, that contains a series of instructions (also referred to as formulas) that must be executed sequentially. These instructions may imply the addition/deletion of beliefs, the execution of actions (e.g. sending a message) or the addition of new goals. The syntax and semantics of the operators for goal addition, which are specially relevant to the present work, are the following:

- **!g:** an achievement goal addition event, $+!g$ is generated. When this event is selected for consideration, if there are no applicable plans, the event is dropped.
- **?g:** a test goal addition event, $+?g$ is generated. This event is dropped if no applicable plans can be found after its selection by the reasoning cycle.

During each iteration of the reasoning cycle, the main steps taken (some cleanup operations are not described) are:

1. The belief base is updated using the information obtained via perception and message-passing.
2. One of the events from the set of events is selected for execution. It is carried out using the agent-specific *event selection function*. Note that this event may carry a related intention, i.e., the intention that introduced the subgoal corresponding to the event.
3. The set of relevant and applicable plans for the selected event are computed. If no applicable plan can be found, the selected event is disregarded (dropped) and, if the event possesses a related intention, a plan failure is added. For simplicity, we do not deal with plan failure in this work, it suffices to consider that such related intention is dropped as well.
4. One of the applicable plans is selected for execution using the agent-specific *option selection function*. The selected plan is put on top of the stack representing its related intention, if any, or as the only element of a new stack. The resulting intention is added to the set of intentions.
5. One of the intentions is selected for execution using the *intention selection function*. The next formula from the plan on top of the selected intention is executed.

In each iteration, one element from different sets (events, applicable plans and intentions) must be chosen. This choice is determined by different agent-specific selection functions. The Jason interpreter implements default versions of such functions that are used when the programmer does not specify its customised ones. This customisation requires the Jason programmer to provide the Java code for the selection functions. Therefore, it does not suffice for the programmer to know the semantics of Jason, but he/she requires knowledge about the implementation of the interpreter. The fact that an agent possesses several intentions (i.e. several foci of attention) enables the agent to carry out several tasks concurrently. However, these intentions compete for the agent's attention, which, as we describe in Section 3, may introduce some vulnerabilities.

We believe that there exists a correspondence between the synchronisation necessities derived from the multiplicity of foci of attention implemented by Jason and those derived from the existence of several concurrent processes. For the sake of clarity, consider a mapping such that the different intentions of an agent represent the different processes that run concurrently in a computer. These processes compete for the use of the processor (in a similar way as the intentions compete for the agent's attention) and share a common memory space (as the intentions share access to the agent's belief base). A scheduler decides which process must be executed by the processor at anytime. Consequently, requesting the Jason programmer to provide ad-hoc selection functions, may be considered similar to asking a programmer to implement its own scheduler (i.e. a complex, error-prone task).

Jason provides a syntactic mechanism to address the syncronisation of intentions, namely *atomic plans*. These plans, identified in the Jason source code by a label *[atomic]* are such that, when the intention to which they belong is selected for execution, the focus of attention of the agent is fixed (i.e. does not change) until the plan is completely executed. The use of atomic plans may relieve the programmer from the necessity of implementing intention selection functions. However, as we show in Section 3.2, this mechanisms does not suffice to satisfy all common synchronisation needs.

3 Regaining Agent Control

In this section, we introduce some of the main difficulties that we have faced as Jason programmers. The problems identified derive from the semantics of the interpreter for the language. We motivate and describe our proposed solutions to recurrent necessities. Besides, we briefly report on the complexity of the inclusion of such solutions in eJason, our implementation of the Jason interpreter.

3.1 Synchronization mechanisms

As introduced in Section 2, the interpreter of Jason allows the agent to possess several foci of attention, corresponding to the different intentions of the agent. These intentions compete for the attention of the agent and the decision on which intention to execute, in each iteration of the reasoning cycle, is determined by the agent's intention selection function. The execution order of the different intentions is not always irrelevant. The plans in the different intentions access and update the information stored in the agent's belief base. Therefore, the modification of the set of beliefs, derived from the execution of an intention, may affect the outcome (or even totally prevent the execution) of the rest of intentions available. The programmer must then consider these data dependencies between the different intentions of an agent's mental state and, if necessary, control the synchronisation of the execution of such intentions.

Data dependencies are more likely (whereas not exclusive) among intentions that contain different instances of the same pre-defined plan, where that plan reads and updates some belief. As a matter of example, consider an agent which maintains a counter of, e.g., the number of files that it has uploaded. The most simple approach is updating such counter (a belief `files_loaded(Num)`) every time that a file is uploaded. This behaviour is implemented by the following Jason plan:

```
+!load(File) <-
   load(File);             // a)
   ?files_loaded(Num);     // b)
   -+files_loaded(Num+1). // c)
```

Consider an agent with only this plan in its plan base and with initial goals g_1 = `!load(file1)` and g_2 = `!load(file2)`. The intentions corresponding to these goals are composed by one instance

of the plan above, i.e. p_1 and p_2 with plan bodies $[a_1, b_1, c_1]$ and $[a_2, b_2, c_2]$, respectively. If we run this agent several times, using the standard intention selection function (i.e. picking one intention at random), we can observe that the counter is not always properly updated, as it only records the upload of one file, when there have been two of them. We can explore then the possible execution traces, represented by the different interleavings of the actions in the plan bodies. The analysis of the execution traces shows that the undesired outcome occurs when the actions b_1 and b_2 (i.e the actions where the value of the counter is read) have been executed without the execution of neither c_1 nor c_2 (i.e. the actions that update the counter) in-between (e.g. traces with the prefix a_1,b_1, a_2 b_2). The reason is that, in these cases, one of the intentions is handling outdated information about the counter and, therefore, the result is incorrect.

A possible solution for this kind of data dependency is the use of an atomic plan in order to update the counter, i.e. replacing the plan above for the plans:

```
+!load(File) <-
   load(File);
   !update_counter.

[@atomic]
+!update_counter <-
   ?files_loaded(Num);
   -+files_loaded(Num+1).
```

This way, the aforementioned faulty execution traces are not allowed. This solution can be used not only for belief updates, but also to implement behaviours that require the agent to maintain its focus of attention on the same intention for several iterations of the reasoning cycle (i.e. executing the formulas from the same intention consecutively). For instance, when a programmer tries to minimise the time that an agent requires exclusive access to a shared resource like, e.g. to write the result of certain computation into a file, the following plans can be used:

```
+!task(A,B, File) <-
   compute(A,B,Result)
   !write_result(Result, File);
   [...] // some other operations
   clean_up.

[@atomic]
+!write_result(Result,File) <-
   open_file(File);
   write(Result,File);
   close(File).
```

However, we find the necessity of introducing additional plans for the management of (almost) every belief update not ideal from a programmer's perspective. It unnecessarily obscures the code, as it increases the number of plans in the agent's plan base, consequently increasing the complexity to maintain the agent's code. Therefore, we propose here an alternative, elegant, approach providing a similar synchronisation functionality. This solution is equivalent to the *critical section* synchronisation mechanism implemented by several classical programming languages like, e.g. the C programming language [10], in order to control the synchronisation of (concurrent) multiprocessed software applications. Consider again the analogy between the multiplicity of foci of attention and multiprocessed systems, introduced in

Section 2. Sometimes, the programmer requires certain degree of control over the scheduling procedure in order to, e.g., prevent the potential hazards derived from the data dependencies existing among the processes. One of the most popular approaches to this problem is the implementation of critical sections, i.e. sections of a program such that, whenever a process p is executing the code within a critical section, no other process p' can enter such section until p has left it. Such critical sections typically delimit code regions that access and update shared resources (e.g. a program variable shared by different execution threads).

Inspired by this classical solution, we propose the introduction of some symbols that enable the specification of critical sections within the body of Jason plans. The symbols proposed are "{{" and "}}" to delimit the beginning and end of a critical section, respectively. The semantics of the Jason critical sections proposed slightly vary from those typically implemented in other programming languages in that when an agent executes a formula within a critical section, there can be no change in the focus of attention as long as the critical section is not left (as happens during the execution of an atomic plan). Therefore, the code for the agent that updates a counter, introduced earlier in this section, would be:

```
+!load(File) <-
   load(File);
   {{?files_loaded(Num);
   -+files_loaded(Num+1)}}.
```

This plan provides the same functionality as the combination of the two plans above, whereas, in our opinion, in a more elegant way.

Note that the customisation of the intention selection function requires the programmer to consider the different intentions that may conflict and establish a priority order for their execution. In contrast, using our proposed critical sections, the programmer only identifies sections of the code that should be executed without interference from other intentions of the agent. It is the task of the programmer to implement minimal critical sections (i.e. ones that are as short as possible).

3.2 Intention selection

Consider a simplified multi-agent system showing a classical client-server architecture. The client agents must write information into different files. In order to avoid conflicts generated by simultaneous attempts of writing different information into the same file (by different agents), the access to the file is managed by a server agent. A client must request to the server (by sending a message to it) the exclusive rights to access a file before it can write into such file. When the exclusive access to the file is no longer necessary, the client agent asks the server (again, via message passing) to "unlock" the resource. The server agent handles the different requests from the agents. The Jason code for the client agents is included in Figure 1. It shows that, when a client agent has the goal of writing some text `Text` into a file `FName`, it sends an achieve message to the server requesting the lock over the file (i.e. delegating a goal of the shape `!lock(FName)` to the server). Then, it adds a mental note, `+waiting_for(FName,Text)`, that records the text and file in which to write. The client agent is notified about the acquisition of exclusive access to the file by a belief update event `+granted(FName)` (see the server code explanation below), handled by the second plan. This plan amounts to actually writing the text into the file (this information has been obtained in the plan context from the corresponding mental note), deleting the mental note added in the previous plan and requesting the server to unlock the file (again, delegating this task as an achievement goal). Note that the client agent requires the execution of two different plans (not two different alternatives) to fulfill its desire of writing some text into a file. This necessity derives from the

```
+!write(FName, Text) :true <-
   .send(server, achieve, lock(FName));
   +waiting_for(FName,Text).

+granted(FName):waiting_for(FName,Text)<-
   write(Text, FName);
   -waiting_for(FName,Text);
   .send(server, achieve, unlock(FName)).
```

Figure 1: Jason code for the client agents

```
+!lock(FName)[source(Client)] :    //PSrv1
   file(FName)&not blocked(_,FName)<-
+blocked(Client,FName);
.print("Agent ",Client," locks ",FName);
.send(Client, tell, granted(FName)).

+!unlock(FName)[source(Client)] :  //PSrv2
   file(FName) & blocked(Client,FName) <-
-blocked(Client,FName);
.print("Agent ",Client," unlocks ",FName);
.send(Client, tell, unlocked(FName)).
```

Figure 2: Jason code for the file server agent

fact that a belief update (a notification from the server agent) must occur for the desire to be fulfilled. In Section 3.3 we propose an alternative that removes such necessity. The Jason code for the server is given in Figure 2. The goal to lock some file, delegated from some client `Client`, requires such file to exist and not to be blocked by other agent. If these conditions hold, the first plan can be applied, which amounts to adding a mental note, `+blocked(Client, FName)`, recording that `Client` has exclusive access to the file `FName`. Then it notifies the client by sending a tell message with the belief `granted(FName)`. The goal to unlock a file checks whether the file exists and whether it is locked by the same agent that attempts to unlock it.

If we run this system several times (to explore different interleavings), we observe that, sometimes, the locks are not properly granted, as several agents acquire access to the same file simultaneously. Analysing the different execution traces we notice that the problem derives from the program semantics implemented by the Jason interpreter, described in Section 2. Consider the following excerpt of an execution trace for the server agent (recall that the behaviour of an agent is determined by the computation carried out within the different steps that compose an iteration of the agent's reasoning cycle):

Iteration #1:

The belief base contains one belief: `file(f1)`.

The set of events includes the goal additions corresponding to lock requests for the same file `f1` from client agents `cl1` and `cl2`:
{`+!lock(f1)[source(cl1)]`, `+!lock(f1)[source(cl2)]`}.

The selected event is `+!lock(f1)[source(cl1)]`

The plan PSrv1 is applicable, as its context conditions `file(f1)` and `not blocked(_,f1)` hold, and is therefore added to the set of intentions (labelled PSrv1-f1cl1).

The set of intentions contains an intention other (which is irrelevant to this example) apart from the new one.

The intention selected for execution is other.

Iteration #2:

The belief base contains one belief: `file(f1).` Note that the belief `+blocked(cl1,f1)` has not been added yet, as the intention PSrv1-f1cl1 was not selected for execution in the previous iteration.

The set of events includes the goal addition not selected in the previous iteration: {`+!lock(f1)[source(cl2)]`}.

The selected event is `+!lock(f1)[source(cl2)]`

The plan PSrv1 is applicable, as its context conditions `file(f1)` and `not blocked(_,f1)` hold and is therefore added to the set of intentions (labeled PSrv1-f1cl2).

The set of intentions contains the intentions PSrv1-f1cl1 and PSrv1-f1cl2.

The intention selected for execution is PSrv1-f1cl1. Therefore, the first formula in its plan body is executed, which adds the belief `+blocked(cl1,f1).` As the plan is not fully executed, the intention containing remaining formulas is added to the set of intentions (labeled PSrv1-f1cl1').

Iteration #3:

The belief base contains two beliefs: {`file(f1)`, `blocked(cl1,f1)`}.

[...]

The set of intentions contains the intentions PSrv1-f1cl1' and PSrv1-f1cl2.

The intention selected for execution is PSrv1-f1cl2. Therefore, the first formula in its plan body is executed, which adds the belief `+blocked(cl2,f1).` As the plan is not fully executed, the intention containing remaining formulas is added to the set of intentions (labeled PSrv1-f1cl2').

Iteration #4:

The belief base contains three beliefs:
{`file(f1)`, `blocked(cl1,f1)`, `blocked(cl2,f1)`}.
Note that both clients `cl1` and `cl2` obtain the lock over file `f1` simultaneously.

[...]

The problem here derives from the fact that the server does not effectively register a lock over a file until the first formula in an instance of the corresponding plan PSrv1 is executed. Therefore, as shown by the execution trace above, there can be several instances of this plan, for the same file, in the set of intentions, leading to write conflicts.

The solution to this kind of problem requires the registration of a lock to happen only when the plan context holds. Therefore, both checking the truth value of the context and executing the formula that registers the lock shall happen in the same iteration of the reasoning cycle. It can only be achieved by the customisation of the agent's intention selection function in order to ensure that whenever a new instance of the plan PSrv1 is added to the set of intentions, the corresponding intention (the one that

```
+!lock(FName)[source(Client)] :    //PSrv1
   file(FName)&not blocked(_,FName)<-
{{+blocked(Client,FName)}};
.print("Agent ",Client," locks ",FName);
.send(Client, tell, granted(FName)).

+!unlock(FName)[source(Client)] :  //PSrv2
   file(FName) & blocked(Client,FName) <-
-blocked(Client,FName);
.print("Agent ",Client," unlocks ",FName);
.send(Client, tell, unlocked(FName)).
```

Figure 3: Jason code for the file server agent using critical sections

contains this instance on top of the stack) is selected for execution during that same iteration. This way, a belief `blocked(Client,FName)` is immediately added, preventing further inclusions of instances of PSrv1 for the same file before this file is unlocked. Note also that this problem cannot be solved just with the use of atomic plans, as an applicable atomic plan is not necessarily chosen for execution during the same iteration in which it is added.

Our proposed solution implies a modification of the Jason interpreter such that, when the intended means for some event start with a critical section (i.e. when the first formula in the body of the plan on top of the last added intention belongs within a critical section), that intention is selected for execution. This way, the programmer can easily identify the plans that, when deemed applicable, should immediately get the agent's attention. The implementation of the server agent using the code given in Figure 3 would then suffice, as the files are immediately locked.

3.3 Goal management

Even though the code in Figure 3 removes the problems derived from having several agents simultaneous blocking the same file, the system does not yet provide the desired functionality. An analysis of the new execution traces shows that, whenever a server agent gets the lock over a file, the requests from different client agents to lock the same file are disregarded (just dropped) by the server agent, as the context of the relevant plan `PSrv1` cannot be satisfied.

The simplest solution to avoid dropping all the achievement goals that cannot be immediately handled (due to the lack of applicable plans) implies adding a new plan, `PSrv3`, whose context matches whenever the file is already blocked by a different client agent. By following this plan, the server agent records the requests that cannot be immediately served by, e.g. returning the achievement goal addition event to the set of events:

```
+!lock(FName)[source(Client)] :    //PSrv3
   file(FName)&  blocked(_,FName)<-
!lock(FName)[source(Client)]. // requeue
```

This solution requires, then, computing a plan context that is satisfied whenever the context of the other (preferred) plans is not. This computation may be trivial for plans with simple contexts, like the one in the example above, but greatly increases its complexity for less simple plans, rapidly becoming a hard-to-solve satisfability problem.

In our opinion, the Jason interpreter (as an interpreter of AgentSpeak), by automatically dropping momentarily unmanageable goal addition events, requires the programmer to handle some time constraints (i.e. the moment at which the agent's reasoning cycle must attempt the computation of the sets of relevant and applicable plans for such events) that are highly demanding, given the weak guarantees typical of distributed systems. This requirement translates into strengthening the vulnerability of multi-agent systems to race conditions. As a matter of example of impact of race conditions, consider the following Jason code for an agent:

```
at(office). // Initial belief

!go(home). // Initial goals
!read(book).

+!read(Item): at(home) <-
  read(Item).

+!go(home): at(office) <-
  drive(home).
```

This agent initially believes to be at the office and simultaneously possesses the desires of going home and reading a book. There are two possible outcomes for the execution of this agent. In both of them, the agent goes home (as the plan to accomplish such desire is applicable). However, the agent does not always satisfy its desire of reading a book, because, if it tries to find an applicable plan for this desire before going home, the desire is automatically dropped.

Our proposed solution consists in the modification of the semantics of the achievement goal addition operator "!", as well as the inclusion of a new test goal addition operator "??". The semantics of these goal addition operators is then:

- **!g:** an achievement goal addition event, $+!g$ is generated. When this event is selected for consideration, if there are no applicable plans, the event is returned to the agent's set of events. This way, the event can be selected again in the future. This semantics is also available, though it is not the default one, as a special configuration of the Jason interpreter described in [3]. Therefore, our proposal amounts to selecting such alternative semantics as the default one.
- **??g:** a test goal addition event, $+??g$ is generated. When this event is selected for consideration, if there are no applicable plans, the event is returned to the agent's set of events. The use of this operator allows the implementation of mechanisms to suspend the execution of intentions until certain conditions are met. This capability is useful to provide simpler code, as shown in Figure 4, where the behaviour of the client agent does no longer need the inclusion of two separate, though semantically dependent, plans.

The implementation of these mechanism into the Jason interpreter allows the implementation of the behaviour of the client and server agents, with the desired functionality, using the code included in Figure 4 and Figure 3, respectively.

3.4 Implementation into eJason

The different solutions proposed above are implemented into eJason. This implementation was rather straightforward. For instance, in the case of the semantics for the operator "!", it only required the

```
+!write(FName, Text) :true <-
   .send(server, achieve, lock(FName));
  ??granted(FName);
   write(Text, FName);
   .send(server, achieve, unlock(FName)).
```

Figure 4: Final Jason code for the client agents

addition of an event to the agent's set of events (implemented as a list). The semantics for the operator "??" amounts to suspending an intention (adding it to a special queue) and periodically re-checking its satisfability conditions. With regards to critical sections, the changes to the reasoning cycle that were necessary (i.e. maintaining the agent's focus of attention fixed) are similar to the ones for atomic plans (already implemented).

Our opinion is then that the benefits obtained from the inclusion of our proposed solutions are considerably higher than the effort required for their implementation.

The latest release of eJason can be downloaded at:

git : *//github.com/avalor/eJason.git*

4 Related Work

Several agent-oriented programming languages include mechanisms of control embedded into the syntax of the language. However, due to the singularities of each language, these mechanisms vary. The programming language 3APL [9] provides a meta-language that enables the definition of an order of preference for the different plans applicable on the basis of, e.g., their utility. This approach is similar to the one of Jason (i.e. the customisation of the option selection function) in that the programmer must consider the different plans that may conflict and implement a specific priority order. Similarly, the programming language JACK [7] allows the programmer to implement a series of meta-plans to help select the desired applicable plan.

The programming language JADEX [6] enables the programmer to easily assign a priority value (establishing then an order) to the different plans in the agent's plan base. Besides, a series of configuration settings are available in order to customise the agent's deliberation process (i.e. the equivalent to Jason's event selection function).

5 Conclusions

Agent programming languages facilitate the design and implementation of intelligent agents because provide all the necessary infrastructure, for instance, Jason provides all necessary mechanisms to design and execute agents following the BDI model. However, several of these agent-oriented languages are not entirely self-contained. Programmers may have to write some specific code to implement some desired agent behaviour, in particular related to the agent's deliberation and means-ends reasoning processes. Concretely, in this paper we have shown how some problems arise because of the execution order of agent's intentions, and because agents drop all the goals that cannot be immediately handled (due to the lack of applicable plans). Most agent-oriented programming languages require the programmer to use a different programming language to implement the necessary mechanisms of control to deal with

problems like the ones described above. In the case of Jason, this means that, in order to program agents that behave correctly, programmers need to write some additional Java code.

In this paper, we propose a series of mechanisms that enrich the syntax of Jason with new operators. Concretely, we give programmers the possibility of specifying critical sections, and we propose a modification of the semantics of the achievement goal addition operator "!", and the inclusion of a new test goal addition operator "??". The use of these operators is very intuitive, as they follow the declarative paradigm.

Summarizing, the advantages of our approach are: (i) it gives more control over the agent to the programmer, (ii) the mechanisms proposed are on the language level, and (iii) their implementation into the eJason interpreter is straightforward.

We have evaluated our approach in a number of small examples like the ones shown in this paper. Clearly, we need to conduct a more thorough evaluation to assess the advantages of our approach. In addition, a formal semantics of eJason is being developed.

References

[1] Rafael H Bordini & Jomi F Hübner (2006): *BDI agent programming in AgentSpeak using Jason*. In: *Computational logic in multi-agent systems*, Springer, pp. 143–164.

[2] Rafael H Bordini, Jomi F Hübner & Renata Vieira (2005): *Jason and the Golden Fleece of agent-oriented programming*. In: *Multi-agent programming*, Springer, pp. 3–37.

[3] Rafael H. Bordini, Michael Wooldridge & Jomi Fred Hübner (2007): *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons.

[4] Michael Bratman (1987): *Intention, plans, and practical reason*. Harvard University Press (Cambridge, MA).

[5] Michael E Bratman, David J Israel & Martha E Pollack (1988): *Plans and resource-bounded practical reasoning*. *Computational intelligence* 4(3), pp. 349–355.

[6] Lars Braubach, Alexander Pokahr & Winfried Lamersdorf (2005): *Jadex: A BDI-Agent System Combining Middleware and Reasoning*. In R. Unland, M. Calisti & M. Klusch, editors: *Software Agent-Based Applications, Platforms and Development Kits*, Whitestein Series in Software Agent Technologies, Birkhuser Basel, pp. 143–168. Available at `http://dx.doi.org/10.1007/3-7643-7348-2_7`.

[7] Paolo Busetta, Ralph Rönnquist, Andrew Hodgson & Andrew Lucas (1999): *Jack intelligent agents-components for intelligent agents in Java*. *AgentLink News Letter* 2(1), pp. 2–5.

[8] Á. Fernández-Díaz, C. Benac-Earle & L. Fredlund (2014): *Adding distribution and fault tolerance to Jason*. *Science of Computer Programming*. Available at `http://dx.doi.org/10.1016/j.scico.2014.01.007`.

[9] KoenV. Hindriks, FrankS. De Boer, Wiebe Van der Hoek & John-JulesCh. Meyer (1999): *Agent Programming in 3APL*. *Autonomous Agents and Multi-Agent Systems* 2(4), pp. 357–401. Available at `http://dx.doi.org/10.1023/A%3A1010084620690`.

[10] Brian W Kernighan, Dennis M Ritchie & Per Ejeklint (1988): *The C programming language*. 2, Prentice-Hall Englewood Cliffs.

[11] Anand S. Rao (1996): *AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language*. In Walter Van de Velde & John W. Perram, editors: *Agents Breaking Away*, *LNCS* 1038, Springer, pp. 42–55. Available at `http://dx.doi.org/10.1007/BFb0031845`. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96), Eindhoven, The Netherlands, 22-25 January 1996, Proceedings.

Validación de tiempos de respuesta usando pruebas basadas en propiedades (Work in Progress)

Macías López
MADS Group
University of A Coruña (Spain)
macias.lopez@udc.es

Laura M. Castro
MADS Group
University of A Coruña (Spain)
lcastro@udc.es

Los desarrolladores de software cuentan con una serie de estrategias de pruebas y herramientas disponibles para probar los aspectos funcionales del software a distintos niveles (desde unitario hasta de sistema), desde distintas perspectivas (caja blanca hasta caja negra) y con distintos objetivos (positivo y negativo). Sin embargo, esta abundancia no es tal cuando queremos probar requisitos no funcionales de un sistema.

Las pruebas basadas en propiedades (PBT) son una aproximación que ha demostrado ser eficiente y efectiva cuando queremos probar requisitos funcionales. El éxito de las herramientas de PBT entre los desarrolladores vendrá dado, entre otros, por el hecho de conseguir extender las técnicas de PBT para abarcar más tipos de requisitos, como los no funcionales. Así, no sólo podremos contar con los beneficios de PBT en una nueva área en la que no ha sido aplicada, sino que también ofrecemos un enfoque común cuando tengamos que abordar las pruebas de requisitos funcionales y no funcionales.

En este trabajo proponemos el uso de PBT como un enfoque válido para probar requisitos no funcionales. Así, presentamos un conjunto de propiedades reutilizables que se pueden usar para medir tiempos de respuesta y que nos servirán de modelo para diseñar propiedades para otros tipos de requisitos no funcionales (i.e., disponibilidad, seguridad, etc.). Presentamos también la aplicación de estas propiedades a un proyecto piloto industrial.

1. Introducción

Las pruebas software se identifican habitualmente con su parte *funcional*, es decir, con evaluar si el comportamiento observado de un sistema o componente se ajusta a la lógica de negocio de su especificación. Sin embargo, a las pruebas *no funcionales* se les está dando cada vez tanta importancia como a las primeras [9], incluso teniéndolas en cuenta en todas las fases del ciclo de vida del software.

Tradicionalmente, los requisitos *funcionales* no se centran, por ejemplo, en aspectos relacionados con el tiempo o la eficiencia; pero en dominios específicos el hecho de saber *cuándo* se obtienen los resultados o *cuánto tiempo* van a tardar en estar disponibles puede ser decisivo. De hecho, estos requisitos no funcionales a veces también son referidos como *extra funcionales*. Entre los muchos requisitos no funcionales que se conocen (disponibilidad, usabilidad, flexibilidad, interoperabilidad, seguridad, etc.) [25],el requisito de tiempo de respuesta es uno de los que primero saltan a la escena [22]. Además, los fallos relacionados con este requisito están ampliamente distribuidos en mucho software en producción [15,21].

En este trabajo presentamos una aproximación basada en propiedades para probar requisitos de tiempo de respuesta, además de una metodología para abordar las pruebas no funcionales como una caja negra. Hemos implementado un conxunto de propiedades para probar los tiempos de respuesta, en el lenguaje de programación Erlang, y demostramos su uso con varios ejemplos, incluyendo un proyecto piloto industrial: un servicio web de una empresa de televisión digital [1].

Submitted to:
PROLE 2014

En vez de implementar pruebas a partir de especificaciones escritas en lenguaje natural, o en vez de diseñar un modelo formal para describir un sistema o componente, las pruebas basadas en propiedades (PBT) usan sentencias declarativas para especificar las propiedades que el software debe satisfacer de acuerdo con su especificación. Con este enfoque, los casos de prueba pueden entonces ser generados a partir de las propiedades, un proceso que se puede automatizar, lo que permite ejecutar un gran número de pruebas para cada propiedad.

PBT se está convirtiendo en un método muy popular en la comunidad de los lenguajes funcionales. Para Haskell, Claessen y Hughes desarrollaron QuickCheck [10] en el año 2000; una versión comercial en Erlang [4] desarrollada por QuviQ se sacó al mercado en el 2006; y PropEr [26] se publicó como un clon en software libre a finales de 2010. Con todo, nunca se ha utilizado PBT para probar requisitos no funcionales, y este trabajo pretende cubrir ese espacio.

Para resumir, las principales contribuciones de este trabajo son:

- Una metodología de pruebas software que extiende las pruebas basadas en propiedades al campo de requisitos no funcionales.
- Un conjunto de propiedades para probar requisitos de tiempo de respuesta, que se puede usar con las herramientas de pruebas basadas en propiedades en el contexto de la programación funcional, concretamente, en Erlang.
- La aplicación de esta metodología y propiedades desarrolladas para probar los tiempos de respuesta de un proyecto piloto industrial.

El trabajo está estructurado como sigue. Primero, introducimos en la Sección 2 las principales áreas de nuestra investigación: pruebas de requisitos no funcionales y PBT. Después, en la Sección 3 explicamos cómo hemos abordado la definición de las propiedades de rendimiento. Usando éstas, en la Sección 4 mostramos cómo se pueden derivar pruebas de tiempos de respuesta en un escenario industrial. Una discusión de las contribuciones y trabajo relacionado están incluidos en la Sección 5. Por último, en la Sección 6 presentamos una serie de conclusiones del trabajo.

2. Estado del arte

2.1. Requisitos no funcionales (NFRs)

Existe un consenso general en la ingeniería del software acerca del significado de *requisitos funcionales*, pero no sucede lo mismo en lo a que los *requisitos no funcionales* se refiere. Los primeros se suelen definir como 1. una función que el sistema tiene que ser capaz de ejecutar [29], 2. lo que el (software) producto debe hacer [31] o 3. lo que el sistema debería hacer [32]. Todas estas definiciones contemplan implícitamente aspectos operacionales del software (*función*, *sistema*).

Por el contrario, existe una amplia variedad de posibles definiciones para requisitos *no funcionales*. En [3] se describen como las características de un sistema relacionadas con las propiedades y restricciones sobre las que el sistema debe funcionar. De acuerdo con [20], un requisito no funcional especifica restricciones físicas con uno funcional, tales como restricciones del entorno o de implementación, tiempos de respuesta, dependencias de la plataforma, mantenibilidad, extensibilidad y fiabilidad. Una definición más general se puede encontrar en [34], afirmando que un requisito no funcional es un requisito que describe *no qué* hace el software, sino *cómo* lo hace; como ejemplos: requisitos de tiempo de respuesta,

requisitos de interfaces externas, restricciones de diseño o de calidad. Una lista completa de definiciones se puede encontrar en [17].

La comunidad de ingeniería de requisitos ha propuesto modelos y lenguajes de especificación para abordar el problema de la definición de requisitos funcionales [8, 30, 35], pero, una vez más, no se tienen en cuenta las características no funcionales de un sistema. De hecho, los requisitos no funcionales, por lo general, se describen de manera informal, y a menudo son contradictorios y difíciles de probar antes de la puesta en producción de un sistema software.

A veces, se hace referencia a los requisitos no funcionales como las propiedades de *calidad* del software. Sin embargo, usar este término para definir sólo los requisitos no funcionales es también problemático, porque cualquier requisito puede tener relación con la calidad de un sistema, tal y como aparece reflejado en la ISO 9000:2005 [19]. Además, el mismo requisito puede ser considerado como funcional y/o no funcional dependiendo de como los expresemos (el nivel de detalle que presente el documento de requisitos, por ejemplo) [33]. Finalmente, la importancia dada a los requisitos no funcionales depende del dominio específico para el que se desarrolle el software: como ejemplo, los requisitos de rendimiento son más decisivos en sistemas embebidos o de tiempo real que en otros escenarios.

Esta falta de una definición apropiada para los requisitos no funcionales ha guiado algunas líneas de investigación en años recientes [12]. Por ejemplo, en [18] los requisitos del software se definen en función de su tipo, representación, satisfacción y rol. También se formularon numerosas clasificaciones para los requisitos no funcionales [23]. El estándar 830-1993 IEEE en Especificaciones de Requisitos del Software [11] los clasifica en (1) requisitos de interfaces externas, (2), requisitos de rendimiento, (3) atributos, y (4) restricciones de diseño, donde los atributos son un conjunto de propiedades de calidad como fiabilidad, disponibilidad, seguridad, etc. Una clasificación más general [32] distingue entre requisitos de proceso (restricciones durante el proceso de desarrollo del sistema), requisitos de producto (las características deseadas que un sistema debe tener), y requisitos externos (derivados del entorno en el que el sistema se va a desplegar). Una lista completa de requisitos no funcionales se puede encontrar en [9].

2.2. Pruebas basadas en propiedades

Las pruebas basadas en propiedades (PBT) son un enfoque de prueba que se apoya en la generalización de los casos de prueba abstrayéndolos de sus entradas y salidas concretas [13]. Usando esta técnica, en vez de proporcionar un oráculo para comparar el valor obtenido con el valor esperado, se escriben predicados genéricos que establecen una relación lógica. En otras palabras, en vez de proporcionar una lista de pares de entrada/salida que describen parcialmente los requisitos, se escribe una sentencia verificable y aplicable a todas las combinaciones de entradas/salidas.

En PBT, la ejecución del test se realiza, normalmente, con la ayuda de una herramienta que usa generadores de datos aleatorios que, al instanciarlos, producen un gran número de casos de prueba, para después evaluar el contenido de la propiedad y probar automáticamente los valores generados. Aunque a día de hoy existen dos herramientas de PBT para Erlang, para el propósito de este trabajo vamos a usar el nombre de QuickCheck para referirnos a cualquiera de ellas, dado que ambas comparten las funcionalidades en las que estamos interesados. Las propiedades se escriben usando el lenguaje Erlang y con una serie de macros y funciones específicas de las bibliotecas de QuickCheck. Así, el proceso de ejecutar pruebas basadas en propiedades se divide en dos pasos:

- La definición de *generadores de datos* que se usan para producir datos de entrada, además de la esperada distribución de probabilidad de los datos aleatoriamente generados. Para QuickCheck, la biblioteca `eqc_gen` proporciona funciones para construir estos generadores.

- La definición de las *propiedades esperadas* del software, derivadas a partir de la especificación. Normalmente, éstas son propiedades cuantificadas universalmente, en las que los datos producidos por los generadores se ligan a las variables universales. En QuickCheck, el cuerpo de la propiedad definida puede estar especificada como una máquina de estados, ejecutando casos de prueba que consisten en secuencias de entradas. Esta característica es muy útil a la hora de probar sistemas con estado.

Como ejemplo, tomemos esta versión incorrectamente implementada de la función de Fibonacci en Erlang:

```
fib(0) -> 1;
fib(1) -> 1;
fib(N) -> fib(N-2) + fib(N-1).
```

Para probar esta función `fib` usando PBT y QuickCheck, podemos escribir un generador de números naturales cuyos primeros valores generados se aproximen a `40`:

```
large_nat() ->
  resize(40, nat()).
```

Las funciones `resize` y `nat` las proporcionan las bibliotecas de QuickCheck. La primera se usa para ligar el tamaño de generación a `40`, y la segunda para generar números naturales. Antes de usarla, podemos mostrar el aspecto que tienen los datos de entrada que proporciona este generador. Podemos hacerlo interactivamente en la *shell* de Erlang:

```
> eqc_gen:sample(nfr_eqc:large_nat()).
2
15
33
29
15
21
...
ok
```

donde `nfr_eqc` es el nombre del módulo donde las funciones están definidas. Una vez satisfechos con el generador de datos, podemos ahora escribir una propiedad sobre la función `fib`. Por ejemplo, podemos probar que *el máximo común divisor de dos números de Fibonacci cualquiera es también un número de Fibonacci* [39]:

$$gcd(fib(a), fib(b)) = fib(gcd(a,b))$$

donde a y b son cualquiera número natural. Traducir la propiedad matemática anterior a una propiedad de QuickCheck es tan directo como:

```
prop_gcd() ->
  ?FORALL(A, large_nat(),
     ?FORALL(B, large_nat(),
        gcd(fib(A),fib(B)) == fib(gcd(A,B)))).
```

donde las variables `A` y `B` están ligadas a los valores generados por la función `large_nat`. La función `gcd` la implementamos usando el algoritmo de Euclides:

```
 gcd(0,A) -> A;
 gcd(A,B) -> gcd(B, A rem B).
```

Ahora podemos ejecutar nuestra propiedad usando QuickCheck y así producir tantos casos de prueba concretos (es decir, pares de variables `A` y `B` ligadas a números naturales mayores que 40, que se usarán en el cuerpo de la propiedad) como queramos, ejecutarlos, y evaluarlos:

```
> eqc:quickcheck(nfr_eqc:prop_gcd()).
...Failed! After 4 tests.
{17,35}
Shrinking..(2 times)
{0,2}
false
```

Una interesante característica de QuickCheck es que, cuando encuentra un caso de prueba que falla, la herramienta lo reduce automáticamente al contraejemplo más pequeño que produce el mismo fallo, haciendo así más fácil la tarea de encontrar la razón de fallo del caso de prueba, y mejorando el proceso de depuración. En el ejemplo anterior, el primer par de valores de `A` y `B` utilizados por QuickCheck que producen un error (es decir, que no cumplen la propiedad porque se evaluó a `false`) son el par de números naturales `{17, 35}`. QuickCheck reduce este caso de prueba fallido a un contraejemplo mucho más pequeño: el par `{0, 2}`. Éste puede parecer un ejemplo muy simple del mecanismo de reducción de contraejemplos, pero cuando tenemos que abordar las pruebas de sistemas más complejos, la habilidad de reducir grandes entradas a la más pequeña es muy útil [41].

Naturalmente, el contraejemplo revela nuestra implementación defectuosa de `fib`: el Fibonacci de `0` es igual a `0`, no a `1`. De hecho, resulta obvio ver que `fib(gcd(0,2))` no es igual a `gdc(fib(0),fib(2))`. Corrigiendo la implementación, podemos ejecutar la propiedad de nuevo y comprobar que se satisface para los 100 casos de prueba que se generan automáticamente:

```
> eqc:quickcheck(nfr_eqc:prop_gcd()).
.....................................................................
.....................
OK, passed 100 tests
true
```

3. Pruebas de tiempo de respuesta con PBT

Nuestra principal pregunta de investigación es: *¿hasta qué punto son las pruebas basadas en propiedades un método válido para probar los requisitos no funcionales?* Para responderla, empezamos analizando qué requisito no funcional sería el más relevante, de acuerdo con la literatura, y hemos decidido centrarnos en el *tiempo de respuesta* o rendimiento [38].

Primero, implementamos una función para medir el tiempo real transcurrido cuando se llama a otra función. Hemos usado las bibliotecas de Erlang relacionadas con el tiempo, concretamente, el módulo `timer`:

```
response_time(Mod,Fun,Args) ->
  {MicroSeconds,_Value} = timer:tc(Mod,Fun,Args),
  MicroSeconds/1000.
```

La notación (`Mod`, `Fun`, `Args`) es ampliamente usada en la programación funcional (y consecuentemente, en Erlang): queremos evaluar la *función* `Fun` definida en el *módulo de implementación* `Mod` con la *lista de argumentos* `Args`. Así, la función `response_time` devuelve el tiempo que tarda en ejecutarse una determinada función, expresado en milisegundos. Nótese que para el propósito de pruebas no funcionales, no nos importa el valor de retorno (`_Value`), de hecho, lo descartamos.

Con esta función podemos calcular el tiempo que se tarda en calcular el Fibonacci de `10`:

```
> nfr_eqc:response_time(nfr_eqc, fib, [10]).
0.005
```

Ahora escribimos una propiedad que genere números naturales aleatorios y calcule el `response_time` de la función `fib` y compruebe que los valores devueltos están por debajo de un límite (en este caso, 1000 milisegundos):

```
prop_fib_performance() ->
  ?FORALL(N, nat(),
    begin
      ElapsedTime = response_time(nfr_eqc, fib, [N]),
      ?WHENFAIL(format_time(ElapsedTime), ElapsedTime < 1000)
    end).
```

En la definición de la propiedad, envolvemos la condición del oráculo `ElapsedTime < 1000` con la macro de QuickCheck `?WHENFAIL(Output, Condition)`, así, cuando la condición se evalúe a falso, la herramienta no sólo imprimirá el valor de entrada (`N`), sino también el tiempo de respuesta que provocó el fallo.

Si ejecutamos esta propiedad para probar si los tiempos de respuesta de las llamadas a `fib` son siempre menores que un segundo:

```
> eqc:quickcheck(nfr_eqc:prop_fib_performance()).
.....................................................................
.....................
OK, passed 100 tests
true
```

no conseguimos ningún caso de prueba fallido, presumiblemente porque las entradas son demasiado pequeñas. De hecho, si usamos el generador de números naturales mayores que 40, presentado en la Sección 2.2 y ejecutamos la propiedad de nuevo:

```
> eqc:quickcheck(nfr_eqc:prop_fib_performance()).
...Failed! After 4 tests.
40
7718.606 ms
Shrinking..(2 times)
36
1126.606 ms
false
```

Encontramos un caso de prueba fallido, correspondiente al tiempo de respuesta de `fib(40)`, ya que es mayor que 7 segundos. QuickCheck reduce este caso de prueba al mínimo contraejemplo, indicando que `fib(36)` tampoco cumpliría el comportamiento no funcional esperado. De hecho, podemos comprobar que el tiempo de respuesta de `fib(35)` está por debajo de 1 segundo:

```
> nfr_eqc:response_time(nfr_eqc, fib, [35]).
696.879
```

Por supuesto, esta propiedad se puede generalizar para que se pueda aplicar a cualquier llamada a módulo y función, con un generador de argumentos y un tiempo límite:

```
prop_simple_performance({Mod,Fun,ArgGen},TLimit) ->
  ?FORALL(Args, ArgGen,
    begin
      ElapsedTime = response_time(Mod,Fun,Args),
      ?WHENFAIL(format_time(ElapsedTime), ElapsedTime < TLimit)
    end).
```

Figura 1: Propiedad para probar el tiempo de respuesta de una llamada a función.

Se podría argumentar que medidas relevantes de los tiempos de respuesta sólo se pueden obtener ejecutando varias veces la llamada a una función, y así obtener una visión real del rendimiento. Con ese objetivo, hemos implementado otra propiedad que comprueba que la media de un determinado número (`T`) de tiempos de respuesta obtenidos son menores que un límite:

```
prop_avg_performance({Mod,Fun,ArgGen},T,TLimit) ->
  ?FORALL(Args, ArgGen,
    ?FORALL(ElapsedTimes, run_ntimes(T,{nfr_eqc,response_time,[Mod,Fun,Args]}),
      begin
        AvgTime = avg(ElapsedTimes),
        ?WHENFAIL(format_time(AvgTime),
                  AvgTime < TLimit)
      end)).
```

Figura 2: Propiedad para comprobar el tiempo medio de respuesta de varias llamadas a una función.

Estas propiedades demuestran que no sólo es posible, sino también la potencia de las pruebas basadas en propiedades para probar requisitos no funcionales. Éstas y otras funciones auxiliares forman el núcleo de nuestro trabajo en progreso para probar requisitos de tiempos de respuesta, una de las contribuciones de este trabajo.

4. Proyecto piloto industrial: VoDKATV

Con el objetivo de realizar una validación temprana de la aplicabilidad de nuestras propiedades de tiempos de respuesta, hemos usado un proyecto piloto industrial: VoDKATV. También lo hemos usado para ilustrar la metodología de pruebas que extiende las pruebas basadas en propiedades a los requisitos no funcionales, otra de las contribuciones de este trabajo.

VoDKATV es un middleware IPTV/OTT que proporciona a los usuarios finales acceso a diferentes servicios en una pantalla de TV, tablet, smartphone, PC, etc., lo que permite una avanzada experiencia multimedia multi-pantalla. VoDKATV es un sistema distribuido compuesto por varios componentes, que se integran a través de servicios web.

A la hora de tener en cuenta los requisitos no funcionales de VoDKATV, los desarrolladores especificaron que en determinados rangos de horas del día VoDKATV tiene que atender un alto número de peticiones HTTP, que coincide con los momentos en los que muchos usuarios están usando el reproductor multimedia o sus móviles al mismo tiempo. En este escenario, los requisitos de tiempo de respuesta son esenciales para la experiencia del usuario: cuando un usuario quiere navegar por una lista de canales, VoDKATV tiene que responder por debajo de un determinado límite.

4.1. Probando los tiempos de respuesta de VoDKATV

Como es habitual en el caso de los servicios web, la API que ofrecen los componentes de VoDKATV están especificados en WSDL [2]. La Figura 3 muestra un extracto del WSDL donde están definidas dos operaciones de VoDKATV, y que, como se puede observar, no incluye ningún tipo de información sobre el rendimiento o tiempos de respuesta de las operaciones:

- `FindAllVideoServers`, que no espera argumentos de entrada, y
- `FindDeviceById`, que toma como argumento `deviceId`, un entero, como entrada.

Para realizar peticiones HTTP a VoDKATV, usamos el *framework WSToolkit* desarrollado por Li et al [24] que, a partir de una descripción WSDL, deriva un módulo Erlang para hacer peticiones HTTP. Este módulo define un conjunto de funciones, una para cada operación WSDL. Dado que estas funciones realizan peticiones HTTP de manera transparente, ellas son las funciones objetivo sobre las que realizaremos las pruebas de tiempos de respuesta En este ejemplo estamos interesados en las funciones anteriormente mencionadas, `find_all_video_servers` y `find_device_by_id`. Primero, un requisito derivado de la especificación de VoDKATV es:

RQ: *el tiempo de respuesta de la operación* `FindDeviceById` *es menor que 100 milisegundos*.

Podemos proceder así:

```
> eqc:quickcheck(nfr_eqc:prop_simple_performance({vodkatv,
                                                  find_device_by_id,
                                                  eqc_gen:choose(1,250)}, 100)).
67.057 ms
65.781 ms
67.660 ms
65.587 ms
65.723 ms
65.813 ms
...
OK, passed 100 tests
true
```

donde hemos sacado varios de los tiempos de respuesta de cada caso de prueba, derivados de la propiedad ejecutada.

Las entradas para la función que estamos probando, `find_device_by_id`, proporcionada por el módulo `vodkatv` es un entero entre `1` y `250` (asumiendo que esos son identificadores válidos de VoDKATV), generados por el generador de QuickCheck `choose(RangeInit, RangeEnd)`.

```
...
<wsdl:operation name="FindAllVideoServers"
  pattern="http://www.w3.org/ns/wsdl/in-out"
  style="http://www.w3.org/ns/wsdl/style/iri"
  wsdlx:safe="true">

  <wsdl:output element="msg:videoServersResponse"/>

</wsdl:operation>
...
<wsdl:operation name="FindDeviceById"
  pattern="http://www.w3.org/ns/wsdl/in-out"
  style="http://www.w3.org/ns/wsdl/style/iri"
  wsdlx:safe="true">

  <wsdl:input element="msg:findDeviceByIdParams"/>
  <wsdl:output element="msg:findDeviceByIdResponse"/>

</wsdl:operation>
<xsd:element name="findDeviceByIdParams">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="deviceId"
                   type="xsd:integer"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
...
```

Figura 3: Extracto del WSDL de VoDKATV

Como podemos ver, después de ejecutar 100 casos de prueba, el tiempo límite de 100 milisegundos no se excede. Sin embargo, si cambiamos el requisito con un tiempo límite más bajo:

RQ': *el tiempo de respuesta de la operación* `FindDeviceById` *es menor que 50 milisegundos*.

y ejecutamos la propiedad de nuevo, obtenemos:

```
> eqc:quickcheck(nfr_eqc:prop_simple_performance({vodkatv,
                                                  find_device_by_id,
                                                  eqc_gen:choose(1,250)},  50).
68.527 ms
Failed! After 1 tests.
{122}
Shrinking.(1 times)
{1}
```

```
62.842 ms
false
```

El primer caso de prueba generado por QuickCheck, con entrada `deviceId = 122` muestra un tiempo de respuesta por encima de 68 milisegundos. El contraejemplo se reduce al entero `1` (`{1}`), presumiblemente porque en el contexto de la funcionalidad `find_device_by_id`, el identificador de dispositivo concreto que se use no es relevante para el tiempo de respuesta de la operación. En otras palabras, no existe una relación entre los argumentos de entrada y el tiempo de respuesta. Este hecho, sin embargo, debería ser contrastado con los desarrolladores del software, ya que las pruebas que estamos ejecutando son de caja negra. De todas maneras, lo que si obtenemos es una cota superior del tiempo de respuesta de la operación, del orden de 70 milisegundos.

Como ya hemos mencionado en la sección anterior, la propiedad complementaria que muestra la Figura 2 puede ser útil para aumentar la fiabilidad de los tiempos de respuesta de una determinada operación. Por ejemplo, podríamos querer probar si

RQ$_2$: *tomando 10 peticiones, el tiempo de respuesta medio de la operación* `FindAllVideoServers` *es menor que 100 milisegundos*.

que probaremos ejecutando

```
> eqc:quickcheck(nfr_eqc:prop_avg_performance({vodkatv_sut,
                                               find_all_video_servers,[]},10,100)).
96.8880000 ms
53.0957000 ms
54.1517005 ms
52.8144995 ms
55.7098999 ms
52.5515999 ms
52.8506000 ms
...
155.933299 ms
Failed! After 18 tests.
[]
[54.102,53.018,52.11,52.791,81.683,1054.12,52.01,
54.13,52.664,52.705]
155.9332999 ms
false
```

Usando la propiedad `prop_avg_performance`, como mostramos arriba, cada caso de prueba incluye `10` llamadas a la función que estamos probando. Después de 18 casos de prueba, QuickCheck encuentra uno donde el tiempo de respuesta medio de las 10 llamadas es mayor que 100 milisegundos. Como podemos ver en el contraejemplo, la razón para incumplir la propiedad es la presencia de un valor muy alto (1054,12 milisegundos), que dista mucho del resto de valores y que hace al fallar al caso de prueba por completo. En este caso, no se hace ninguna reducción del contraejemplo porque la función `find_all_video_servers` no recibe ninguna entrada.

Si volvemos a ejecutar esta propiedad varias veces más, descubrimos que esos valores anormalmente altos se van sucediendo constantemente cada cierto intervalo de tiempo cuando se ejecuta esta funcionalidad de VoDKATV. Aunque es muy difícil inferir la razón de este comportamiento usando pruebas de

caja negra, este ejemplo demuestra la utilidad de nuestras propiedades: somos capaces de sacar a la luz extraños comportamientos del software después de realizar varios cientos de peticiones al servicio web de manera automática.

4.2. Metodología para probar NFR usando PBT

La metodología expuesta en esta sección se puede reutilizar para probar los requisitos de rendimiento de cualquier componente o sistema, desde un enfoque de caja negra, usando pruebas basadas en propiedades.

Los pasos de la metodología propuesta son:

1. Identificar la lista de funcionalidades u operaciones de interés para las que se definen requisitos de rendimiento que se desean validar.
2. Para cada funcionalidad, definir un requisito de tiempo de respuesta estableciendo un tiempo como cota superior.
   - Usar la propiedad `prop_simple_performance` para probar y/o redefinir el valor límite, y por lo tanto el requisito de tiempo de respuesta de la función que estamos probando.
3. Para cada funcionalidad, definir un requisito de tiempo medio de respuesta estableciendo un tiempo como cota superior (idealmente, el límite refinado del paso anterior).
   - Usar la propiedad `prop_avg_performance` para descubrir posibles anomalías en los tiempos de respuesta del software, detectando posibles valores fuera de rango.

5. Discusión

Entre todos los requisitos no funcionales que aparecen en la literatura (cf. Sección 2.1), medir el tiempo de respuesta de una determinada funcionalidad es una de los problemas que con mayor frecuencia aparecen en las actividades de pruebas de software [22]. Probar los tiempos de respuesta y su degradación es también relevante [14, 40], incluso crítico en dominios como las telecomunicaciones [37]. Éste es el motivo por el que hemos seleccionado el tiempo de respuesta como el primer requisito no funcional en el que centrarse para responder a la pregunta de investigación que formulamos al principio de la Sección 3, y que nos llevó a implementar las propiedades presentadas en las Figuras 1 y 2, e ilustradas en la Sección 4 con el caso de estudio.

Dos aproximaciones principales se han usado en investigaciones en marcha en al campo de las pruebas de requisitos no funcionales: modelos y algoritmos genéticos [6]. Con respecto a la definición y uso de modelos, la mayor parte de la investigación se centra en extender el Lenguaje Unificado de Modelado (UML) [28] para especificar estos requisitos. Un ejemplo es UML Testing Profile (UML-TP) [5], una notación estándar para diseñar pruebas de caja negra que ayuda en el diseño, visualización, análisis y documentación de la fase de pruebas. Los autores de [16] proponen una aproximación para generar pruebas de tiempos de respuesta a partir de diagramas de actividad UML. Otras contribuciones han propuesto un *framework* completo [14] para especificar propiedades no funcionales desde un enfoque basado en modelos, pero normalmente están restringidos al nivel de especificación (y no permiten derivar los casos de prueba, ejecutarlos y evaluarlos). La traducción a casos de prueba, como mucho, se realizó en estudios muy específicos [7].

Por otro lado, el uso de algoritmos genéticos para encontrar el mejor y el peor valor de tiempo de respuesta se propuso en [36]. Más recientemente, los autores de [27] han usado esta aproximación para

derivar la mejor estrategia para generar casos de prueba de rendimiento, en arquitecturas orientadas a servicios (SOA, Service Oriented Architecture).

A diferencia de estas aproximaciones, la metodología (y propiedades) que proponemos nos permite abordar todas las fases relacionadas con las pruebas de requisitos no funcionales, desde la especificación hasta la ejecución de los casos de prueba. Utilizamos un lenguaje de programación funcional, Erlang, como lenguaje de especificación, lo que permite a quien realiza las pruebas trabajar a un nivel de abstracción más alto. El uso de PBT traslada el esfuerzo de escribir los casos de prueba concretos a modelar el comportamiento general del sistema, lo que hace más fácil el mantenimiento de la fase de pruebas. El uso de una herramienta como QuickCheck automatiza la derivación de los casos de prueba concretos usando generadores de datos, la recolección de resultados y la comparación con el oráculo: la especificación de prueba o propiedad. Por último, podemos usar la misma metodología (PBT) y herramienta (QuickCheck) para ejecutar tanto pruebas funcionales como no funcionales, proporcionando así un método unificado para medir la calidad del software.

Aunque la metodología para generar pruebas de rendimiento que hemos presentado, donde primero se miden los tiempos de respuesta y después se comparan, ya se usa en la industria a día de hoy, nuestra contribución se restringe al ámbito de la programación funcional. En el mundo Erlang, QuickCheck es ampliamente usado para escribir pruebas basadas en propiedades, pero nunca había sido usado para probar requisitos de rendimiento, un vacío que pretendemos rellenar con este trabajo.

Un asunto que ha salido a la luz durante esta investigación es el hecho de que el proceso de reducción de contraejemplos de QuickCheck resulta no ser significativo cuando no hay una relación entre las distintas entradas de una función y el tiempo de respuesta para cada una de ellas. Mientras que en ejemplo de la función de Fibonacci el incremento del tiempo es exponencial en relación al incremento en el valor del número natural generado (algo esperado debido al propio algoritmo en si), el caso de estudio de VoDKATV muestra dos ejemplos contrarios, donde el proceso de reducción no es necesario (porque la función que probamos no tiene argumentos), o siempre nos proporciona el valor más pequeño que el generador puede producir. Este hecho hace que el proceso de reducción de contraejemplo no sea interesante para depurar el programa.

Otro aspecto tiene que ver con la presencia de valores extraños cuando usamos la propiedad de probar el tiempo de respuesta medio. Ya que PBT es un enfoque de caja negra, actualmente no podemos proporcionar información adicional cuando la propiedad se incumple. Consecuentemente, es la persona que realiza las pruebas quien tiene que evaluar el comportamiento y tratar de dar una interpretación o significado, y proponer posibles causas del comportamiento anómalo. Los incumplimientos de requisitos no funcionales normalmente pasan desapercibidos durante las fases de pruebas de unidad o incluso de integración, ya que las pruebas de tiempos de respuesta normalmente se escriben para el sistema completo. Sin embargo, usando las pruebas basadas en propiedades podríamos obtener datos en fases anteriores, o podríamos combinar nuestra metodología de caja negra con estrategias de caja blanca para depurar el software. Por ejemplo, podríamos usar las propiedades que hemos desarrollado con la biblioteca `eqc_component` de QuickCheck para recolectar estos datos e incluso tomar o modificar decisiones de diseño del sistema.

Cuando estamos probando aplicaciones del mundo real, los requisitos de rendimiento usualmente están relacionados con escenarios de carga específicos en los que se deben ejecutar los casos de prueba. Ésto es particularmente importante en las aplicaciones SOA, que ofrecen una API a través de la web, y a la que realizan peticiones muchos clientes al mismo tiempo. La degradación de determinados servicios en escenarios de alta carga suele ser uno de los motivos principales de fallos, provocando interrupciones en el servicio. Las pruebas de tiempos de respuesta y de estrés (es decir, simular escenarios de alta carga y probar los servicios así) es uno de los aspectos más desafiantes de las pruebas software, especialmente

si queremos simular comportamientos "típicos" del usuario, ya que la tipificación del comportamiento habitual es muy dependiente del entorno de negocio y actividad.

Nuestras propiedades de tiempos de respuesta pueden mejorarse, incluyendo otras medidas estadísticas como la mediana o la desviación típica, sobre todo si tratamos con componentes o sistemas en los que durante las pruebas se quiere hacer caso omiso de valores atípicos, por considerarlos poco relevantes.

Por último, en relación a la pregunta de investigación acerca de la viabilidad para usar pruebas basadas en propiedades para probar requisitos no funcionales, hemos demostrado con nuestro trabajo que la respuesta es afirmativa, por lo menos en lo que a requisitos de rendimiento se refiere. De todas maneras, necesitamos investigar más para extender estas ideas a otros requisitos como seguridad, disponibilidad, flexibilidad, etc.

6. Conclusiones y trabajo futuro

En este trabajo hemos presentado un conjunto de propiedades para probar requisitos de tiempo de respuesta usando PBT, así como una metodología de aplicación, que hemos ilustrado con un proyecto piloto industrial.

Hemos mostrado como los requisitos de rendimiento se pueden definir y refinar, y como su no cumplimiento puede ser detectado usando las propiedades. Aún así, se necesita más investigación para descubrir las razones de comportamientos anormales detectados.

Planeamos añadir más propiedades para probar otros requisitos no funcionales, así como integrarlas en una arquitectura más general, capaz de especificar requisitos no funcionales desde un enfoque basado en modelos, en vez de escribir las propiedades a mano.

Otra línea de trabajo futuro es añadir a las propiedades mecanismos para incrementar/decrementar el uso de recursos de la máquina en la que se realizan pruebas (como CPU, memoria, disco o uso de red), y así probar los requisitos de tiempo de respuesta en esos escenarios.

7. Agradecimientos

Este trabajo ha sido parcialmente financiado por EU FP7-ICT-2011-8 Reference 317820.

Referencias

[1] *Interoud Innovation*. `http://interoud.com`.

[2] (2007): *Web Services Description Language (WSDL) 2.0*. `http://www.w3.org/TR/wsdl20/`.

[3] Ana I Anton (1997): *Goal identification and refinement in the specification of software-based information systems*.

[4] Thomas Arts, John Hughes, Joakim Johansson & Ulf Wiger (2006): *Testing Telecoms Software with Quviq QuickCheck*. In: *Proceedings of the ACM SIGPLAN Workshop on Erlang*, ACM Press, New York, NY, USA.

[5] Paul Baker (2009): *Model-driven testing: Using the UML testing profile*. Springer-Verlag.

[6] Antonia Bertolino (2007): *Software testing research: Achievements, challenges, dreams*. In: *Future of Software Engineering, 2007. FOSE'07*, IEEE, pp. 85–103.

[7] Antonia Bertolino, Antonello Calabrò, Francesca Lonetti, Antinisca Di Marco & Antonino Sabetta (2011): *Towards a model-driven infrastructure for runtime monitoring*. In: *Software Engineering for Resilient Systems*, Springer, pp. 130–144.

[8] B.H.C. Cheng & J.M. Atlee (2007): *Research directions in requirements engineering*. pp. 285–303.

[9] Lawrence Chung & JulioCesarSampaio Prado Leite (2009): *On Non-Functional Requirements in Software Engineering*. In: *Conceptual Modeling: Foundations and Applications*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, pp. 363–379.

[10] Koen Claessen & John Hughes (2000): *Quickcheck: a lightweight tool for random testing of haskell programs*. *ICFP*, pp. 268–279.

[11] IEEE Computer Society. Software Engineering Standards Committee, Inc. Electronics Engineers & IEEE-SA Standards Board (1998): *IEEE recommended practice for software requirements specifications: approved 25 June 1998*. IEEE.

[12] L.M. Cysneiros & J.C.S.P. Leite (2004): *Nonfunctional requirements: From elicitation to conceptual models*. *IEEE Transactions on Software Engineering* 30(5), pp. 328–350.

[13] J. Derrick, N. Walkinshaw, T. Arts, C. Benac Earle, F. Cesarini, L.-A. Fredlund, V. Gulias, J. Hughes & S. Thompson (2010): *Property-based testing - The ProTest project*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6286 LNCS, pp. 250–271.

[14] Antinisca Di Marco, Claudio Pompilio, Antonia Bertolino, Antonello Calabrò, Francesca Lonetti & Antonino Sabetta (2011): *Yet another meta-model to specify non-functional properties*. In: *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, ACM, pp. 9–16.

[15] Q. Fu, J.-G. Lou, Q.-W. Lin, R. Ding, D. Zhang, Z. Ye & T. Xie (2012): *Performance issue diagnosis for online service systems*. pp. 273–278.

[16] Antonio García-Domínguez, Inmaculada Medina-Bulo & Mariano Marcos-Bárcena (2013): *An Approach for Model-Driven Design and Generation of Performance Test Cases with UML and MARTE*. In: *Software and Data Technologies*, Springer, pp. 136–150.

[17] M. Glinz (2007): *On Non-Functional Requirements*. In: *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pp. 21–26.

[18] Martin Glinz (2005): *Rethinking the notion of non-functional requirements*. In: *Proc. Third World Congress for Software Quality*, 2, pp. 55–64.

[19] BSEN ISO (2005): *9000: 2005 Quality management systems. Fundamentals and vocabulary*. *British Standards Institution*.

[20] Ivar Jacobson, Grady Booch & James E Rumbaugh (1999): *The unified software development process-the complete guide to the unified process from the original designers*. Addison-Wesley.

[21] G. Jin, L. Song, X. Shi, J. Scherpelz & S. Lu (2012): *Understanding and detecting real-world performance bugs*. pp. 77–87.

[22] P.C. Jorgensen (1995): *Software Testing a Craftsman's Approach*. CRC Press.

[23] Mazhar Khaliq, Riyaz A Khan & MH Khan (2010): *Software quality measurement: A Revisit*. *Oriental Journal of Computer Science & Technology* 3(1), pp. 05–11.

[24] Huiqing Li, Simon Thompson, Pablo Lamela Seijas & Miguel Angel Francisco (2014): *Automating property-based testing of evolving web services*. In: *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation*, ACM, pp. 169–180.

[25] Pericles Loucopoulos & Vassilios Karakostas (1995): *System Requirements Engineering*. McGraw-Hill, Inc.

[26] M. Papadakis and K. Sagonas: *PropEr: A QuickCheck-Inspired Property-Based Testing Tool for Erlang*. `http://proper.softlab.ntua.gr`.

[27] Simone Meixler & Uwe Brinkschulte (2010): *Test case generation for non-functional and functional testing of services*. In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, IEEE, pp. 245–249.

[28] OMG: *Unified Modeling Language*. `http://www.uml.org`.

[29] Jane Radatz, Anne Geraci & Freny Katki (1990): *IEEE standard glossary of software engineering terminology*. *IEEE Std* 610121990, p. 121990.

[30] B. Ramesh & M. Jarke (2001): *Toward reference models for requirements traceability*. *IEEE Transactions on Software Engineering* 27(1), pp. 58–93.

[31] James Robertson (2006): *Mastering The Requirements Process, 2/E*. Pearson Education India.

[32] Ian Sommerville (2004): *Software Engineering. International computer science series*.

[33] Ian Sommerville & Gerald Kotonya (1998): *Requirements engineering: processes and techniques*. John Wiley & Sons, Inc.

[34] Richard H Thayer, Sidney C Bailin & M Dorfman (1997): *Software requirements engineerings*. IEEE Computer Society Press.

[35] S. Uchitel, J. Kramer & J. Magee (2003): *Synthesis of behavioral models from scenarios*. *IEEE Transactions on Software Engineering* 29(2), pp. 99–115.

[36] Joachim Wegener & Matthias Grochtmann (1998): *Verifying timing constraints of real-time systems by means of evolutionary testing*. *Real-Time Systems* 15(3), pp. 275–298.

[37] E.J. Weyuker (2000): *Experience with performance testing of software systems: issues, an approach, and case study*. *IEEE Transactions on Software Engineering* 26(12), pp. 1147–1156.

[38] Wikipedia: *Computer performance*. `http://en.wikipedia.org/wiki/Computer_performance`.

[39] Wikipedia: *Fibonacci number, divisibility properties*. `http://en.wikipedia.org/wiki/Fibonacci_number#Divisibility_properties`.

[40] Xusheng Xiao, Shi Han, Dongmei Zhang & Tao Xie (2013): *Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks*. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, ACM, pp. 90–100.

[41] Andreas Zeller & Ralf Hildebrandt (2002): *Simplifying and isolating failure-inducing input*. *Software Engineering, IEEE Transactions on* 28(2), pp. 183–200.

SACO: Static Analyzer for Concurrent Objects (High-level Work)*

Elvira Albert[1] Puri Arenas[1] Antonio Flores-Montoya[2] Samir Genaim[1]
Miguel Gómez-Zamalloa[1] Enrique Martin-Martin[1]
Germán Puebla[3] Guillermo Román-Díez[3]

[1] Complutense University of Madrid (UCM), Spain

[2] Technische Universität Darmstadt (TUD), Germany

[3] Technical University of Madrid (UPM), Spain

We present the main concepts, usage and implementation of SACO, a static analyzer for concurrent objects. Interestingly, SACO is able to infer both *liveness* (namely termination and resource boundedness) and *safety* properties (namely deadlock freedom) of programs based on concurrent objects. The system integrates auxiliary analyses such as *points-to* and *may-happen-in-parallel*, which are essential for increasing the accuracy of the aforementioned more complex properties. SACO provides accurate information about the dependencies which may introduce deadlocks, loops whose termination is not guaranteed, and upper bounds on the resource consumption of methods.

*Appeared in the Proceedings of the *20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'14). Springer, Lecture Notes in Computer Science Volume 8413, 2014, pp 562-567.

Submitted to:
PROLE 2014

SpecSatisfiabilityTool: A tool for testing the satisfiability of specifications on XML documents (Tool System)

Javier Albors Marisa Navarro
Departamento de LSI, UPV/EHU
San Sebastián, Spain
jalbors001@gmail.com marisa.navarro@ehu.es

We present a prototype that implements a set of logical rules to prove the satisfiability for a class of specifications on XML documents. Specifications are given by means of constrains built on Boolean XPath patterns. The main goal of this tool is to test whether a given specification is satisfiable or not, and justify the decision showing the execution history. It can also be used to test whether a given document is a model of a given specification and, as a subproduct, it permits to look for all the relations (monomorphisms) between two patterns and to combine patterns in different ways. The results of these operations are visually shown and therefore the tool makes these operations more understandable. The implementation of the algorithm has been written in Prolog but the prototype has a Java interface for an easy and friendly use. In this paper we show how to use this interface in order to test all the desired properties.

1 Introduction

Our aim is to define specifications of XML documents as sets of constraints (of some specific class) on these documents, and to provide a form of reasoning about these specifications. XML documents will be represented by trees and the constraints will be based on some kind of XPath queries [8, 2, 3].

To define the constraints on some XPath notation, we have selected the representation of Boolean XPath queries given in [5], where Miklau and Suciu study the containment and equivalence problems for a class of XPath queries that contain branching and label wildcards and can express descendant relationships between nodes. In particular, they introduce *Boolean patterns* as an alternative representation of this class of queries. These patterns are trees consisting of nodes with labels (or $*$, for a wildcard in the query) and two kind of edges, child edges (/) and descendant edges (//), for the corresponding axis in the query. For instance, the pattern p in Figure 1 (on the left) corresponds to the XPath expression $/a[b][.//*[c][d]]$. We define three sorts of constraints (positive, negative, and conditional constraints) on these patterns. A specification is defined as a set of clauses, where a clause is a disjunction of constraints.

Our main question is about satisfiability, that is, given a specification $\mathscr{S}$, whether or not there exists an XML document satisfying all constraints in $\mathscr{S}$. Moreover, we are looking for adequate inference rules to build a sound and complete refutation procedure for checking satisfiability of a given specification. In addition to checking satisfiability, these rules would be used to deduce other constraints, which can permit us to optimize a satisfiable specification.

Our approach follows the main ideas given in [7] (here it is shown how to use graph constraints as a specification formalism on graphs and how to reason about these specifications, providing refutation procedures based on inference rules that are sound and complete) and try to apply such ideas to XML documents. However, the particularization of graph constraints to our setting is not trivial (mainly because our patterns are more expressive). Similarly, our inference rules take a similar format to the inference rules given in [7], but the particularization to our setting needs to define appropriate operators

Submitted to:
PROLE 2014

and to prove new results. The formal study of our work is now submitted for presentation, but the ideas and preliminaries of such work were introduced in [6].

In this paper we present a prototype that implements our refutation procedure. The algorithm is written in Prolog [4] but it also has a Java interface for an easy and friendly use. The main goal of this tool is to test whether a given specification is satisfiable or not, and justify the decision by showing the rules applied during the procedure execution. It can also be used to test whether a given document is a model of a given specification and, as a subproduct, it permits to look for all the monomorphisms between two patterns or to look for the result of doing the operations $p \otimes q$ and $p \otimes_{c,m} q$ which are necessary for implementing some rules. The results of these operations are visually shown and therefore it makes them more understandable.

2 Formal Background

We consider an *XML document* as an unordered and unranked tree with nodes labelled from an infinite alphabet Σ. The symbols in Σ can represent the element labels, attribute labels, and text values that can occur in XML documents. As said in the introduction, we use patterns as an alternative representation of queries. Here are the formal definitions of documents and patterns.

Definition 2.1 *Given a signature Σ, a* document on Σ *is a tree t whose nodes are labelled with symbols from Σ and with one sort of edges denoted /. $Nodes(t)$ and $Edges(t)$ denote respectively the sets of nodes and edges in t; $Root(t)$ denotes its root node; and for each $n \in Nodes(t)$, $Label(n)$ denotes the label of such a node n. Each edge in $Edge(t)$ is represented (x,y) with $x,y \in Nodes(t)$. Each $(x,y) \in Edges^+(t)$ represents a path in t from node x to node y.*

Definition 2.2 *Given a signature Σ, a* pattern on Σ *is a tree p whose nodes are labelled with symbols from $\Sigma \cup \{*\}$ and with two sorts of edges: the descendant edges denoted //, and the child edges denoted /. $Nodes(p)$, $Edges(p)$, $Root(p)$, and $Label(n)$ are defined as in the previous definition; but now the edges are distinguished: $Edges(p) = Edges_{//}(p) \cup Edges_{/}(p)$, therefore $(x,y) \in Edges^+(p)$ represents a path in p from node x to node y with edges of type / or // along the path.*

Note that documents are patterns without labels $*$ or edges //. We define here the notion of homomorphism between two patterns (and therefore between a pattern and a document).

Definition 2.3 *Given two patterns p and q, a* homomorphism *from p to q is a function $h : Nodes(p) \to Nodes(q)$ satisfying the following conditions:*

- *Root-preserving: $h(Root(p)) = Root(q)$;*
- *Label-preserving: For each $n \in Nodes(p)$, $Label(n) = *$ or $Label(n) = Label(h(n))$;*
- *Child-edge-preserving: For each $(x,y) \in Edges_{/}(p)$, $(h(x),h(y)) \in Edges_{/}(q)$.*
- *Descendant-edge-preserving: For each $(x,y) \in Edges_{//}(p)$, $(h(x),h(y)) \in Edges^+(q)$.*

Definition 2.4 *Given a pattern p and a document t, we say that* t satisfies p*, denoted $t \models p$, if there exists a monomorphism (i.e, an injective homomorphism) from p into t. The model set of a pattern p is the set of documents satisfying p: $Mod(p) = \{t \mid t \models p\}$.*

In Figure 1 there is a pattern p (on the left), a document t (on the right) and a monomorphism $h : p \to t$ (which is drawn with dotted arrows). Then t satisfies (or is a model of) p. It corresponds to evaluate the XPath expression $/ = /a[b][.//*[c][d]]$ against the document t.

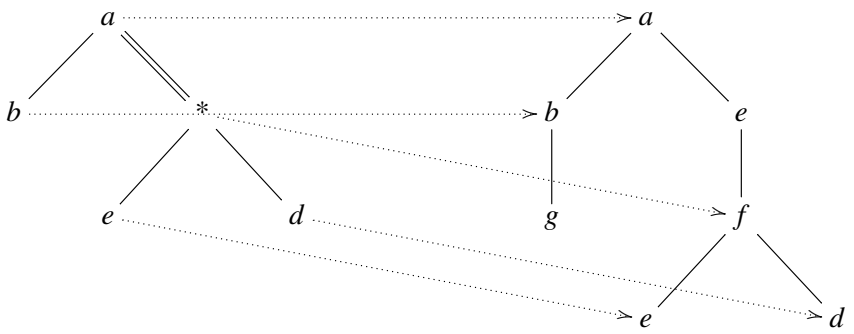


Figure 1: A monomorphism $h : p \to t$ from a pattern p to a document t

2.1 Specifications

We assume that a specification consists of a set of clauses, where a clause is a disjunction of constraints (the empty disjunction is the clause *FALSE*). Now we introduce the three kinds of constraints we are going to use. Then the satisfaction of clauses is defined inductively.

Definition 2.5 *Given a pattern p, $\exists p$ denotes a* positive constraint *and $\neg\exists p$ denotes a* negative constraint. *A* conditional constraint *is denoted $\forall(c : p \to q)$ where p and q are patterns, p is a prefix of q (seen both as trees) and $c : Nodes(p) \to Nodes(q)$ is such prefix relation.*

Definition 2.6 *A document t satisfies a clause α, denoted $t \models \alpha$, if it holds:*

- $t \models \exists p$ *if* $t \vDash p$ *(that is, if there exists a monomorphism $h : p \to t$);*
- $t \models \neg\exists p$ *if* $t \nvDash p$ *(that is, if there does not exist a monomorphism $h : p \to t$);*
- $t \models \forall(c : p \to q)$ *if for every monom. $h : p \to t$ there is a monom. $f : q \to t$ such that $h = f \circ c$.*
- $t \models L_1 \vee L_2 \vee \ldots \vee L_n$ *if* $t \models L_i$ *for some* $i \in \{1, \ldots, n\}$.

An example: The document t in Figure 1 (on the right) does not satisfy the constraint $\forall(c : p \to q)$ where p is the pattern corresponding to the XPath expression $/a[.//e]$ and q the pattern corresponding to the XPath expression $/a[.//e[f]]$, since not all descendant nodes labelled e in t have a child labelled f.

2.2 Inference Rules for a Refutation Procedure

A *refutation procedure* for a specification $\mathscr{S}$ can be seen as a sequence of inferences $\mathscr{C}_0 \Rightarrow \mathscr{C}_1 \Rightarrow \ldots \Rightarrow \mathscr{C}_i \Rightarrow \ldots$ where the initial state is the original specification (i.e., $\mathscr{C}_0 = \mathscr{S}$) and each $\mathscr{C}_{i+1}$ is obtained from $\mathscr{C}_i$ by applying a rule. The main inference rules of our refutation procedure are the following:

$$\frac{\exists p_1 \vee \Gamma_1 \quad \neg\exists p_2 \vee \Gamma_2}{\Gamma_1 \vee \Gamma_2} \quad \textbf{(R1)}$$

if there exists a monomorphism $m : p_2 \to p_1$

Rule (R1) is like a resolution rule, since the two premises have literals that are, in some sense, "complementary": one is a positive constraint, the other one is a negative one, and the condition about the

monomorphism from p_2 to p_1 plays the same role as unification. Note that when Γ_1 and Γ_2 are empty, the rule (R1) infers the clause *FALSE*.

$$\frac{\exists p_1 \vee \Gamma_1 \quad \exists p_2 \vee \Gamma_2}{(\bigvee_{s \in p_1 \otimes p_2} \exists s) \vee \Gamma_1 \vee \Gamma_2} \quad \textbf{(R2)}$$

Rule (R2) builds a disjunction of positive constraints from two positive constraints. It uses the operator $\otimes$ that we define below. Informally speaking, $p_1 \otimes p_2$ denotes the set of patterns that can be obtained by "combining" p_1 and p_2 in all possible ways.

$$\frac{\exists p_1 \vee \Gamma_1 \quad \forall (c : p_2 \to q) \vee \Gamma_2}{(\bigvee_{s \in p_1 \otimes_{c,m} q} \exists s) \vee \Gamma_1 \vee \Gamma_2} \quad \textbf{(R3)}$$

if there is a monomorphism $m : p_2 \to p_1$ that cannot be extended to $f : q \to p_1$ such that $f \circ c = m$.

Rule (R3) is similar to rule (R2): From a positive constraint $\exists p_1$ and a conditional constraint $\forall(c : p_2 \to q)$, it builds a disjunction of positive constraints. It uses the operator $\otimes_{c,m}$ that we define below. Informally speaking, $p_1 \otimes_{c,m} q$ denotes the set of patterns that can be obtained by combining p_1 and q in all possible ways, but maintaining p_2 shared.

Definition 2.7 *Given two patterns p_1 and p_2, $p_1 \otimes p_2$ is the following set of patterns: $p_1 \otimes p_2 = \{s \mid$ there exist jointly surjective monomorphisms $inc_1 : p_1 \to s$ and $inc_2 : p_2 \to s\}$.*

Definition 2.8 *Given two patterns p_1, p_2, a prefix function $c : p_2 \to q$, and a monomorphism $m : p_2 \to p_1$, $p_1 \otimes_{c,m} q$ is the following set of patterns: $p_1 \otimes_{c,m} q = \{s \mid$ there exist jointly surjective monomorphisms $inc_1 : p_1 \to s$ and $inc_2 : q \to s$ such that $inc_1 \circ m = inc_2 \circ c\}$.*

We have formally proven that the refutation procedure consisting of the three inference rules (R1), (R2), and (R3) is sound [6]. That is, whenever the procedure infers the clause *FALSE* from a input set of clauses $\mathscr{S}$, then $\mathscr{S}$ is unsatisfiable. The prototype we explain in the next section implements this refutation procedure when we choose to execute "Version 1". Moreover, the refutation procedure also uses sound rules for deleting and simplifying clauses and the implementation applies them as soon as possible to get a better performance. To resume, given a specification as input, if the result of running "Version 1" is that the procedure stops with *FALSE*, then we are sure that the specification is unsatisfiable.

However, the procedure is not complete: It may happen that the clause *FALSE* is not inferred although $\mathscr{S}$ is unsatisfiable (see [6]). Looking for a complete procedure, we have studied how to transform a positive constraint containing a descendant edge (//) into a (semantically equivalent) disjunction of positive constraints, in order to apply inference rules that could not be applied before such transformation. We call it "the unfolding process" and we have incorporated it to the refutation procedure. Lack of space in this paper does not permit us to give details of this study, but it can be found in [1]. The refutation procedure obtained by adding this "unfolding process" has been implemented and it can be tested running "Version 2" of the prototype. Although we are still working in a formal proof, we believe that the new procedure is complete. This means that if the input specification is unsatisfiable then the procedure stops and returns *FALSE*.

Finally, we must observe that for satisfiable specifications, the procedure can stop (without obtaining the clause *FALSE*) or not stop. We are studying the causes of non-termination and we think that our procedure does not stop only in the case of satisfiable specifications whose models are all infinite. Such specifications are possible due to the conditional constraints. If we restrict to specifications with only positive and negative constraints, the refutation procedure is finite.

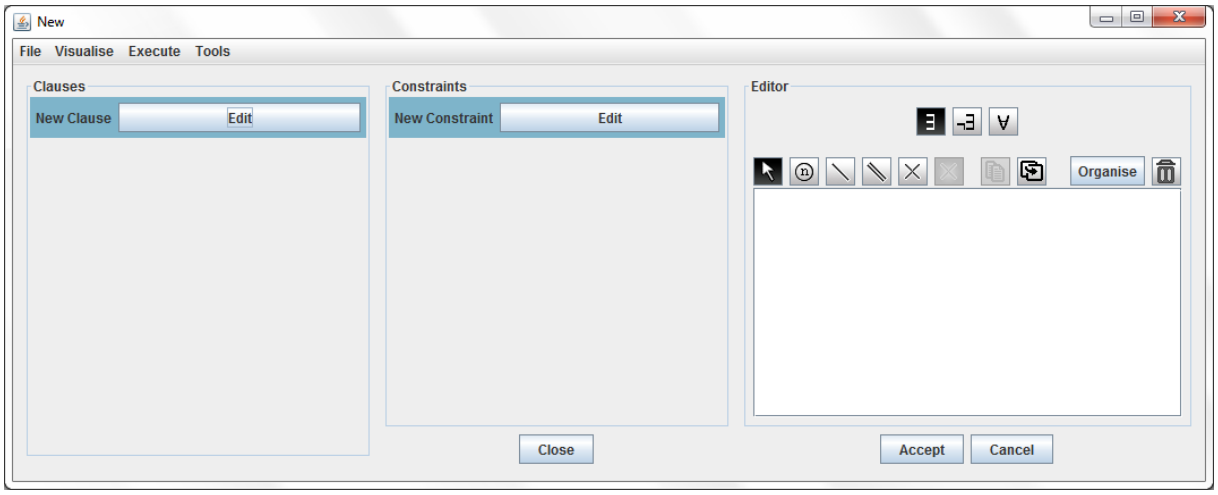


Figure 2: Home screen of the application

3 Showing the Prototype

In this section we explain how to use the application. In particular, introducing the clauses, executing the refutation procedure, testing whether a document is a model of a specification, and other operations.

3.1 Introducing Clauses

The Home screen of the application consists of three different panels, besides the menu bar: the clauses' panel, the constraints' panel, and the pattern editor. In order to create a clause, click on the "Edit" button of "New Clause". After that, the constraints' panel will show the selected clause's constraints. Since the clause is new, it will only appear the option of creating a new constraint. By clicking on the "Edit" button of "New Constraint", the pattern editor will be shown, as it can be noticed in Figure 2.

The type of constraint ($\exists, \neg\exists, \forall$) is indicated by clicking on one of the upper buttons. Then, build the pattern by using the nodes, children edges (/), and descendant edges (//) creation buttons. If the selected constraint is conditional, $\forall(c : p \rightarrow q)$, the editor screen will change into the one in Figure 3, where p must be drawn on the upper left box and q on the upper right box. Once they are set, click on the "Generate pre-tree" button and the system will find all the prefix relations that exist between the two patterns. Click on the arrow-form buttons to choose the correct relation and click on "Accept".

It is also possible to load an existing specification (that was previously saved) by selecting the option "Open" from the "File" menu.

3.2 Executing the Refutation Procedure

Once all the clauses have been created, let us see how to execute the refutation procedure. To do so, pick one of the two versions from the "Execute" menu, as it is shown in Figure 4. After finishing the procedure, a message is displayed (see Figure 5) in which appears the satisfiability result of the input specification and the elapsed time. The execution history will automatically be opened in other window.

In this new screen (shown in Figure 6), besides the history, it is possible to consult clauses and constraints. Enter the identifier of a clause (e.g. c4) in the clause searching area and it will be shown. The constraint search works exactly like the clause search, but with constraint identifiers (e.g. ct1). The

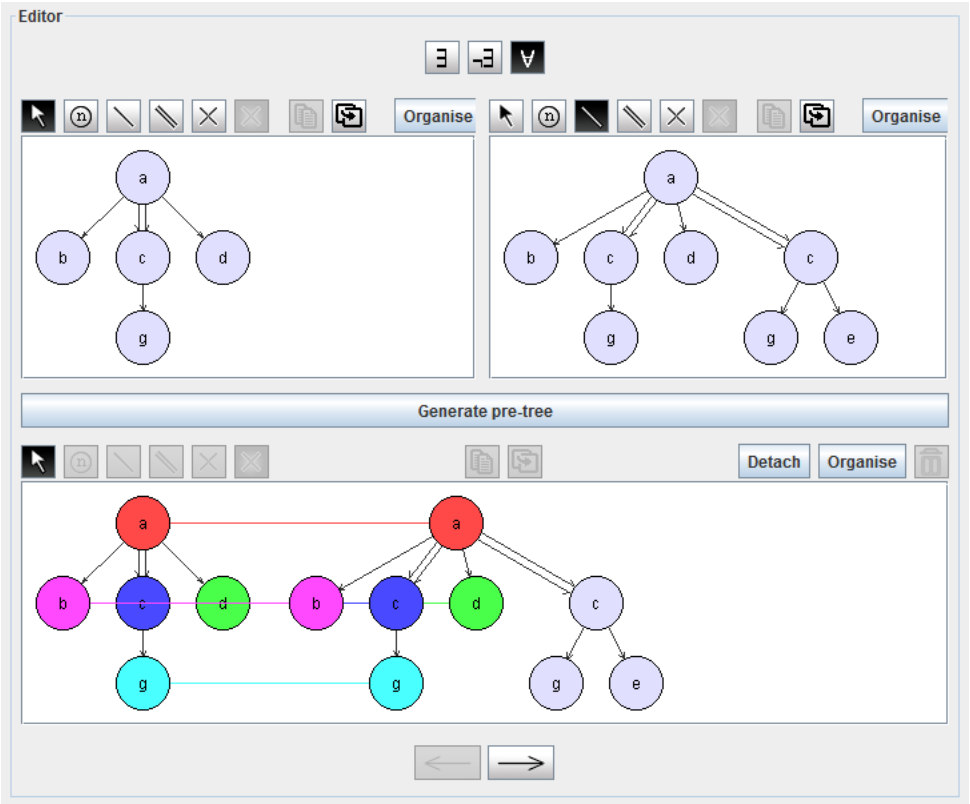


Figure 3: Editor for conditional constraints

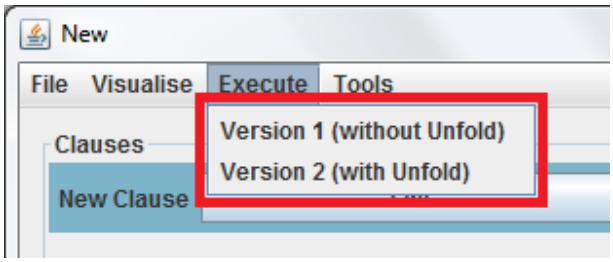


Figure 4: Menu for executing the procedure

Figure 5: The result of the procedure

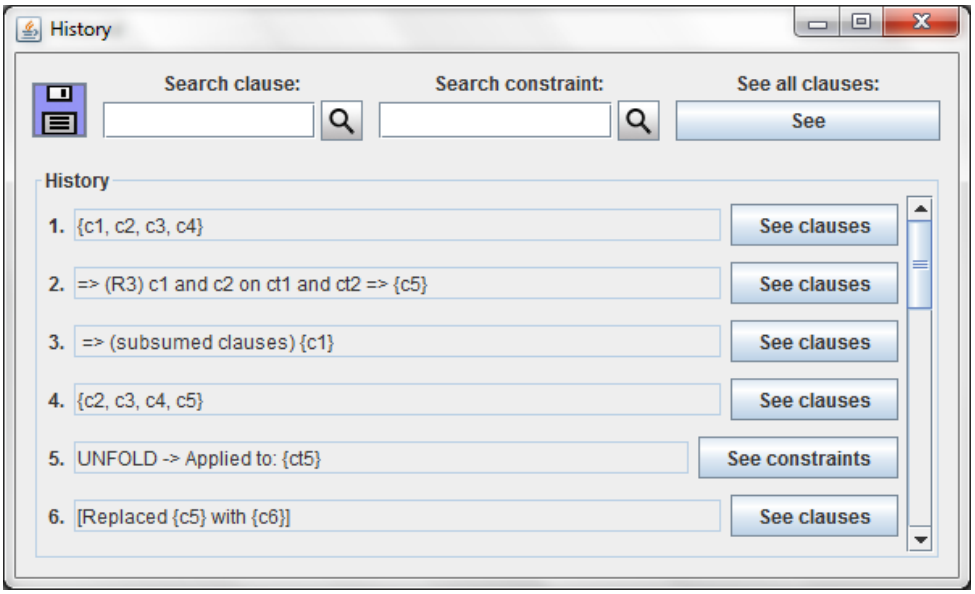


Figure 6: History of the procedure

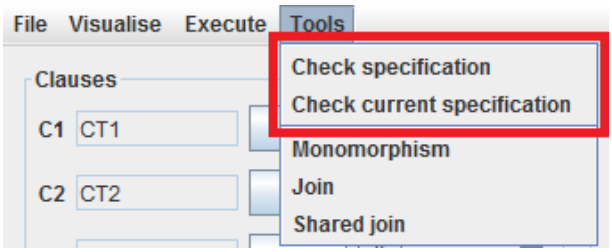


Figure 7: Document checking option

upper right button loads every existing clause and displays them. Also, with the “See clauses" buttons, it is possible to consult specific clauses from the different steps of the history. For instance, if we click on the button of the second step in Figure 6, the system will load the clauses c1, c2, and c5. Finally, to export the history to a text file, click on the save button on the upper left corner.

3.3 Document Checking

Another important aim of this application is to check whether a given document satisfies a given specification or not. For that, click on “Check specification" in the “Tools" menu (see Figure 7). This operation will open a window, similar to the Home screen, where the clauses of the specification and the XML document will be introduced. The application also allows one to copy the set of clauses from the Home screen to this window. For that, click on “Check current specification" in the “Tools" menu (see Figure 7). After being copied, new clauses can be introduced or existing ones can be deleted without compromising the original ones. In this case, the XML document must be introduced too.

After introducing the XML document and once loaded the specification by any of the two possible ways, click on the “Check" button and a message with the result will be shown. The message will be *TRUE* when the document is a model of the specification, and *FALSE* otherwise.

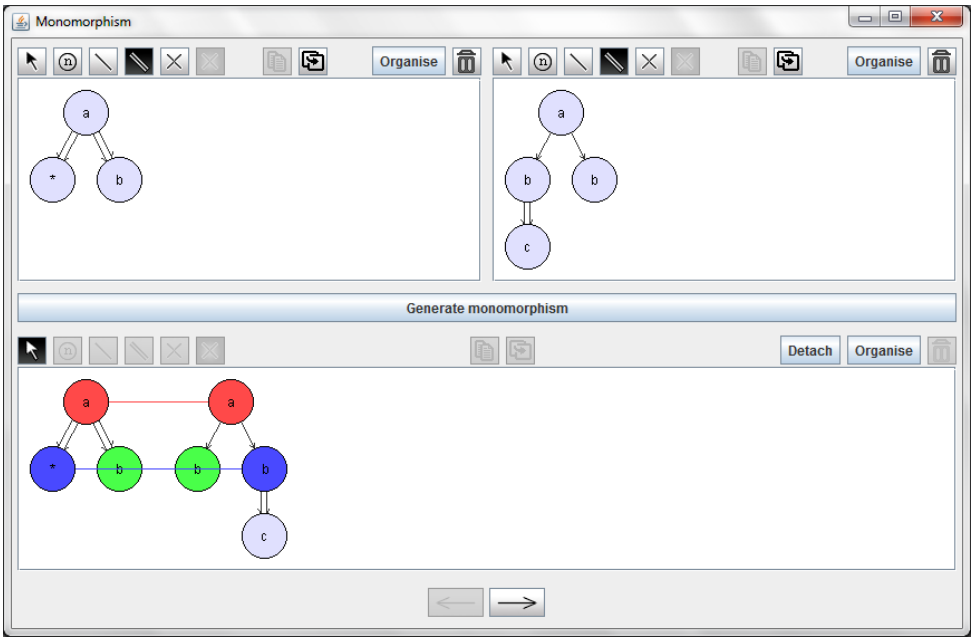


Figure 8: Monomorphism

3.4 Other Tools

Throughout the refutation process three operations are used: monomorphism from p to q, $p_1 \otimes p_2$, and $p_1 \otimes_{c,m} q$. The application includes tools to execute such operations visually, called respectively Monomorphism, Join, and Shared join.

3.4.1 Monomorphism

When selecting “Monomorphism" from the “Tools" menu, a new screen will appear, very similar to the one for creating a conditional constraint. Provided that we want to find out whether there exists a monomorphism from p to q, we introduce the pattern p into the upper left box and the pattern q into the right box. Then, click on “Generate monomorphism" and the system will find every possible solution. The different solutions can be consulted by clicking on the arrow-form buttons (see Figure 8).

3.4.2 Join operation ($p_1 \otimes p_2$)

The “Join" tool will also open a similar window to the conditional constraint screen. We will introduce the patterns we want to operate, p_1 and p_2, into the two upper editors. After that, we click on the “Join" button and the solution will be calculated. On the lower editor will appear a set of patterns $s_1, s_2, \ldots, s_n$ which express the different ways of “combining" p_1 and p_2 (see Figure 9). Recall this operation is used in rule (R2) to obtain $(\bigvee_{s \in p_1 \otimes p_2} \exists s)$ from $\exists p_1$ and $\exists p_2$.

3.4.3 Shared join operation ($p_1 \otimes_{c,m} q$)

Similarly, the operation $p_1 \otimes_{c,m} q$ is used in rule (R3) to obtain $(\bigvee_{s \in p_1 \otimes_{c,m} q} \exists s)$ from the constraints $\exists p_1$ and $\forall (c : p_2 \to q)$. Since one of them is a conditional constraint, the tool is comprised by two windows.

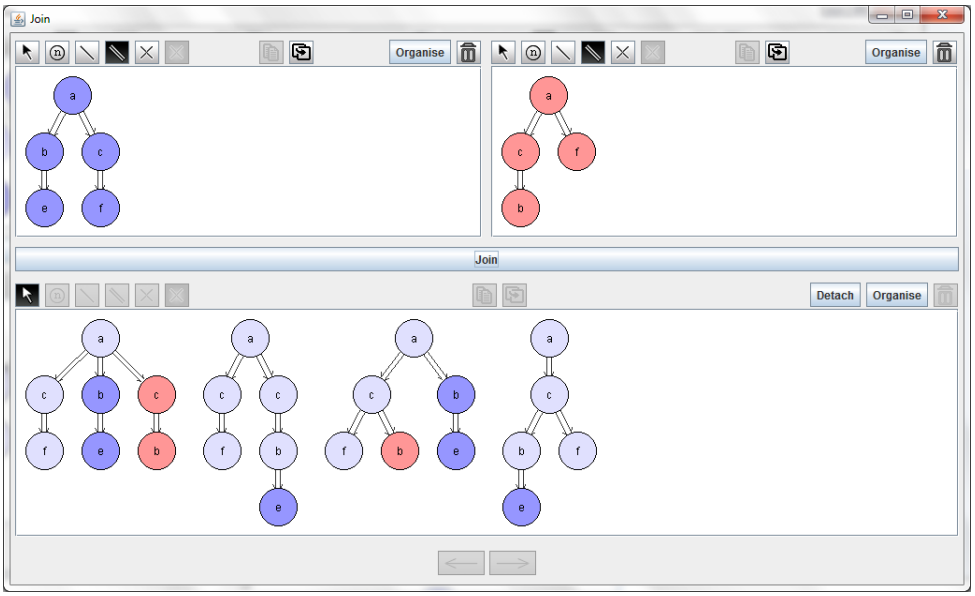


Figure 9: $p_1 \otimes p_2$

In the first one, we will introduce the conditional constraint as shown in Figure 3 and, after clicking on the “Next" button, this conditional constraint appears on the upper left box of the second window (see Figure 10); whereas the positive constraint is introduced into the upper right box. Then, we click on “Shared join". Due to the possibility of having more than one monomorphism from p_2 to p_1 (that cannot be extended to a monomorphism from q to p_1), different solutions will be shown. We can change the solution by clicking on the arrow-form buttons.

4 Implementation Notes

The prototype implementing the previously described refutation procedure is available at `http://www.sc.ehu.es/jiwnagom/PaginaWebLorea/SpecSatisfiabilityTool.html`, where we also explain the application’s requirements and the configuration of the Java-Prolog bridge. The code of this application consists of around 1300 Prolog lines (in SWI-Prolog version 6.0.2) for the refutation procedure and around 4000 Java lines (in Java version jre7) for the interface.

Now, we roughly explain the algorithms designed in each version of the refutation procedure. See [1] for more details about the implementation or for a user guide of the application.

4.1 Version 1 algorithm

We give here the idea of the algorithm implementing this refutation procedure. We start with the initial specification S_0. Clause by clause and constraint by constraint the procedure applies every possible rule (R1), (R2), or (R3) obtaining a set S'_0 of new clauses. Now the system divides the application of the rules in two parts: first, it applies every possible rule between two clauses, but being one from S_0 and the other one from the new set S'_0. After finishing this part, it applies every possible rule among the clauses in S'_0. In this way, all the clauses (resolvents) produced by applying these rules on $S_1 = S_0 \cup S'_0$ are in a new set S'_1. This process will be repeated until the clause *FALSE* comes out or until no rule can be applied. As

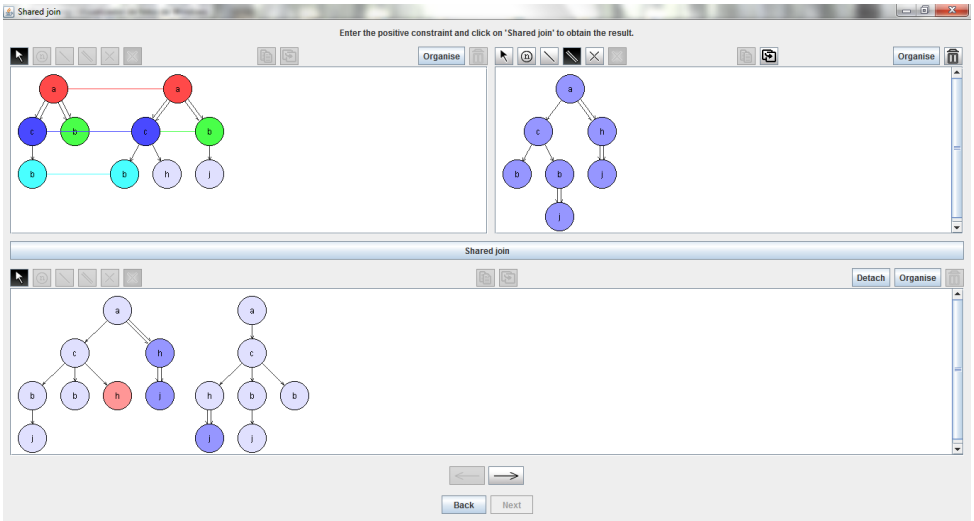


Figure 10: $p_1 \otimes_{c,m} q$

said above, other rules for deleting and simplifying clauses are also applied (as soon as possible) in order to get a better performance.

4.2 Version 2 algorithm

This version is an ongoing work. The algorithm is as follows: It starts by calling to the Version 1. Then, if the result returns the clause *FALSE*, it finishes (since it has been proven that the input specification is unsatisfiable). If Version 1 finishes returning *TRUE*, then the "unfolding process" is done. If this process does not obtain new clauses, the algorithm finishes with the result of *TRUE*, meaning that the input specification is satisfiable. But if the "unfolding process" obtains a new set of clauses, then the whole procedure is repeated with this new set of clauses as input specification.

References

[1] Albors, J. *Procedimiento de refutación para un conjunto de restricciones sobre documentos XML*, Proyecto Fin de Carrera, Facultad de Informática, San Sebastián, Julio 2014. (in Spanish). Available at `http://www.sc.ehu.es/jiwnagom/PaginaWebLorea/Javier_Albors_PFC.pdf`

[2] Benedikt, M., and Koch, C. *XPath Leashed*, ACM Computing Surveys, Vol 41, N.1, Article 3 (2008).

[3] Benedikt, M., Fan, W., and Geerts, F. *XPath satisfiability in the presence of DTDs*. Journal of the ACM 55, N. 2 (2008).

[4] Clocksin, W. F., Mellish, C.S. Programming in Prolog. Springer-Verlag (2003).

[5] Miklau, G., and Suciu, D. *Containment and equivalence for a fragment of XPath*, Journal of the ACM, Vol. 51, N.1, (2004) 2-45.

[6] Navarro, M., and Orejas, F. *Proving Satisfiability of Constraint Specifications on XML Documents*, PROLE2010, Valencia. `http://www.sc.ehu.es/jiwnagom/PaginaWebLorea/cedi.pdf`

[7] Orejas, F., Ehrig, H., and Prange, U. *A Logic of Graph Constraints*, Fundamental Approaches to Software Engineering, 11th Int. Conference, FASE 2008. LNCS 4961 (2008) 179-198.

[8] WORLD WIDE WEB CONSORTIUM. 2007. *XML path language (XPath) 2.0.*

A Fuzzy Logic Programming Environment for Managing Similarity and Truth Degrees (Tool System)*

Pascual Julián-Iranzo
Department of Technologies and Information Systems
University of Castilla-La Mancha
13071 Ciudad Real (Spain)
Pascual.Julian@uclm.es

Ginés Moreno
Department of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)
Gines.Moreno@uclm.es

Jaime Penabad
Department of Mathematics
University of Castilla-La Mancha
02071 Albacete (Spain)
Jaime.Penabad@uclm.es

Carlos Vázquez
Department of Computing Systems
University of Castilla-La Mancha
02071 Albacete (Spain)
Carlos.Vazquez@uclm.es

FASILL (acronym of "Fuzzy Aggregators and Similarity Into a Logic Language") is a fuzzy logic programming language with implicit/explicit truth degree annotations, a great variety of connectives and unification by similarity. FASILL integrates and extends features coming from MALP (*Multi-Adjoint Logic Programming*, a fuzzy logic language with explicitly annotated rules) and Bousi∼Prolog (which uses a weak unification algorithm and is well suited for flexible query answering). Hence, it properly manages similarity and truth degrees in a single framework combining the expressive benefits of both languages. This paper presents the main features and implementations details of FASILL. Along the paper we describe its syntax and operational semantics and we give clues of the implementation of the lattice module and the similarity module, two of the main building blocks of the new programming environment which enriches the FLOPER system developed in our research group.

Keywords: Fuzzy Logic Programming, Similarity Relations, Software Tools

1 Introduction

The challenging research area of *Fuzzy Logic Programming* is devoted to introduce *fuzzy logic* concepts into *logic programming* in order to explicitly treat with uncertainty in a natural way. It has provided a wide variety of PROLOG dialects along the last three decades. *Fuzzy logic languages* can be classified (among other criteria) regarding the emphasis they assign when fuzzifying the original unification/resolution mechanisms of PROLOG. So, whereas some approaches are able to cope with similarity/proximity relations at unification time [3, 2, 16], other ones extend their operational principles (maintaining syntactic unification) for managing a wide variety of fuzzy connectives and truth degrees on rules/goals beyond the simpler case of *true* or *false* [7, 8, 13]. Our research group has been involved in both alternatives, as reveals the design of the Bousi∼Prolog language[1] [5, 6, 15], where clauses cohabit with similarity/proximity equations, and the development of the FLOPER system[2], which manages fuzzy programs composed by rules richer than clauses [9, 12]. Our current goal for fusing both worlds

*This work was supported by the EU (FEDER), and the Spanish MINECO Ministry (*Ministerio de Economía y Competitividad*) under grant TIN2013-45732-C4-2-P.

[1]Two different programming environments for Bousi∼Prolog are available at http://dectau.uclm.es/bousi/.

[2]The tool is freely accessible from the Web site http://dectau.uclm.es/floper/.

$\dot{\&}_{\mathtt{P}}(x,y) \triangleq x * y$	$\dot{\mid}_{\mathtt{P}}(x,y) \triangleq x+y-xy$	*Product*
$\dot{\&}_{\mathtt{G}}(x,y) \triangleq \min(x,y)$	$\dot{\mid}_{\mathtt{G}}(x,y) \triangleq max(x,y)$	*Gödel*
$\dot{\&}_{\mathtt{L}}(x,y) \triangleq \max(0,x+y-1)$	$\dot{\mid}_{\mathtt{L}}(x,y) \triangleq \min(x+y,1)$	*Łukasiewicz*

Figure 1: Conjunctions and disjunctions in $[0,1]$ for *Product*, *Łukasiewicz*, and *Gödel* fuzzy logics

is somehow inspired by [1], but in our framework we admit a wider set of connectives inside the body of programs rules. In this paper, we give a first step in our pending task from some years ago for embedding into FLOPER the *weak unification* algorithm of Bousi∼Prolog.

FASILL is a first order language built upon a signature Σ, that contains the elements of a countably infinite set of variables $\mathscr{V}$, function symbols and predicate symbols with an associated arity –usually expressed as pairs f/n or p/n where n represents its arity–, the implication symbol ($\leftarrow$) and a set of connectives. The language combines the elements of Σ as terms, atoms, rules and formulas. A *constant* c is a function symbol with arity zero. A *term* is a variable, a constant or a function symbol f/n applied to n terms $t_1,\ldots,t_n$, and is denoted as $f(t_1,\ldots,t_n)$. We allow values of a lattice L as part of the signature Σ. Therefore, a well-formed formula can be either:

- r, if $r \in L$
- $p(t_1,\ldots,t_n)$, if $t_1,\ldots,t_n$ are terms and p/n is an n-ary predicate. This formula is called *atom*. Particularly, atoms containing no variables are called *ground atoms*, and atoms built from nullary predicates are called *propositional variables*
- $\varsigma(\mathscr{F}_1,\ldots,\mathscr{F}_n)$, if $\mathscr{F}_1,\ldots,\mathscr{F}_n$ are well-formed formulas and ς is an n-ary connective with truth function $\dot{\varsigma} : L^n \to L$

Definition 1.1 (Complete lattice)**.** *A complete lattice is a partially ordered set $(L,\leq)$ such that every subset S of L has infimum and supremum elements. Then, it is a bounded lattice, i.e., it has bottom and top elements, denoted by $\bot$ and $\top$, respectively. L is said to be the carrier set of the lattice, and $\leq$ its ordering relation.*

The lattice is equipped with a set of *connectives*[3] including

- aggregators denoted by @, whose truth functions $\dot{@}$ fulfill the boundary condition: $\dot{@}(\top,\top) = \top$, $\dot{@}(\bot,\bot) = \bot$, and monotonicity: $(x_1,y_1) \leq (x_2,y_2) \Rightarrow \dot{@}(x_1,y_1) \leq \dot{@}(x_2,y_2)$.
- t-norms and t-conorms [14] (also named conjunctions and disjunctions, that we denote by & and |, respectively) whose truth functions fulfill the following properties:

· Commutative:	$\dot{\&}(x,y) = \dot{\&}(y,x)$	$\dot{\mid}(x,y) = \dot{\mid}(y,x)$
· Associative:	$\dot{\&}(x,\dot{\&}(y,z)) = \dot{\&}(\dot{\&}(x,y),z)$	$\dot{\mid}(x,\dot{\mid}(y,z)) = \dot{\mid}(\dot{\mid}(x,y),z)$
· Identity element:	$\dot{\&}(x,\top) = x$	$\dot{\mid}(x,\bot) = x$
· Monotonicity in each argument: $z \leq t \Rightarrow$	$\dot{\&}(z,y) \leq \dot{\&}(t,y)$	$\dot{\&}(x,z) \leq \dot{\&}(x,t)$
	$\dot{\mid}(z,y) \leq \dot{\mid}(t,y)$	$\dot{\mid}(x,z) \leq \dot{\mid}(x,t)$

In this paper we use the lattice $([0,1],\leq)$, where $\leq$ is the usual ordering relation on real numbers, and three sets of connectives corresponding to the fuzzy logics of Gödel, Łukasiewicz and Product, defined in Figure 1, where labels `L`, `G` and `P` mean respectively *Łukasiewicz logic*, *Gödel logic* and *product logic* (with different capabilities for modeling *pessimistic*, *optimistic* and *realistic scenarios*.)

[3]Here, the connectives are binary operations but we usually generalize them with an arbitrary number of arguments.

Definition 1.2 (Similarity relation)**.** *Given a domain $\mathscr{U}$ and a lattice L with fixed t-norm $\wedge$, a similarity relation $\mathscr{R}$ is a fuzzy binary relation on $\mathscr{U}$, that is a fuzzy subset on $\mathscr{U}\times\mathscr{U}$ (namely, a mapping $\mathscr{R}:\mathscr{U}\times\mathscr{U}\to L$), such that fulfills the following properties*[4]*:*

- *Reflexive:* $\mathscr{R}(x,x)=\top,\forall x\in\mathscr{U}$
- *Symmetric:* $\mathscr{R}(x,y)=\mathscr{R}(y,x),\forall x,y\in\mathscr{U}$
- *Transitive:* $\mathscr{R}(x,z)\geq\mathscr{R}(x,y)\wedge\mathscr{R}(y,z),\forall x,y,z\in\mathscr{U}$

Certainly, we are interested in fuzzy binary relations on a syntactic domain. We primarily define similarities on the symbols of a signature, Σ, of a first order language. This makes possible to treat as indistinguishable two syntactic symbols which are related by a similarity relation $\mathscr{R}$. Moreover, a similarity relation $\mathscr{R}$ on the alphabet of a first order language can be extended to terms by structural induction in the usual way [16]:

1. let x be a variable, $\hat{\mathscr{R}}(x,x)=\mathscr{R}(x,x)=1$,
2. let f and g be two n-ary function symbols and let $t_1,\ldots,t_n$, $s_1,\ldots,s_n$ be terms,
   $\hat{\mathscr{R}}(f(t_1,\ldots,t_n),g(s_1,\ldots,s_n))=\mathscr{R}(f,g)\wedge(\bigwedge_{i=1}^{n}\hat{\mathscr{R}}(t_i,s_i))$
3. otherwise, the approximation degree of two terms is zero.

Analogously for atomic formulas. Note that, following on, we shall not make a notational distintion between the relation $\mathscr{R}$ and its extension $\hat{\mathscr{R}}$.

Definition 1.3 (Rule)**.** *A rule has the form $A\leftarrow\mathscr{B}$, where A is an atomic formula called head and $\mathscr{B}$, called body, is a well-formed formula (ultimately built from atomic formulas $B_1,\ldots,B_n$, truth values of L and connectives)* [5]*. In particular, when the body of a rule is $r\in L$ (an element of lattice L), this rule is called fact and can be written as $A\leftarrow r$ (or simply A if $r=\top$).*

Definition 1.4 (Program)**.** *A program $\mathscr{P}$ is a tuple $\langle\Pi,\mathscr{R},L\rangle$ where Π is a set of rules, $\mathscr{R}$ is a similarity relation between the elements of Σ, and L is a complete lattice.*

2 Operational Semantics of FASILL

Rules in a FASILL program have the same role than clauses in PROLOG (or MALP [8, 4, 11]) programs, that is, stating that a certain predicate relates some terms (the *head*) if some conditions (the *body*) hold.

As a logic language, FASILL inherits the concepts of substitution, unifier and most general unifier (*mgu*). Some of them are extended to cope with similarities. Concretely, the most general unifier is replaced by the concept of *weak most general unifier* (w.m.g.u.), following the line of Bousi~Prolog [5]. Roughly speaking, the *weak unification algorithm* states that two *expressions* (i.e, terms or atomic formulas) $f(t_1,\ldots,t_n)$ and $g(s_1,\ldots,s_n)$ weakly unify if the root symbols f and g are close with a certain degree (i.e. $\mathscr{R}(f,g)=r>\bot$) and each of their arguments t_i and s_i weakly unify. Therefore, there is a weak unifier for two expressions even if the symbols at their roots are not syntactically equals ($f\not\equiv g$).

More technically, the weak unification algorithm we are using is a reformulation/extension of the one which appears in [16] for arbitrary complete lattices. We formalize it as a transition system supported by a similarity-based unification relation "$\Rightarrow$". The unification of the expressions $\mathscr{E}_1$ and $\mathscr{E}_2$ is

[4] For convenience, $\mathscr{R}(x,y)$, also denoted $x\mathscr{R}y$, refers both the syntactic expression (that symbolizes that the elements $x,y\in\mathscr{U}$ are related by $\mathscr{R}$) and the truth degree $\mu_{\mathscr{R}}(x,y)$, i.e., the affinity degree of the pair $(x,y)\in\mathscr{U}\times\mathscr{U}$ with the verbal predicate $\mathscr{R}$.

[5] In order to subsume the syntactic conventions of MALP, in our programs we also admit *weighted rules* with shape "$A\leftarrow_i\mathscr{B}$ *with* v", which are internally treated as "$A\leftarrow(v\&_i\mathscr{B})$" (this transformation preserves the meaning of rules as proved in [10]).

obtained by a state transformation sequence starting from an initial state $\langle G \equiv \{\mathscr{E}_1 \approx \mathscr{E}_2\}, id, \alpha_0\rangle$, where id is the identity substitution and $\alpha_0 = \top$ is the supreme of $(L, \leq)$: $\langle G, id, \alpha_0\rangle \Rightarrow \langle G1, \theta_1, \alpha_1\rangle \Rightarrow \cdots \Rightarrow \langle G_n, \theta_n, \alpha_n\rangle$. When the final state $\langle G_n, \theta_n, \alpha_n\rangle$, with $G_n = \emptyset$, is reached (i.e., the equations in the initial state have been solved), the expressions $\mathscr{E}_1$ and $\mathscr{E}_2$ are unifiable by similarity with w.m.g.u. θ_n and *unification degree* α_n. Therefore, the final state $\langle \emptyset, \theta_n, \alpha_n\rangle$ signals out the unification success. On the other hand, when expressions $\mathscr{E}_1$ and $\mathscr{E}_2$ are not unifiable, the state transformation sequence ends with failure (i.e., $G_n = Fail$).

The *similarity-based unification relation*, "$\Rightarrow$", is defined as the smallest relation derived by the following set of transition rules (where $\mathscr{V}ar(t)$ denotes the set of variables of a given term t)

$$\frac{\langle \{f(t_1,\ldots,t_n) \approx g(s_1,\ldots,s_n)\} \cup E, \theta, r_1\rangle \qquad \mathscr{R}(f,g) = r_2 > \bot}{\langle \{t_1 \approx s_1,\ldots,t_n \approx s_n\} \cup E, \theta, r_1 \wedge r_2\rangle}\ 1$$

$$\frac{\langle \{X \approx X\} \cup E, \theta, r_1\rangle}{\langle E, \theta, r_1\rangle}\ 2 \qquad \frac{\langle \{X \approx t\} \cup E, \theta, r_1\rangle \qquad X \notin \mathscr{V}ar(t)}{\langle (E)\{X/t\}, \theta\{X/t\}, r_1\rangle}\ 3$$

$$\frac{\langle \{t \approx X\} \cup E, \theta, r_1\rangle}{\langle \{X \approx t\} \cup E, \theta, r_1\rangle}\ 4 \qquad \frac{\langle \{X \approx t\} \cup E, \theta, r_1\rangle \qquad X \in \mathscr{V}ar(t)}{\langle Fail, \theta, r_1\rangle}\ 5$$

$$\frac{\langle \{f(t_1,\ldots,t_n) \approx g(s_1,\ldots,s_n)\} \cup E, \theta, r_1\rangle \qquad \mathscr{R}(f,g) = \bot}{\langle Fail, \theta, r_1\rangle}\ 6$$

Rule 1 decomposes two expressions and annotates the relation between the function (or predicate) symbols at their root. The second rule eliminates spurious information and the fourth rule interchanges the position of the symbols to be coped by other rules. The third and fifth rules perform an occur check of variable X in a term t. In case of success, it generates a substitution $\{X/t\}$; otherwise the algorithm ends with failure. It can also end with failure if the relation between function (or predicate) symbols in $\mathscr{R}$ is $\bot$, as stated by Rule 6.

Usually, given two expressions $\mathscr{E}_1$ and $\mathscr{E}_2$, if there is a successful transition sequence, $\langle \{\mathscr{E}_1 \approx \mathscr{E}_2\}, id, \top\rangle \Rightarrow^\star \langle \emptyset, \theta, r\rangle$, then we write that $wmgu(\mathscr{E}_1, \mathscr{E}_2) = \langle \theta, r\rangle$, being θ the *weak most general unifier* of $\mathscr{E}_1$ and $\mathscr{E}_2$, and r is their *unification degree*.

Finally note that, in general, a w.m.g.u. of two expressions $\mathscr{E}_1$ and $\mathscr{E}_2$ is not unique [16]. Certainly, the weak unification algorithm only computes a representative of a w.m.g.u. class, in the sense that, if $\theta = \{x_1/t_1,\ldots,x_n/t_n\}$ is a w.m.g.u., with degree β, then, by definition, any substitution $\theta' = \{x_1/s_1,\ldots,x_n/s_n\}$, satisfying $\mathscr{R}(s_i,t_i) > \bot$, for any $1 \leq i \leq n$, is also a w.m.g.u. with approximation degree $\beta' = \beta \wedge (\bigwedge_1^n \mathscr{R}(s_i,t_i))$, where "$\wedge$" is a selected t-norm. However, observe that, the w.m.g.u. representative computed by the weak unification algorithm is one with an approximation degree equal or greater than other w.m.g.u. As in the case of the classical syntactic unification algorithm, our algorithm always terminates returning a success or a failure.

In order to describe the procedural semantics of the FASILL language, in the following we denote by $\mathscr{C}[A]$ a formula where A is a sub-expression (usually an atom) which occurs in the –possibly empty– context $\mathscr{C}[]$ whereas $\mathscr{C}[A/A']$ means the replacement of A by A' in the context $\mathscr{C}[]$. Moreover, $\mathscr{V}ar(s)$ denotes the set of distinct variables occurring in the syntactic object s and $\theta[\mathscr{V}ar(s)]$ refers to the substitution obtained from θ by restricting its domain to $\mathscr{V}ar(s)$. In the next definition, we always consider that A is the selected atom in a goal $\mathscr{Q}$ and L is the complete lattice associated to Π.

Definition 2.1 (Computational Step)**.** *Let $\mathscr{Q}$ be a goal and let σ be a substitution. The pair $\langle \mathscr{Q}; \sigma\rangle$ is a* state*. Given a program $\langle \Pi, \mathscr{R}, L\rangle$ and a t-norm $\wedge$ in L, a* computation *is formalized as a state transition*

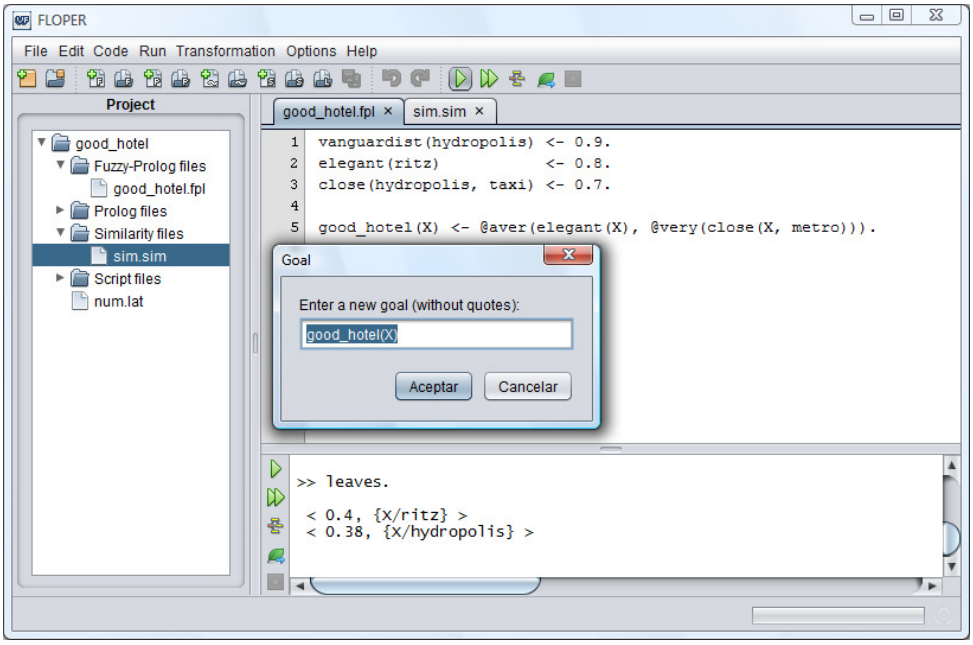


Figure 2: Screen-shot of a work session with FLOPER managing a FASILL program

system, whose transition relation $\rightsquigarrow$ *is the smallest relation satisfying these rules:*

1) Successful step *(denoted as* $\overset{SS}{\rightsquigarrow}$*):*

$$\frac{\langle \mathscr{Q}[A], \sigma \rangle \qquad A' \leftarrow \mathscr{B} \in \Pi \qquad wmgu(A, A') = \langle \theta, r \rangle}{\langle \mathscr{Q}[A/\mathscr{B} \wedge r]\theta, \sigma\theta \rangle} \text{ SS}$$

2) Failure step *(denoted as* $\overset{FS}{\rightsquigarrow}$*):*

$$\frac{\langle \mathscr{Q}[A], \sigma \rangle \qquad \nexists A' \leftarrow \mathscr{B} \in \Pi : wmgu(A, A') = \langle \theta, r \rangle, r > \bot}{\langle \mathscr{Q}[A/\bot], \sigma \rangle} \text{ FS}$$

3) Interpretive step *(denoted as* $\overset{IS}{\rightsquigarrow}$*):*

$$\frac{\langle \mathscr{Q}[@(r_1, \ldots, r_n)]; \sigma \rangle \qquad \dot{@}(r_1, \ldots, r_n) = r_{n+1}}{\langle \mathscr{Q}[@(r_1, \ldots, r_n)/r_{n+1}]; \sigma \rangle} \text{ IS}$$

A *derivation* is a sequence of arbitrary lenght $\langle \mathscr{Q}; id \rangle \rightsquigarrow^* \langle \mathscr{Q}'; \sigma \rangle$. As usual, rules are renamed apart. When $\mathscr{Q}' = r \in L$, the state $\langle r; \sigma \rangle$ is called a *fuzzy computed answer* (f.c.a.) for that derivation.

3 Implementation of FASILL in FLOPER

During the last years we have developed the FLOPER tool, initially intended for manipulating MALP programs[6]. In its current development state, FLOPER has been equipped with new features in order to

[6] The MALP language is nowadays fully subsumed by the new FASILL language just introduced in this paper, since, given a FASILL program $\mathscr{P} = \langle \Pi, \mathscr{R}, L \rangle$, if $\mathscr{R}$ is the identity relation (that is, the one where each element of a signature Σ is only similar to itself, with the maximum similarity degree) and L is a complete lattice also containing *adjoint pairs* [8], then $\mathscr{P}$ is a MALP program too.

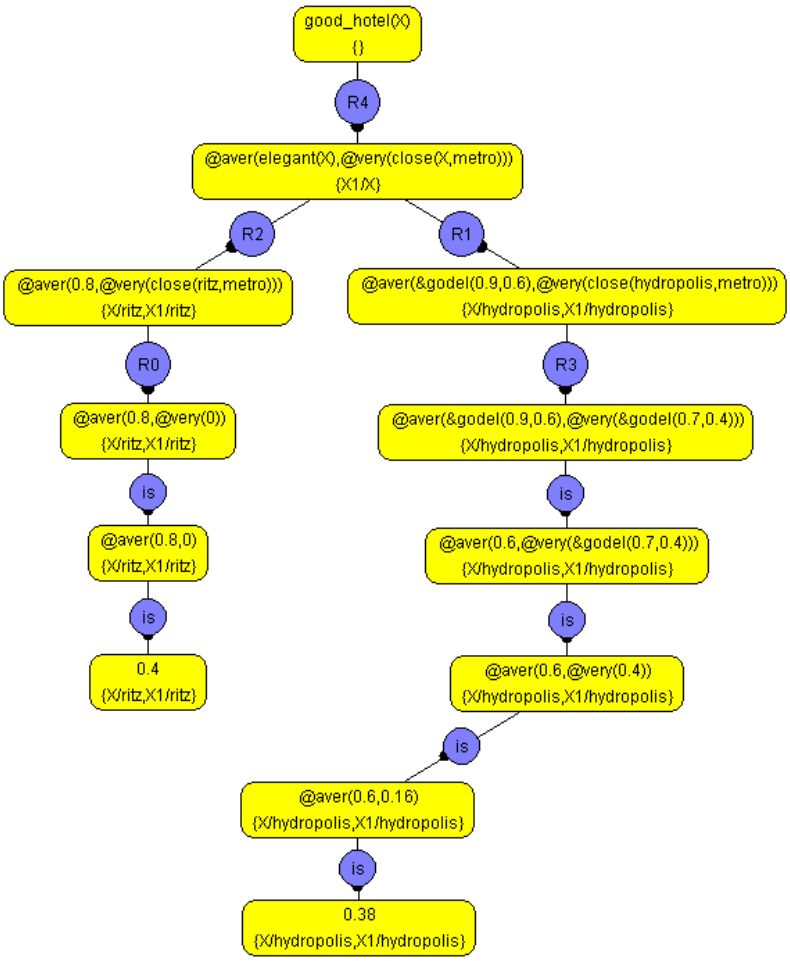


Figure 3: An execution tree as shown by the FLOPER system

cope with more expressive languages and, in particular, with FASILL (that is freely accessible in its url `http://dectau.uclm.es/floper/?q=sim` where it is possible to test/download the new prototype incorporating the management of similarity relations. In this section we briefly describe the main features of this tool before presenting the novelties introduced in this work.

FLOPER has been implemented in Sicstus Prolog v.3.12.5 (rounding about 1083 lines of code, where our last update supposes approximately a 30% of the final code) and it has been recently equipped with a graphical interface written in Java (circa 2000 lines of code). More detailed, the FLOPER system consists in a ".jar" java program that runs the graphical interface. This ".jar" program calls a ".pl" file containing the two main independent blocks: 1) the Parsing block parses FASILL files into two kinds of prolog code (a high level platform-independent Prolog program and a set of facts to be used by FLOPER), and 2) the Procedural block performs the evaluation of a goal against the program, implementing the procedural semantics previously described. This code is completed with a configuration file indicating the location of the Prolog interpreter as well as some other data.

When the graphical interface is executed, it offers a menu with a set of commands grouped in four

submenus:

- "Program Menu": includes options for *parsing* a FASILL program from a file with extension "`.fpl`", *saving* the generated PROLOG code to a "`.pl`" file, *loading/parsing* a pure PROLOG program, *listing* the rules of the parsed program and *cleaning* the database.
- "Lattice Menu": allows the user to change and show the lattice (implemented in PROLOG) associated to a fuzzy program through options *lat* and *show*, respectively.
- "Similarity Menu": option *sim* allows the user to load a similarity file (with extension "`.sim`", and whose syntax is detailed further in the Similarity Module subsection) and *tnorm* sets the conjunction to be used in the transitive closure of the relation.
- "Goal Menu": by choosing option *intro* the user introduces the goal to be evaluated. Option *tree* draws the execution tree for that goal whereas *leaves* only shows the set of fuzzy computed answer contained on it, and *depth* is used for fixing its maximum depth.

The syntax of FASILL presented in Section 1 is easily translated to be written by a computer. As usual in logic languages, variables are written as identifiers beginning by an upper case character or an underscore "_", while function and predicate symbols are expressed with identifiers beginning by a lower case character, and numbers are literals. Terms and atoms have the usual syntax (the function or predicate symbol, if no nullary, is followed by its arguments between parentheses and separated by a colon). Connectives are labeled with their name immediately after. The implication symbol is written as "`<-`", and each rule ends with a dot. Additionally it is possible to include pure PROLOG expressions inside the body of a rule by encapsuling them between curly brackets "{}", and PROLOG clauses together with FASILL rules between the dollar symbol "`$`".

In the recent years we have equipped the tool with a graphical interface (written in Java) for allowing a friendship interaction with the user, as seen in Figure 2. The graphical interface shows three areas. The leftmost one draws the project tree (grouping each category of file into its own directory). In the right part, the upper area displays the selected file of the tree and the lower one shows the code and the solutions of executing a goal. This interface groups files into projects which include a set of *fuzzy* files (`.fpl`), PROLOG files (`.pl`), *similarity* files (`.sim`), *script* files -containing a list of commands to be executed consecutively- (`.vfs`) and just one lattice file (`.lat`). When executing a goal, the tool considers the whole program merged from the set of files, thus obtaining only one fuzzy program, one similarity relation, one lattice and one PROLOG file.

The lattice module. Lattices are described in a `.lat` file using a language that is a subset of PROLOG where the definition of some predicates are mandatory, and the definition of aggregations follows a certain syntax. The mandatory predicates are `member/1`, that identifies the elements of the lattice, `bot/1` and `top/1`, that stand for the infimum and supremum elements of the lattice, and `leq/2`, that implements the ordering relation. Predicate `members/1`, that returns in a list all the elements of the lattice, is only required if it is finite. Connectives are defined as predicates whose meaning is given by a number of clauses. The name of the predicate has the form `and_`*label*, `or_`*label* or `agr_`*label* whether it implements a conjunction, a disjunction or an aggregator, where *label* is an identifier of that particular connective (this way one can define several conjunctions, disjunctions and other kind of aggregators instead of only one). The arity of the predicate is $n+1$, where n is the arity of the connective that it implements, so its last parameter is a variable to be unified with the value resulting of its evaluation.

$$\left.\begin{array}{l} ?- \ agr\_label(r_1,\ldots,r_n,R). \\ R = r. \end{array}\right\} \texttt{if } @_{label}(r_1,\ldots,r_n) = r$$

Example 1. *For instance, the following clauses show the* PROLOG *program modeling the lattice of the real interval* $[0,1]$ *with the usual ordering relation and connectives (conjunction and disjunction of the* Product logic*, as well as the average aggregator):*

```
member(X):- number(X), 0=<X, X=<1.                    leq(X,Y):- X=<Y.
and_prod(X,Y,Z) :- Z is X*Y.                          bot(0).
or_prod(X,Y,Z)  :- U1 is X*Y, U2 is X+Y, Z is U2-U1.  top(1).
agr_aver(X,Y,Z) :- U1 is X+Y, Z is U1/2.
```

The similarity module. We describe now the main novelty performed in the tool, that is the ability to take into account a similarity relation. The similarity relation $\mathscr{R}$ is loaded from a file with extension `.sim` through option *sim*. The relation is represented following a concrete syntax:

$\langle Relation\rangle$ ::= $\langle Sim\rangle$ $\langle Relation\rangle$ | $\langle Sim\rangle$
$\langle Sim\rangle$ ::= $\langle Id_f\rangle$['/' $\langle Int_n\rangle$] '$\sim$' $\langle Id_g\rangle$['/' $\langle Int_n\rangle$] '=' $\langle r\rangle$ '.' | '$\sim$' 'tnorm' '=' $\langle tnorm\rangle$

The *Sim* option parses expressions like "$f \sim g = r$", where f and g are propositional variables or constants and r is an element of L. It also copes with expressions including arities, like "$f/n \sim g/n = r$" (then, f and g are function or predicate symbols). In this case, both arities have to be the same. It is also possible to explicit, through a line like "$\sim tnorm = \langle label\rangle$" the conjunction to be used further in the construction of the transitive closure of the relation. Internally FLOPER stores each relation as a fact r in an ad hoc module *sim* as $r(f/n, g/n, r)$, where $n = 0$ if it has not been specified (that is, the symbol is considered as a constant). The `.sim` file contains only a small set of similarity equations that FLOPER completes by performing the reflexive, symmetric and transitive closure. The first one simply consists of the assertion of the fact $r(A, A, \top)$. The symmetric closure produces, for each $r(a,b,r)$, the assertion of its symmetric entry $r(b,a,r)$ if there is not already some $r(b,a,r')$ where $r \leq r'$ (in this case $r(a,b,r)$ will be rewritten as $r(a,b,r')$ when considering $r(b,a,r)$). The transitive closure is computed by the next algorithm[7], where $\wedge$ stands for the conjunction specified by the directive "*tnorm*", and "*assert*" and "*retract*" are self-explainable and defined as in PROLOG:

Transitive Closure
forall r(A,B,r_1) **in** `sim`
  forall r(B,C,r_2) **in** `sim`
    $r = r_1 \wedge r_2$
    if r(A,C,r') **in** `sim` **and** $r' < r$
      retract r(A,C,r') **from** `sim`
      retract r(C,A,r') **from** `sim`
    end if
    if r(A,C,r') **not in** `sim`
      assert r(A,C,r) **in** `sim`
      assert r(C,A,r) **in** `sim`
    end if
  end forall
end forall

It is important to note that, it is not relevant if the user provides (apparently) inconsistent similarity equations, since FLOPER automatically changes the user values by the appropriate approximation de-

[7] It is important to note that this algorithm must be executed right after performing the symmetric, reflexive closure.

grees in order to preserve the properties of a similarity. For instance, if a user provides a set of equations such as, $a \sim b = 0.8$, $b \sim c = 0.6$ and $a \sim c = 0.3$, after the application of our algorithm for the construction of a similarity, results in the set of equations $a \sim b = 0.8$, $b \sim c = 0.6$ and $a \sim c = 0.6$, which positively preserves the transitive property[8].

Example 2. *In order to illustrate the enhanced expressiveness of* FASILL*, consider the program* $\langle \Pi, \mathscr{R}, L \rangle$ *(where L is the real interval* $[0,1]$ *and* $\leq$ *is the usual ordering relation on real numbers), that models the concept of* good hotel*, that is, an elegant hotel that is very close to a metro entrance, as seen in Figure 2. Here, we use an average aggregator defined as* $\dot{@}_{avg}(x,\ y) \triangleq (x+y)/2$*, whereas* very *is a linguistic modifier implemented as well as an aggregator (with arity 1) with truth function* $\dot{@}_{very}\ x \triangleq x^2$*. The similarity relation* $\mathscr{R}$ *states that* elegant *is similar to* vanguardist*, and* metro *to* bus *and (by transitivity) to* taxi*:*

```
~tnorm  = godel                          metro ~ bus  = 0.5.
elegant/1 ~ vanguardist/1  = 0.6.        bus   ~ taxi = 0.4.
```

We also state that the t-norm to be used in the transitive closure is the conjunction of Gödel (i.e., the infimum between two elements). With respect to this program (the set of rules from Figure 2, the lattice [0,1] with the usual ordering relation and the similarity relation just described before), the goal `good_hotel(X)` *produces two fuzzy computed answers:* `<0.4, X/ritz>` *and* `<0.38, X/hydropolis>`*. Each one corresponds to the leaves of the tree*[9] *depicted in Figure 2. Note that for reaching these solutions, a failure step was performed in the derivation of the left-most branch, whereas in the right-most one (and this is the crucial novelty w.r.t. previous versions of the* FLOPER *tool) there exist two successful steps exploiting the similarity relation which firstly relates* `elegant` *and* `vanguardist` *and secondly (by transitivity)* `metro` *and* `taxi` *when solving atom* $close(hydropolis, metro)$*, which illustrates the flexibility of our system.*

Ending this section, it is worthy to say that our approach differs from the one presented in [1] since they employ a combination of transformation techniques to first extract the definition of a predicate "$\sim$", simulating weak unification in terms of a set of complex program rules that extends the original program. Finally, this predicate "$\sim$" is reduced to a built-in proximity/similarity unification operator (in this case not implemented by rules and very close to the implementation of our weak unification algorithm) that highly improves the efficiency of their previos programming systems.

4 Conclusions and Future Work

This work was concerned with the last enrichment performed on our FLOPER system to cope with similarity relations. In [5, 4, 11] we provide some advances in the design of declarative semantics and/or correctness properties regarding the development of fuzzy logic languages dealing with similarity/proximity relations (Bousi$\sim$Prolog) or highly expressive lattices modeling truth degrees (MALP). As a matter of future work we want to establish that analogous –but reinforced– features also hold in the twofold integrated fuzzy language FASILL whose syntax, procedural principle (based on weak -instead of syntactic- unification for managing similarity relations) and implementation details were described along this paper.

[8] For simplicity we have omitted the equations obtained during the construction of the reflexive, symmetric closure.

[9] Each state contains its corresponding goal and substitution components and they are drawn inside yellow ovals. Computational steps, colored in blue, are labeled with the program rule they exploit in the case of *successful* steps or the annotation "R0" in the case of *failure* steps (observe that, "R0" is a simple notation and do not correspond with any existing rule). Finally, the blue circles annotated with the word "is", correspond to *interpretive* steps.

References

[1] R. Caballero, M. Rodríguez-Artalejo & C. Romero-Díaz (2014): *A Transformation-based implementation for CLP with qualification and proximity*. Theory and Practice of Logic Programming 14(1), pp. 1–63. Available at http://dx.doi.org/10.1017/S1471068412000014.

[2] F. A. Fontana (2002): *Likelog for flexible query answering*. Soft Computing 7(2), pp. 107–114.

[3] F. Formato, G. Gerla & M.I. Sessa (2000): *Similarity-based Unification.* Fundamenta Informaticae 41(4), pp. 393–414.

[4] P. Julián-Iranzo, G. Moreno & J. Penabad (2009): *On the Declarative Semantics of Multi-Adjoint Logic Programs*. In: Proc. of 10th Intl Work-Conference on Artificial Neural Networks (Part I), IWANN'09, Springer Verlag, LNCS 5517, pp. 253–260. Available at http://dx.doi.org/10.1007/978-3-642-02478-8_32.

[5] P. Julián-Iranzo & C. Rubio-Manzano (2009): *A declarative semantics for Bousi∼Prolog*. In: Proc. of 11th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'09, ACM, pp. 149–160. Available at http://doi.acm.org/10.1145/1599410.1599430.

[6] P. Julián-Iranzo & C. Rubio-Manzano (2010): *An efficient fuzzy unification method and its implementation into the Bousi∼Prolog system*. In: Proc. of the 2010 IEEE International Conference on Fuzzy Systems, IEEE, pp. 1–8. Available at http://dx.doi.org/10.1109/FUZZY.2010.5584193.

[7] M. Kifer & V.S. Subrahmanian (1992): *Theory of generalized annotated logic programming and its applications.* Journal of Logic Programming 12, pp. 335–367.

[8] J. Medina, M. Ojeda-Aciego & P. Vojtáš (2004): *Similarity-based Unification: a multi-adjoint approach*. Fuzzy Sets and Systems 146, pp. 43–62.

[9] P.J. Morcillo, G. Moreno, J. Penabad & C. Vázquez (2010): *A Practical Management of Fuzzy Truth Degrees using FLOPER*. In: Proc. of 4nd. Intl. Symposium on Rule Interchange and Applications, RuleML'10, Springer Verlag, LNCS 6403, pp. 20–34. Available at http://dx.doi.org/10.1007/978-3-642-16289-3_4.

[10] G. Moreno, J. Penabad & C. Vázquez (2013): *Relaxing the Role of Adjoint Pairs in Multi-adjoint Logic Programming*. In I. Hamilton & J. Vigo-Aguiar, editors: Proc. of 13th International Conference on Mathematical Methods in Science and Engineering, CMMSE'13 (Volume III), pp. 1156–1167.

[11] G. Moreno, J. Penabad & C. Vázquez (2014): *Fuzzy Sets for a Declarative Description of Multi-adjoint Logic Programming*. In: Proc. of the 9th International Conference on Rough Sets and Current Trends in Soft Computing, RSCTC 2014, Springer Verlag, LNCS 8536, pp. 71–82. Available at http://dx.doi.org/10.1007/978-3-319-08644-6_7.

[12] G. Moreno & C. Vázquez (2014): *Fuzzy Logic Programming in Action with FLOPER*. Journal of Software Engineering and Applications 7, pp. 237–298. Available at http://dx.doi.org/10.4236/jsea.2014.74028.

[13] S. Muñoz-Hernández, V. Pablos Ceruelo & H. Strass (2011): *RFuzzy: Syntax, semantics and implementation details of a simple and expressive fuzzy tool over Prolog*. Information Sciences 181(10), pp. 1951–1970. Available at http://dx.doi.org/10.1016/j.ins.2010.07.033.

[14] H. T. Nguyen & E. A. Walker (2006): *A First Course in Fuzzy Logic*. Chatman & Hall, Boca Ratón, Florida. Third edition.

[15] C. Rubio-Manzano & P. Julián-Iranzo (2014): *A Fuzzy linguistic prolog and its applications*. Journal of Intelligent and Fuzzy Systems 26(3), pp. 1503–1516. Available at http://dx.doi.org/10.3233/IFS-130834.

[16] M.I. Sessa (2002): *Approximate reasoning by similarity-based SLD resolution.* Theoretical Computer Science 275(1-2), pp. 389–426. Available at http://dx.doi.org/10.1016/S0304-3975(01)00188-8.

The ModelCC Model-Driven Parser Generator: A Tutorial

Fernando Berzal Francisco J. Cortijo Juan-Carlos Cubero Luis Quesada
CITIC & Department of Computer Science and Artificial Intelligence
University of Granada, Spain
{berzal|cb|jc.cubero|lquesada}@modelcc.org

Syntax-directed translation tools require the specification of a language by means of a formal grammar. This grammar must also conform to the specific requirements of the parser generator to be used. Software engineers then annotate the resulting grammar with semantic actions for the resulting system to perform its desired functionality. In this paper, we introduce ModelCC, a model-based parser generator that decouples language specification from language processing, avoiding some of the problems caused by grammar-driven parser generators. ModelCC receives a conceptual model as input, along with constraints that annotate it. It is then able to create a parser for the desired textual syntax and the generated parser fully automates the instantiation of the language conceptual model. ModelCC includes a reference resolution mechanism so that, rather than mere abstract syntax trees, it is able to instantiate abstract syntax graphs.

1 Introduction

The most widely-used language processing tools typically require language designers to provide a textual description of the language syntax as a BNF-like grammar. The proper specification of such a grammar is a nontrivial process that depends on the lexical and syntactic analysis techniques to be used, since each kind of technique requires the grammar to comply with different constraints. Each analysis technique is characterized by its expression power and this expression power determines whether a given analysis technique is suitable for a particular language. The most significant constraints on formal language specification originate from the need to consider context-sensitivity, the need of performing an efficient analysis, and some techniques' inability to consider grammar ambiguities or resolve conflicts caused by them.

Whenever the language syntax has to be modified, the language designer has to manually propagate changes throughout the entire language processor tool chain. These updates are time-consuming, tedious, and error-prone. By making such changes labor-intensive, the traditional approach hampers the maintainability and evolution of the language [16].

Moreover, it is not uncommon that different tools use the same language, including compilers, code generators, and debuggers. Multiple copies of the same language specification have to be maintained in sync, since language specification (i.e. its grammar) is tightly coupled to language processing (i.e. the semantic actions that annotate that grammar).

A grammar is a model of the language it defines, but a language can also be defined by a conceptual data model that represents the abstract syntax of the desired language, focusing on the elements the language will represent and their relationships. In conjunction with the declarative specification of some constraints, such model can be automatically converted into a grammar-based language specification [21]. The model representing the language can be modified as needed, without having to worry about the language processor and the peculiarities of the chosen parsing technique, since the corresponding language processor will be automatically updated.

Submitted to:
PROLE 2014

Furthermore, the conceptual model can be naturally implemented as a set of collaborating classes in object-oriented programming languages. Following proper software design principles, that implementation avoids the embedding of semantic actions within the language specification, as it is typically done with grammar-driven language processors.

Finally, as the language model is not bound to a particular parsing technique, evaluating alternative and/or complementary parsing techniques is possible without having to propagate their constraints into the language model. By using an annotated data model, model-based language specification completely decouples language specification from language processing, which can be performed using whichever parsing techniques that might be suitable for the formal language implicitly defined by the model.

It should be noted that, while, in general, the result of the parsing process is an abstract syntax tree that corresponds to a valid parsing of the input text according to the language concrete syntax, nothing prevents the model-based language designer from modeling non-tree structures. Indeed, a model-driven parser generator can automate the implementation of reference resolution mechanisms, among other syntactic and semantic checks that are typically deferred to later stages in the language processing pipeline [24]. ModelCC is able to resolve references and obtain abstract syntax graphs as the result of the parsing process, rather than the traditional abstract syntax trees obtained from conventional parser generators.

2 Model-Based Language Specification

In this Section, we analyze the concepts of abstract and concrete syntax (2.1), discuss the potential advantages of model-based language specification (2.2), and compare our proposed approach with the traditional grammar-driven language design process (2.3).

2.1 Abstract Syntax and Concrete Syntaxes

The abstract syntax of a language is just a representation of the structure of the different elements of a language without the superfluous details related to its particular textual representation [17]. On the other hand, a concrete syntax is a particularization of the abstract syntax that defines, with precision, a specific textual or graphical representation of the language. It should also be noted that a single abstract syntax can be shared by several concrete syntaxes [17].

For example, the abstract syntax of the typical <*if*>-<*then*>-<*optional else*> statement in imperative programming languages could be described as the concatenation of a conditional expression and one or two statements. Different concrete syntaxes could be defined for such an abstract syntax, which would correspond to different textual representations of a conditional statement, e.g. {“`if`”, “`(`”, expression, “`)`”, statement, optional “`else`” followed by another statement} and {“`if`”, expression, “`then`”, statement, optional “`else`” followed by another statement, “`endif`”}.

The idea behind model-based language specification is that, starting from a single abstract syntax model (ASM) representing the core concepts in a language, language designers would later develop one or several concrete syntax models (CSMs). These concrete syntax models would suit the specific needs of the desired textual or graphical representation for the language sentences. The ASM-CSM mapping could be performed, for instance, by annotating the abstract syntax model with the constraints needed to transform the elements in the abstract syntax into their concrete representation.

2.2 Advantages of Model-Based Language Specification

Focusing on the abstract syntax of a language offers some benefits [17] and provides some potential advantages to model-based language specification over the traditional grammar-based language specification approach:

- When reasoning about the features a language should include, specifying its abstract syntax seems to be a better starting point than working on its concrete syntax details. In fact, we control complexity by building abstractions that hide details when appropriate [1].
- Sometimes, different incarnations of the same abstract syntax might be better suited for different purposes (e.g. an human-friendly syntax for manual coding, a machine-oriented format for automatic code generation, a Fit-like [18] syntax for testing, different architectural views for discussions with project stakeholders...). Therefore, it might be useful for a given language to support multiple syntaxes.
- Since model-based language specification is independent from specific lexical and syntactic analysis techniques, the constraints imposed by specific parsing algorithms do not affect the language design process. In principle, however, it might not be even necessary for the language designer to have advanced knowledge on parser generators when following a model-driven language specification approach.
- A full-blown model-driven language workbench [11, 25, 6, 14, 5, 13] would allow the modification of a language abstract syntax model and the automatic generation of a working IDE on the run. The specification of domain-specific languages would become easier, as the language designer could play with the language specification and obtain a fully-functioning language processor on the fly, without having to worry about the propagation of changes throughout the complete language processor tool chain.

In short, the model-driven language specification approach brings domain-driven design [9] to the domain of language design. It provides the necessary infrastructure for what Evans would call the 'supple design' of language processing tools: the intention-revealing specification of languages by means of abstract syntax models, the separation of concerns in the design of language processing tools by means of declarative ASM-CSM mappings, and the automation of a significant part of the language processor implementation.

2.3 Comparison with the Traditional Approach

A diagram summarizing the traditional language design process is shown in Figure 1, whereas the corresponding diagram for the model-based approach proposed in this paper is shown in Figure 2.

When following the traditional grammar-driven approach, the language designer starts by designing the grammar corresponding to the concrete syntax of the desired language, typically in BNF or a similar format. Then, the designer annotates the grammar with attributes and, probably, semantic actions, so that the resulting attribute grammar can be fed into lexer and parser generator tools that produce the corresponding lexer and parser, respectively. The resulting syntax-directed translation process generates abstract syntax trees from the textual representation in the concrete syntax of the language.

When following the model-driven approach, the language designer starts by designing the conceptual model that represents the abstract syntax of the desired language, focusing on the elements the language will represent and their relationships. Instead of dealing with the syntactic details of the language from

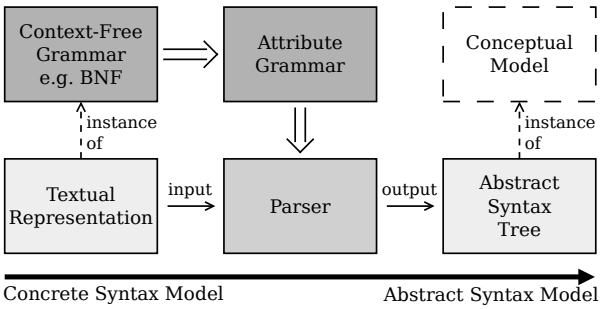


Figure 1: Traditional language processing approach.

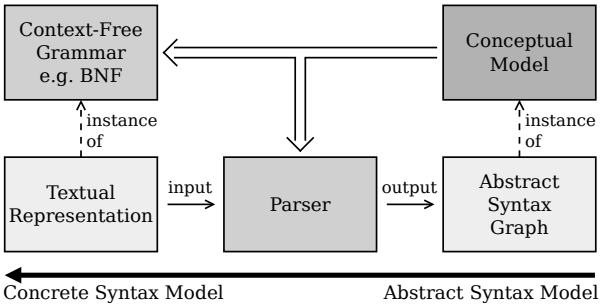


Figure 2: Model-based language processing approach.

the start, the designer devises a conceptual model for it (i.e. the abstract syntax model, or ASM), the same way a database designer starts with an implementation-independent conceptual database schema before he converts that schema into a logical schema that can be implemented in the particular kind of DBMS that will host the final database. In the model-driven language design process, the ASM would play the role of entity-relationship diagrams in database design and each particular CSM would correspond to the final table layout of the physical database schema in a relational DBMS.

Even though the abstract syntax model of the language could be converted into a suitable concrete syntax model automatically, the language designer will often be interested in specifying the details of the ASM-CSM mapping. With the help of constraints imposed over the abstract model, the designer will be able to guide the conversion from the ASM to its concrete representation using a particular CSM. This concrete model, when it corresponds to a textual representation of the abstract model, will be described by a formal grammar. It should be noted, however, that the specification of the ASM is independent from the peculiarities of the desired CSM, as a database designer does not consider foreign keys when designing the conceptual schema of a database. Therefore, the grammar specification constraints enforced by particular parsing tools will not impose limits on the design of the ASM. The model-driven language processing tool will take charge of that and, ideally, it will employ the most efficient parsing technique that works for the language resulting from the ASM-CSM mapping.

While the traditional language designer specifies the grammar for the concrete syntax of the language, annotates it for syntax-directed processing, and obtains an abstract syntax tree that is an instance of the implicit conceptual model defined by the grammar, the model-based language designer starts with an explicit full-fledged conceptual model and specifies the necessary constraints for the ASM-CSM mapping. In both cases, parser generators create the tools that parse the input text in its concrete syntax. The difference lies in the specification of the grammar that drives the parsing process, which is hand-crafted

in the traditional approach and automatically-generated as a result of the ASM-CSM mapping in the model-driven process.

Another difference stems from the fact that the result of the parsing process is an instance of an implicit model in the grammar-driven approach while that model is explicit in the model-driven approach. An explicit conceptual model is absent in the traditional language design process albeit that does not mean that it does not exist. On the other hand, the model-driven approach enforces the existence of an explicit conceptual model, which lets the proposed approach reap the benefits of domain-driven design.

There is a third difference between the grammar-driven and the model-driven approaches to language specification. While, in general, the result of the parsing process is an abstract syntax tree that corresponds to a valid parsing of the input text according to the language concrete syntax, nothing prevents the conceptual model designer from modeling non-tree structures, which describe grammars with a power of expression similar to reference attribute grammars [7]. Hence the use of the 'abstract syntax graph' term in Figure 2. This might be useful, for instance, for modeling graphical languages, which are not constrained by the linear nature of the traditional syntax-driven specification of text-based languages.

Instead of going from a concrete syntax model to an implicit abstract syntax model, as it is typically done, the model-based language specification process goes from the abstract to the concrete. This alternative approach facilitates the proper design and implementation of language processing systems by decoupling language processing from language specification, which is now performed by imposing declarative constraints on the ASM-CSM mapping.

3 ModelCC Model Specification

Once we have described model-driven language specification in general terms, we now proceed to introduce ModelCC [23], a tool that supports our proposed approach to the design of language processing systems. ModelCC, at its core, acts as a parser generator. The starting abstract syntax model is created by defining classes that represent language elements and establishing relationships among those elements (associations in UML terms). Once the abstract syntax model is established, its incarnation as a concrete syntax is guided by the constraints imposed over language elements and their relationships as annotations on the abstract syntax model. In other words, the declarative specification of constraints over the ASM establishes the desired ASM-CSM mapping.

In this section, we introduce the basic constructs that allow the specification of abstract syntax models, while we will discuss how model constraints help us establish a particular ASM-CSM mapping in the following section of this paper. Basically, the ASM is built on top of basic language elements, which might be viewed as the tokens in the model-driven specification of a language. Model-driven language processing tools such as ModelCC provide the necessary mechanisms to combine those basic elements into more complex language constructs, which correspond to the use of concatenation, selection, and repetition in the syntax-driven specification of languages.

Our final goal is to allow the specification of languages in the form of abstract syntax models such as the one shown in Figure 6, which will be used as an example in Section 5. This model, in UML format, specifies the abstract syntax model of the language supported by a simple arithmetic expression language. The annotations that accompany the model provide the necessary information for establishing the complete ASM-CSM mapping that corresponds to the traditional infix notation for arithmetic expressions. Moreover, the model also incorporates the method that lets us evaluate such arithmetic expressions. Therefore, Figure 6 represents a complete interpreter for arithmetic expressions in infix notation using ModelCC.

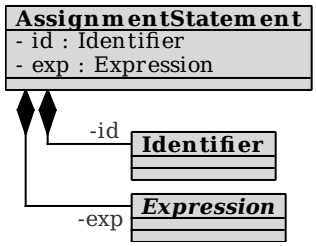


Figure 3: An assignment statement as an example of element composition (concatenation in textual CSM terms).

As mentioned above, the specification of the ASM in ModelCC starts with the definition of basic language elements, which can be modeled as simple classes in an object-oriented programming language. The ASM-CSM mapping of those basic elements will establish their correspondence to the tokens that appear in the concrete syntax of the language whose ASM we design in ModelCC.

In the following subsections, we describe the mechanisms provided by ModelCC to implement the three main constructs that let us specify complete abstract syntax models on top of basic language elements.

3.1 Concatenation

Concatenation is the most basic construct we can use to combine sets of language elements into more complex language elements. In textual languages, this is achieved just by joining the strings representing its constituent language elements into a longer string that represents the composite language element.

In ModelCC, concatenation is achieved by object composition. The resulting language element is the composite element and its members are the language elements the composite element collates.

When translating the ASM into a textual CSM, each composite element in a ModelCC model generates a production rule in the grammar representing the CSM. This production, with the nonterminal symbol of the composite element in its left-hand side, concatenates the nonterminal symbols corresponding to the constituent elements of the composite element in its right-hand side. By default, the order of the constituent elements in the production rule is given by the order in which they are specified in the object composition, but such an order is not strictly necessary (e.g. many ambiguous languages might require differently ordered sequences of constituent elements and even some unambiguous languages allow for unordered sequences of constituent elements).

The model in Figure 3 shows an example of object composition in ASM terms that corresponds to string concatenation in CSM terms. In this example, an assignment statement is composed of an identifier, i.e. a reference to its l-value, and an expression, which provides its r-value. In a textual CSM, the composite *AssignmentStatement* element would be translated into the following production rule: *<AssignmentStatement> ::= <Identifier> <Expression>*. Obviously, such production would probably include some syntactic sugar in an actual programming language, either for avoiding potential ambiguities or just for improving its readability and writability, but that is the responsibility of ASM-CSM mappings, which will be analyzed in Section 4.

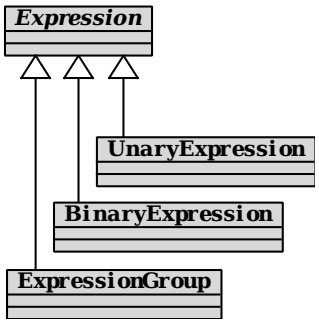


Figure 4: Subtyping for representing choices in ModelCC.

3.2 Selection

Selection is necessary as a language modeling primitive operation to represent choices, so that we can specify alternative elements in language constructs.

In ModelCC, selection is achieved by subtyping. Specifying inheritance relationships among language elements in an object-oriented context corresponds to defining 'is-a' relationships in a more traditional database design setting. The language element we wish to establish alternatives for is the superelement (i.e. the superclass in OO or the supertype in DB modeling), whereas the different alternatives are represented as subelements (i.e. subclasses in OO, subtypes in DB modeling). Alternative elements are always kept separate to enhance the modularity of ModelCC abstract syntax models and their integration in language processing systems.

In the current version of ModelCC, multiple inheritance is not supported, albeit the same results can be easily simulated by combining inheritance and composition. We can define subelements for the different inheritance hierarchies representing choices so that those subelements are composed by the single element that appears as a common choice in the different scenarios. This solution fits well with most existing programming languages, which do not always support multiple inheritance, and avoids the pollution of the shared element interface in the ASM, which would appear as a side effect of allowing multiple inheritance in abstract syntax models.

Each inheritance relationship in ModelCC, when converting the ASM into a textual CSM, generates a production rule in the CSM grammar. In those productions, the nonterminal symbol corresponding to the superelement appears in its left-hand side, while the nonterminal symbol of the subelement appears as the only symbol in the production right-hand side. Obviously, if a given superelement has k different subelements, k different productions will be generated representing the k alternatives defined by the abstract syntax model.

The model shown in Figure 4 illustrates how an arithmetic *Expression* can be either an *UnaryExpression*, a *BinaryExpression*, or an *ExpressionGroup* in the language defined for a simple arithmetic calculator, as defined in Section 5. The context-free grammar resulting from the conversion of this ASM into a textual CSM would be: <*Expression*> ::= <*UnaryExpression*> | <*BinaryExpression*> | <*ExpressionGroup*>.

3.3 Repetition

Representing repetition is also necessary in abstract syntax models, since a language element might appear several times in a given language construct. When a variable number of repetitions is allowed,

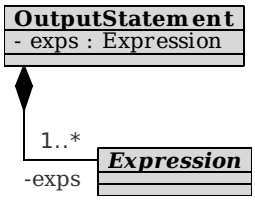


Figure 5: Multiple composition for representing repetition in ModelCC.

mere concatenation does not suffice.

Repetition is also achieved though object composition in ModelCC, just by allowing different multiplicities in the associations that connect composite elements to their constituent elements. The cardinality constraints described in Section 4 can be used to annotate ModelCC models in order to establish specific multiplicities for repeatable language elements.

Each composition relationship representing a repetitive structure in the ASM will lead to two additional production rules in the grammar defining a textual CSM: a recursive production of the form <*ElementList*> ::= <*Element*> <*ElementList*> and a complementary production <*ElementList*> ::= <*Element*>, where <*Element*> is the nonterminal symbol associated to the repeating element.

It should also be noted that <*ElementList*> will take the place of the nonterminal <*Element*> in the production derived from the composition relationship that connects the repeating element with its composite element (see the above section on how composition is employed to represent concatenation in ModelCC).

In practice, repeating elements will often appear separated in the concrete syntax of a textual language, hence repeatable elements can be annotated with separators, as we will see in Section 4. In case separators are employed, the recursive production derived from repeatable elements will be of the form <*ElementList*> ::= <*Element*> <*Separator*> <*ElementList*>.

When a repeatable language element is optional, i.e. its multiplicity can be 0, an additional epsilon production is appended to the grammar defining the textual CSM derived from the ASM: <*ElementList*> ::= ε.

For example, the model in Figure 5 shows that an *OutputStatement* can include several *Expression*s, which will be evaluated for their results in order for them to be sent to the corresponding output stream. This ASM would result in the following textual CSM grammar:

<*OutputStatement*> ::= <*ExpressionList*>

<*ExpressionList*> ::= <*Expression*> <*ExpressionList*> | <*Expression*>

4 ModelCC Model Constraints

Once we have examined the mechanisms that let us create abstract syntax models in ModelCC, we now proceed to describe how constraints can be imposed on such models in order to establish the desired ASM-CSM mapping:

- A first set of constraints is used for pattern specification, a necessary feature for defining the lexical elements of the concrete syntax model, i.e. its tokens.
- A second set of constraints is employed for defining delimiters in the concrete syntax model, whose use is common for eliminating language ambiguities or just as syntactic sugar in many languages.

Constraints on	Annotation	Function
... patterns	@Pattern	Pattern matching specification of basic language elements.
	@Value	Field where the recognized input token will be stored.
... delimiters	@Prefix	Element prefix(es).
	@Suffix	Element suffix(es).
	@Separator	Element separator(s) in lists of elements.
... cardinality	@Optional	Optional elements.
	@Multiplicity	Minimum and maximum element multiplicity.
... evaluation order	@Associativity	Element associativity (e.g. left-to-right).
	@Composition	Eager or lazy composition for nested composites.
	@Priority	Element precedence level/relationships.
... composition order	@Position	Element member relative position.
	@FreeOrder	When there is no predefined order among element members.
... references	@ID	Identifier of a language element.
	@Reference	Reference to a language element.
Custom constraints	@Constraint	Custom user-defined constraint.

Table 1: The constraints supported by the ModelCC model-based parser generator.

- A third set of ModelCC constraints lets us impose cardinalities on language elements, which control element repeatability and optionality.
- A fourth set of constraints lets us impose evaluation order on language elements, which are employed to declaratively resolve further ambiguities in the concrete syntax of a textual language by establishing associativity, precedence, and composition policies, the latter employed, for example, for resolving the ambiguities that cause the typical shift-reduce conflicts in LR parsers.
- A fifth set of constraints lets us specify the element constituent order in composite elements.
- A sixth set of constraints lets us specify referenceable language elements and references to them.
- Finally, custom constraints let us provide specific lexical, syntactic, and semantic constraints that take into consideration context information.

Table 1 summarizes the set of constraints supported by ModelCC for establishing ASM-CSM mappings between abstract syntax models and their concrete representation in textual CSMs.

As soon as that ASM-CSM mapping is established, ModelCC is able to generate the suitable parser for the concrete syntax defined by the CSM. ModelCC allows the definition of ASM-CSM constraints using metadata annotations or a domain-specific language.

Now supported by all the major programming platforms, metadata annotations have been used in reflective programming and code generation [10]. Among many other things, they can be employed for dynamically extending the features of your software development runtime [4] or even for building complete model-driven software development tools that benefit from the infrastructure provided by your compiler and its associated tools [12].

The ModelCC domain-specific language for ASM-CSM mappings [22] supports the separation of concerns in the design of language processing tools by allowing the definition of different CSMs for a common ASM.

5 A Simple Example

An interpreter for arithmetic expressions in infix notation can be used to illustrate the differences between ModelCC and more conventional tools. Its full implementation using two well-known parser generators (lex & yacc, and ANTLR) is available at `http://www.modelcc.org/examples`. Albeit the arithmetic expression example is necessarily simplistic, it already provides some hints on the potential benefits model-driven language specification can bring to more challenging endeavors. This simple language is also used in the next section as the basis for a more complex language, which illustrates how ModelCC supports language composition.

Using conventional tools, the language designer would start by specifying the grammar defining the arithmetic expression language in a BNF-like notation.

When using lex & yacc, the language designer converts the BNF grammar into a grammar suitable for LR parsing. Since lex does not support lexical ambiguities, the unary and binary operator nonterminals from the BNF grammar have to be refactored in order to avoid the ambiguities introduced by the use of + and - both as unary and binary operators. A similar solution is required for distinguishing operator priorities. Unfortunately, the resolution of ambiguities involves the introduction of a certain degree of duplication in the language specification: separate token types in the lexer and multiple parallel production rules in the parser. Once the ambiguities have been resolved, the language designer completes the lex & yacc introducing semantic actions to perform the necessary operations: albeit somewhat verbose using the C programming language syntax, the implementation of an arithmetic expression interpreter is relatively straightforward.

When using ANTLR, the language designer converts the BNF grammar into a grammar suitable for LL parsing. LL(*) parsers do not support left-recursion, so left-recursive grammar productions must be refactored. Since ANTLR provides no mechanism for the declarative specification of token precedences, such precedences have to be incorporated into the grammar. The usual solution involves the creation of different nonterminal symbols in the grammar, so that productions corresponding to the same precedence levels are grouped together. Once the grammar is adjusted to satisfy the constraints imposed by the ANTLR parser generator, the language designer can define the semantic actions needed to implement our arithmetic expression interpreter. The streamlined syntax of the scannerless ANTLR parser generator makes this implementation significantly more concise than the equivalent lex & yacc implementation.

When following a model-based language specification approach, the language designer starts by elaborating an abstract syntax model, which will later be mapped to a concrete syntax model by imposing constraints on the abstract syntax model. These constraints can also be specified as metadata annotations on the abstract syntax model and the resulting annotated model can be processed by automated tools, such as ModelCC, to generate the corresponding lexers and parsers. Annotated models can be represented graphically, as the UML diagram in Figure 6, or implemented using conventional programming languages, as the Java implementations available at `http://www.modelcc.org/examples`.

Using modern programming languages, metadata annotations can be used for the ASM-CSM mapping corresponding to the desired concrete syntax model. In case several CSMs were needed, the ModelCC domain-specific language for ASM-CSM mappings could be used to specify alternative CSMs for the language ASM [22].

The parser that ModelCC generates from the arithmetic expression model parses input strings such as “10/(2+3)*0.5+1” and instantiates *Expression* objects from them. The `eval()` method yields the final result for any expression (2, in the previous example). Figure 9 shows the actual code needed to generate and invoke the parser in ModelCC.

In its current version, ModelCC generates Lamb lexers [19] and Fence parsers [20], albeit traditional

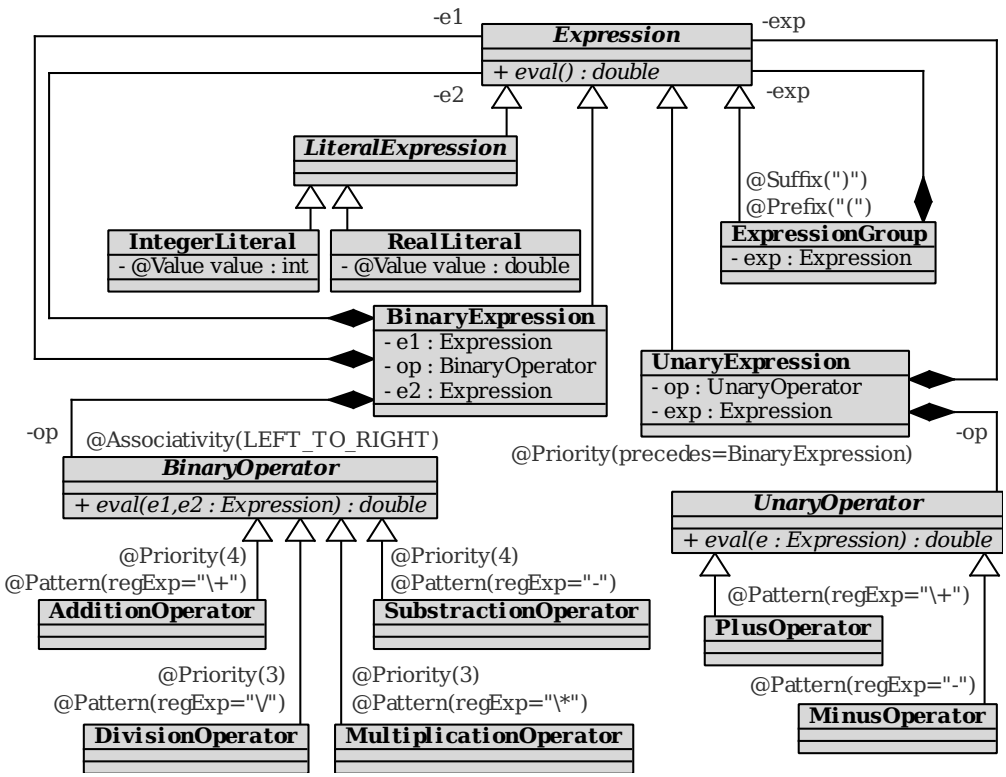


Figure 6: ModelCC specification of the arithmetic expression language.

LL and LR parsers might also be generated whenever the ASM-CSM mapping constraints make LL and LR parsing feasible. Whitespace and comments in the language can be defined by specifying what to ignore in the input text when the parser is created.

It should be noted that parse error handling is also completely dependant on the parser being used. Indeed, most parsers are able to provide comprehensive parsing error tracebacks.

However, ModelCC provides a testing framework that integrates well with existing IDEs and JUnit. Since separate language elements are models themselves, it is possible to implement both unitary and integration tests that focus on specific language elements. For example, assertions can check whether a model matches a certain string, whether a model does not match a certain string, or whether a model matches a string in a specific number of ways (e.g. matching without ambiguities). Assertions can, of course, take into consideration the contents or members of language elements for in-depth testing.

Since the abstract syntax model in ModelCC is not constrained by the vagaries of particular parsing algorithms, the language design process can be focused on its conceptual design, without having to introduce artifacts in the design just to satisfy the demands of particular tools:

- As we saw in the lex & yacc example, conventional tools, unless they are scannerless, force the creation of artificial token types in order to avoid lexical ambiguities, which leads to duplicate grammar production rules and semantic actions in the language specification. As in any other software development project, duplication hinders the evolution of languages and affects the maintainability of language processors. ModelCC, even though it is not scannerless, supports lexical ambiguities and each basic language element is defined as a separate and independent entity, even when their pattern specification are in conflict. Therefore, duplication in the language model does not have

```
public abstract class Expression implements IModel {
  public abstract double eval();
}

@Prefix("\\(") @Suffix("\\)")
public class ExpressionGroup extends Expression implements IModel {
  Expression e;
  @Override public double eval() { return e.eval(); }
}

public abstract class LiteralExpression extends Expression implements IModel {
}

public class UnaryExpression extends Expression implements IModel {
  UnaryOperator op;
  Expression e;
  @Override public double eval() { return op.eval(e); }
}

public class BinaryExpression extends Expression implements IModel {
  Expression e1;
  BinaryOperator op;
  Expression e2;
  @Override public double eval() { return op.eval(e1,e2); }
}

public class IntegerLiteral extends LiteralExpression implements IModel {
  @Value int value;
  @Override public double eval() { return (double)value; }
}

public class RealLiteral extends LiteralExpression implements IModel {
  @Value double value;
  @Override public double eval() { return value; }
}

public abstract class UnaryOperator implements IModel {
  public abstract double eval(Expression e);
}

@Associativity(AssociativityType.LEFT_TO_RIGHT)
public abstract class BinaryOperator implements IModel {
  public abstract double eval(Expression e1,Expression e2);
}
```

Figure 7: Complete Java implementation of the arithmetic expression interpreter using ModelCC (1/2): Java classes define the language ASM, metadata annotations specify the desired ASM-CSM mapping, and the `eval()` method implements arithmetic expression evaluation.

to be included to deal with lexical ambiguities: token type definitions do not have to be adjusted, duplicate syntactic constructs rules will not appear in the language model, and, as a consequence, semantic predicates do not have to be duplicated either.

- As we also saw both in the lex & yacc interpreter and in the ANTLR solution to the same problem, established parser generators require modifications to the language grammar specification in order to comply with parsing constraints, let it be the elimination of left-recursion for LL parsers or

```
@Priority(value=2) @Pattern(regExp="\\+")
public class AdditionOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()+e2.eval(); }
}

@Priority(value=2) @Pattern(regExp="-")
public class SubtractionOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()-e2.eval(); }
}

@Priority(value=1) @Pattern(regExp="\\*")
public class MultiplicationOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()*e2.eval(); }
}

@Priority(value=1) @Pattern(regExp="\\/")
public class DivisionOperator extends BinaryOperator {
  @Override public double eval(Expression e1,Expression e2) { return e1.eval()/e2.eval(); }
}

@Pattern(regExp="\\+")
public class PlusOperator extends UnaryOperator {
  @Override public double eval(Expression e) { return e.eval(); }
}

@Pattern(regExp="-")
public class MinusOperator extends UnaryOperator {
  @Override public double eval(Expression e) { return -e.eval(); }
}
```

Figure 8: Complete Java implementation of the arithmetic expression interpreter using ModelCC (2/2): Arithmetic operators.

```
// Read the model.
Model model = JavaModelReader.read(Expression.class);

// Generate the parser.
Parser<Expression> parser = ParserFactory.create(model);

// Parse the input string and instantiate the corresponding expression.
Expression expr = parser.parse("10/(2+3)*0.5+1");

// Evaluate the expression.
double value = expr.eval();
```

Figure 9: Code snippet showing how the arithmetic expression parser is generated and the resulting interpreter is invoked.

the introduction of new nonterminals to restructure the language specification so that the desired precedence relationships are fulfilled. In the model-driven language specification approach, the left-recursion problem disappears since it is something the underlying tool can easily deal with in a fully automated way when an abstract syntax model is converted into a concrete syntax model. Moreover, the declarative specification of constraints, such as the evaluation order constraints in Section 4, is orthogonal to the abstract syntax model that defines the language. Those constraints

determine the ASM-CSM mapping and, since ModelCC takes charge of everything in that conversion process, the language designer does not have to modify the abstract syntax model just because a given parser generator might prefer its input in a particular format. This is the main benefit that results from raising your abstraction level in model-based language specification.

- When changes in the language specification are necessary, as it is often the case when a software system is successful, the traditional language designer will have to propagate changes throughout the entire language processing tool chain, often introducing significant changes and making profound restructurings in the working code base. The changes can be time-consuming, quite tedious, and extremely error-prone. In contrast, modifications are more easily done when a model-driven language specification approach is followed. Any modifications in a language will affect either to the abstract syntax model, when new capabilities are incorporated into a language, or to the constraints that define the ASM-CSM mapping, whenever syntactic details change or new CSMs are devised for the same ASM. In either case, the more time-consuming, tedious, and error-prone modifications are automated by ModelCC, whereas the language designer can focus his efforts on the essential part of the required changes.
- Traditional parser generators typically mix semantic actions with the syntactic details of the language specification. This approach, which is justified when performance is the top concern, might lead to poorly-designed hard-to-test systems when not done with extreme care. Moreover, when different applications or tools employ the same language, any changes to the syntax of that language have to be replicated in all the applications and tools that use the language. The maintenance of several versions of the same language specification in parallel might also lead to severe maintenance problems. In contrast, the separation of concerns provided by ModelCC, as separate ASM and ASM-CSM mappings, promotes a more elegant design for language processing systems. By decoupling language specification from language processing and providing a conceptual model for the language, different applications and tools can now use the same language without having duplicate language specifications. A similar result could be hand-crafted using traditional parser generators (i.e. making their implicit conceptual model explicit and working on that explicit model), but ModelCC automates this part of the process.

In summary, while traditional language processing tools provide different mechanisms for resolving ambiguities and implementing language constraints, the solutions they provide typically interfere with the conceptual modeling of languages: relatively minor syntactic details might significantly affect the structure of the whole language specification. Model-driven language specification, as exemplified by ModelCC, provides a cleaner separation of concerns: the abstract syntax model is kept separate from its incarnation in concrete syntax models, thereby separating the specification of abstractions in the ASM from the particularities of their textual representation in CSMs.

6 More Complex Examples

In this section, we include the model of a full-fledged imperative programming language that illustrates language composition and reference resolution in ModelCC. The UML class diagrams in Figure 11 presents our annotated imperative programming language, which is complemented by the arithmetic expression language in Figure 6 and extended with new binary operators defined as *BinaryOperator* subclasses. This example illustrates ModelCC capabilities for language composition: the simple arithmetic expression language described in Section 5 is not only used within the imperative programming language, but it is also extended with new expression types and binary operators.

```
// Read the model.
Model model = JavaModelReader.read(Expression.class);

// Create the parser.
Parser<Expression> parser = ParserFactory.create(model);

// Define a constant
parser.add(new Constant("pi", 3.1415927));

// Use the predefined constant in JUnit tests for arithmetic expressions
assertEquals(3.1415927,   parser.parse("pi").eval(),   EPSILON);
assertEquals(2*3.1415927, parser.parse("2*pi").eval(), EPSILON);
```

Figure 10: Code snippet showing ModelCC support for separate compilation using predefined model elements.

ModelCC is able to automatically generate a grammar from the ASM defined by the class model and the ASM-CSM mapping, which is specified as a set of metadata annotations on the class model. These annotations also provide a mechanism for reference resolution that allows the automatic instantiation of complete object graphs. References are automatically resolved by ModelCC, resulting in abstract syntax graphs rather than mere abstract syntax trees. In our imperative language example, variables are automatically connected to the expressions and assignment statements where they appear. Likewise, function calls are automatically linked to the corresponding function definitions, without further intervention by the programmer.

Another interesting application of the reference resolution mechanism in ModelCC is illustrated by the code snippet in Figure 10. In this exam, a constant is predefined before the parser is invoked to parse an expression that includes a reference to the predefined constant, whose definition does not have to be included in the textual input of the parser, thus providing a crude but elegant form of separate compilation.

A fully-functional version of ModelCC for Java, additional examples of its use, and a detailed user manual describing all the annotations that can be used to annotate class models in ModelCC can be found at the ModelCC web site: `http://www.modelcc.org`.

7 Conclusions and Future Work

In this paper, we have introduced ModelCC, a model-based tool for language specification. ModelCC lets language designers create explicit models of the concepts a language represents, i.e. the abstract syntax model (ASM) of the language. Then, that abstract syntax can be represented in textual or graphical form, using the concrete syntax defined by a concrete syntax model (CSM). ModelCC automates the ASM-CSM mapping by means of metadata annotations on the ASM, which let ModelCC act as a model-based parser generator.

ModelCC is not bound to particular scanning and parsing techniques, so that language designers do not have to tweak their models to comply with the constraints imposed by particular parsing algorithms. ModelCC abstracts away many details traditional language processing tools have to deal with. It cleanly separates language specification from language processing. Given the proper ASM-CSM mapping definition, ModelCC-generated parsers are able to automatically instantiate the ASM given an input string representing the ASM in a concrete syntax.

Apart from being able to deal with ambiguous languages, ModelCC also allows the declarative resolution of any language ambiguities by means of constraints defined over the ASM. The current version of ModelCC also supports lexical ambiguities and custom pattern matching classes.

ModelCC also incorporates reference resolution within the parsing process. Instead of returning abstract syntax trees, ModelCC is able to obtain abstract syntax graphs from its input string. Such abstract syntax graphs are not restricted to directed acyclic graphs, since ModelCC supports the resolution of anaphoric, cataphoric, and recursive references.

The proposed model-driven language specification approach promotes the domain-driven design of language processors. Its model-driven philosophy supports language evolution by improving the maintainability of languages processing system. It also facilitates the reuse of language specifications across product lines and different applications, eliminating the duplication required by conventional tools and improving the modularity of the resulting systems.

In the future, we plan to further study the possibilities tools such as ModelCC open up in different application domains, including traditional language processing systems (compilers and interpreters) [3], domain-specific languages and language workbenches [11], model-driven software development tools [27, 12], natural language processing [15], text mining applications [2], data integration [8], and information extraction [26].

Acknowledgements

Work partially supported by research project TIN2012-36951, "NOESIS: Network-Oriented Exploration, Simulation, and Induction System", funded by the Spanish Ministry of Economy and the European Regional Development Fund (FEDER).

References

[1] Harold Abelson & Gerald J. Sussman (1996): *Structure and Interpretation of Computer Programs*, 2nd edition. MIT Press.

[2] Charu C. Aggarwal & ChengXiang Zhai, editors (2012): *Mining Text Data*. Springer.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi & Jeffrey D. Ullman (2006): *Compilers: Principles, Techniques, and Tools*, 2nd edition. Addison Wesley.

[4] Fernando Berzal, Juan-Carlos Cubero, Nicolás Marín & María-Amparo Vila (2005): *Lazy Types: Automating Dynamic Strategy Selection*. *IEEE Software* 22(5), pp. 98–106.

[5] Elizabeth Bjarnason (1996): *APPLAB – A Laboratory for Application Languages*. In: *Proceedings of the 7th Nordic Workshop on Programming Environment Research*, pp. 99–104.

[6] Patrick Borras, Dominique Clement, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang & V. Pascual (1988): *CENTAUR: the system*. In: *Proceedings of the 3rd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 14–24.

[7] Christoff Bürger, Sven Karol, Christian Wende & Uwe Aßman (2010): *Reference Attributed Grammars for metamodel semantics*. In: *Proceedings of the 3rd International Conference on Software Language Engineering*, pp. 22–41.

[8] AnHai Doan, Alon Halevy & Zachary Ives (2012): *Principles of Data Integration*. Elsevier Science.

[9] Eric Evans (2003): *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[10] Martin Fowler (2002): *Using Metadata*. *IEEE Software* 19(6), pp. 13–17.

[11] Martin Fowler (2005): *Language Workbenches: The Killer-App for Domain Specific Languages?* Http://martinfowler.com/articles/languageWorkbench.html.

[12] Julián Garrido, M. Ángeles Martos & Fernando Berzal (2007): *Model-driven development using standard tools*. In: *Proceedings of the 9th International Conference on Enterprise Information Systems, IDEAL 2007* DISI, pp. 433–436.

[13] John Grundy, John Hosking, Karen Na Li, Norhayati Mohd Ali, Jun Huh & Richard Lei Lu (2013): *Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications*. *IEEE Transactions on Software Engineering* 39, pp. 487–515.

[14] Görel Hedin & Boris Magnusson (1988): *The Mjølner Environment: Direct Interaction with Abstractions*. In: *Proceedings of the 2nd European Conference on Object-Oriented Programming, Lecture Notes in Computer Science* 322, pp. 41–54.

[15] Daniel Jurafsky & James H. Martin (2009): *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*, 2nd edition. Prentice Hall.

[16] Lennart C. L. Kats, Eelco Visser & Guido Wachsmuth (2010): *Pure and declarative syntax definition: Paradise lost and regained*. In: *Proceedings of the ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'10)*, pp. 918–932.

[17] Anneke Kleppe (2007): *Towards the Generation of a Text-Based IDE from a Language Metamodel*. In: *Proceedings of the 4th European Conference on Model-Driven Architecture - Foundations and Applications, Lecture Notes in Computer Science* 4530, pp. 114–129.

[18] Rick Mugridge & Ward Cunningham (2005): *Fit for Developing Software: Framework for Integrated Tests (Robert C. Martin)*. Prentice Hall PTR.

[19] Luis Quesada, Fernando Berzal & Francisco J. Cortijo (2011): *Lamb — A Lexical Analyzer with Ambiguity Support*. In: *Proceedings of the 6th International Conference on Software and Data Technologies*, 1, pp. 297–300, doi:10.5220/0003476802970300.

[20] Luis Quesada, Fernando Berzal & Francisco J. Cortijo (2012): *Fence — A Context-Free Grammar Parser with Constraints for Model-Driven Language Specification*. In: *Proceedings of the 7th International Conference on Software Paradigm Trends*, pp. 5–13, doi:10.5220/0003949800050013.

[21] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2011): *A Language Specification Tool for Model-Based Parsing*. In: *Proceedings of the 12th International Conference on Intelligent Data Engineering and Automated Learning. Lecture Notes in Computer Science*, 6936, pp. 50–57, doi:10.1007/978-3-642-23878-9_7.

[22] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2014): *A Domain-Specific Language for Abstract Syntax Model to Concrete Syntax Model Mappings*. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pp. 158–165.

[23] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2014): *ModelCC — A Pragmatic Parser Generator*. *International Journal of Software Engineering and Knowledge*. (accepted for publication).

[24] Luis Quesada, Fernando Berzal & Juan-Carlos Cubero (2014): *Parsing Abstract Syntax Graphs with ModelCC*. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development*, pp. 151–157.

[25] Steven P. Reiss (1985): *PECAN: Program Development Systems that Support Multiple Views*. *IEEE Transactions on Software Engineering* SE-11(3), pp. 276–285.

[26] Sunita Sarawagi (2008): *Information Extraction*. *Foundations and Trends in Databases* 1(3), pp. 261–377. Available at `http://dx.doi.org/10.1561/1900000003`.

[27] Douglas C. Schmidt (2006): *Model-Driven Engineering*. *IEEE Computer* 39(2), pp. 25–31.

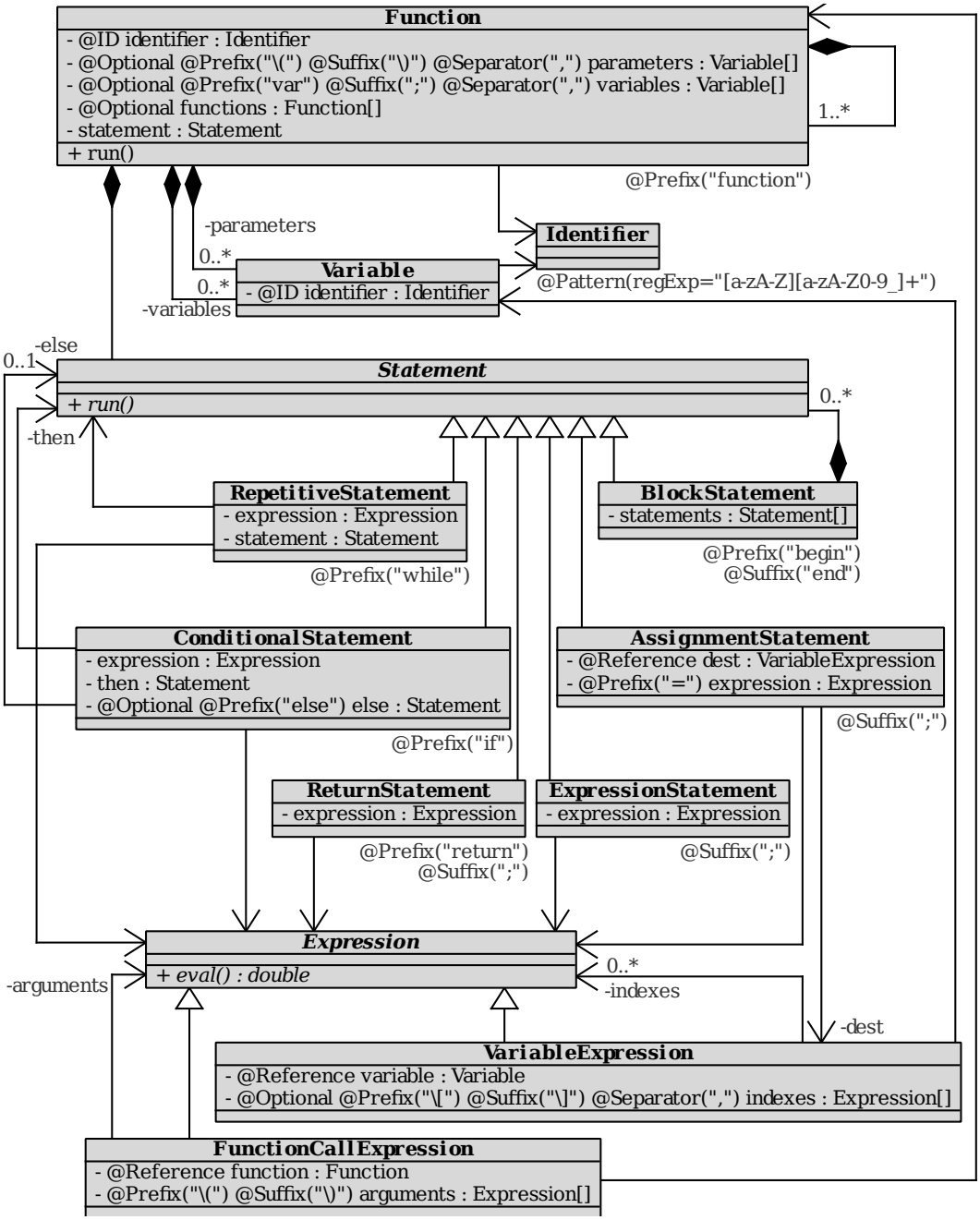


Figure 11: ModelCC specification of an imperative programming language. ModelCC reference resolution support is used to allow the declaration of variables and functions. ModelCC language composition support is used to include *Expressions*, which were defined as a separate language in Figure 6.

A Certified Reduction Strategy for Homological Image Processing (High-level Work)*

María Poza César Domínguez Jónathan Heras Julio Rubio

Department of Mathematics and Computer Science
University of La Rioja, Spain

{maria.poza, cedomin, jonathan.heras, julio.rubio}@unirioja.es

The analysis of digital images using homological procedures is an outstanding topic in the area of Computational Algebraic Topology. In this work, we describe a certified reduction strategy to deal with digital images, but preserving their homological properties. We stress both the advantages of our approach (mainly, the formalisation of the mathematics allowing us to verify the correctness of algorithms) and some limitations (related to the performance of the running systems inside proof assistants). The drawbacks are overcome using techniques that provide an integration of computation and deduction. Our driving application is a problem in bioinformatics, where the accuracy and reliability of computations are specially requested.

This paper will appear in ACM Transactions on Computational Logic, 2014, vol. 85, n. 3. The paper is also available at http://arxiv.org/abs/1306.0806.

*The work was partially supported by Ministerio de Educación y Ciencia project MTM2009-13842-C02-0114, and by the European Unions 7th Framework Programme under grant agreement nr. 243847.

Submitted to:
PROLE 2014

Lifting Term Rewriting Derivations in Constructor Systems by Using Generators (Original Work)*

Adrián Riesco
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es

Juan Rodríguez-Hortalá
Lambdoop Solutions
juan.rodriguez@lambdoop.com

Narrowing is a procedure that was first studied in the context of equational E-unification and that has been used in a wide range of applications. The classic completeness result due to Hullot states that any term rewriting derivation starting from an instance of an expression can be 'lifted' to a narrowing derivation, whenever the substitution employed is normalized. In this paper we adapt the generator-based extra-variables-elimination transformation used in functional-logic programming to overcome that limitation, so we are able to lift term rewriting derivations starting from arbitrary instances of expressions. The proposed technique is limited to constructor systems and to derivations reaching a ground expression. We also present a Maude-based implementation of the technique, using natural rewriting for the on-demand evaluation strategy.

1 Introduction

Narrowing [BN98] is a procedure that was first studied in the context of equational E-unification and that has been used in a wide range of applications [MT07, GHLR99]. Narrowing can be described as a modification of term rewriting in which matching is replaced by unification so, in a derivation starting from a goal expression, it is able to deduce the instantiation of the variables of the goal expression that is needed for the computation to progress. The key result for the completeness of narrowing w.r.t. term rewriting is *Hullot's lifting lemma* [Hul80], which states that any term rewriting derivation $e_1\theta \to^* e_2$ can be *lifted* into a narrowing derivation $e_1 \leadsto^*_\sigma e_3$ such that e_3 and σ are more general than e_2 and θ—w.r.t. to the usual instantiation preorder [BS01], and for the variables involved in the derivations—, provided that the starting substitution θ is normalized [MH94]. This latter condition is essential, so it is fairly easy to break Hullot's lifting lemma by dropping it: e.g. under the term rewriting system (TRS) $\{f(0,1) \to 2, coin \to 0, coin \to 1\}$ the term rewriting derivation $f(X,X)[X/coin] \to^* 2$ cannot be lifted by any narrowing derivation. Several variants and extensions of narrowing have been developed in order to improve that result under certain assumptions or for particular classes of term rewriting systems [MH94, MT07, DEE+11].

In this paper we show how to adapt the generator-based extra variable elimination transformation used in functional-logic programming (FLP) to drop the normalization condition required by Hullot's lifting lemma. The proposed technique is devised for constructor systems (CS's) with extra variables, and it is limited to derivations reaching a ground expression. To test the feasibility of this approach, we have also developed a prototype in Maude [CDE+07], relying on the natural rewriting on-demand strategy [Esc04] to obtain an effective operational procedure.

The rest of the paper is organized as follows. In Section 2 we introduce the semantics for CS's that we have used to formally prove the results, and that first suggested us the feasibility of the approach.

*Research supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04).

Submitted to:
PROLE 2014

In Section 3 we show our adaptation of the generators technique from FLP, and use the semantics for proving the adequacy of the technique for lifting term rewriting derivations reaching ground c-terms. In Section 4 we outline the implementation and commands of our prototype. Finally Section 5 concludes and outlines some lines of future work.

2 Preliminaries and formal setting

We mostly use the notation from [BN98], with some additions from [LRS09a]. We consider a first order signature $\Sigma = CS \uplus FS$, where CS and FS are two disjoint sets of *constructor* and defined *function* symbols respectively, all of them with associated arity. We use $c, d, \ldots$ for constructors, $f, g, \ldots$ for functions and $X, Y, \ldots$ for variables of a numerable set $\mathscr{V}$. The notation $\overline{o}$ stands for tuples of any kind of syntactic objects. The set Exp of *total expressions* is defined as $Exp \ni e ::= X \mid h(e_1, \ldots, e_n)$, where $X \in \mathscr{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set $CTerm$ of *total constructed terms* (or *c-terms*) is defined like Exp, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that Exp stands for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. We say that an expression e is *ground* iff no variable appears in e. We will frequently use *one-hole contexts*, defined as $Cntxt \ni \mathscr{C} ::= [\,] \mid h(e_1, \ldots, \mathscr{C}, \ldots, e_n)$.

Example 1 *We will use a simple example throughout this section to illustrate these definitions. Assume we want to represent the staff of a shop, so we have* $CS = \{madrid^0, vigo^0, man^0, woman^0, pepe^0, luis^0, pilar^0, maria^0, e^2, p^2\}$*, where e will be the constructor for employees and p the constructor for pairs, and* $FS = \{branches^0, search^1, employees^1\}$*. Using this signature, we can build the set* $Exp = \{madrid, vigo, employees\ (madrid), p(pilar, X), \ldots\}$*. From this set, we have* $CTerm = \{madrid, vigo, p(pilar, X), \ldots\}$*, while the ground terms are* $\{employees(madrid), madrid, vigo, \ldots\}$*. Finally, a possible one-hole context is* $p([\,], X)$*.*

We also consider the extended signature $\Sigma_\bot = \Sigma \cup \{\bot\}$, where $\bot$ is a new 0-arity constructor symbol that does not appear in programs and which stands for the undefined value. Over this signature we define the sets $Exp_\bot$ and $CTerm_\bot$ of *partial* expressions and c-terms, respectively. The intended meaning is that Exp and $Exp_\bot$ stand for evaluable expressions, i.e., expressions that can contain function symbols, while $CTerm$ and $CTerm_\bot$ stand for data terms representing total and partial values, respectively. Partial expressions are ordered by the *approximation* ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow \mathscr{C}[e] \sqsubseteq \mathscr{C}[e']$ for all $e, e' \in Exp_\bot, \mathscr{C} \in Cntxt$. The *shell* $|e|$ of an expression e represents the outer constructed part of e and is defined as: $|X| = X$; $|c(e_1, \ldots, e_n)| = c(|e_1|, \ldots, |e_n|)$; $|f(e_1, \ldots, e_n)| = \bot$. It is trivial to check that for any expression e we have $|e| \in CTerm_\bot$, that any total expression is maximal w.r.t. $\sqsubseteq$, and that as consequence if t is total then $t \sqsubseteq |e|$ implies $t = e$.

Example 2 *Using the signature from Example 1, we have* $employees(\bot) \in Exp_\bot$*,* $p(\bot, X) \in CTerm_\bot$*, and* $|p(search(branches), X)| = p(\bot, X)$*.*

Substitutions $\theta \in Subst$ are finite mappings $\theta : \mathscr{V} \longrightarrow Exp$, extending naturally to $\theta : Exp \longrightarrow Exp$. We write ε for the identity (or empty) substitution. We write $e\theta$ to apply of θ to e, and $\theta\theta'$ for the composition, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain and variable range of θ are defined as $dom(\theta) = \{X \in \mathscr{V} \mid X\theta \neq X\}$ and $vran(\theta) = \bigcup_{X \in dom(\theta)} var(X\theta)$. By $[X_1/e_1, \ldots, X_n/e_n]$ we denote a substitution σ such that $dom(\sigma) = \{X_1, \ldots, X_n\}$ and $\forall i.\sigma(X_i) = e_i$. If $dom(\theta_0) \cap dom(\theta_1) = \emptyset$, their disjoint union $\theta_0 \uplus \theta_1$ is defined by $(\theta_0 \uplus \theta_1)(X) = \theta_i(X)$, if $X \in dom(\theta_i)$ for some θ_i; $(\theta_0 \uplus \theta_1)(X) = X$ otherwise. Given

$W \subseteq \mathscr{V}$ we write $\theta|_W$ for the restriction of θ to W, i.e. $(\theta|_W)(X) = \theta(X)$ if $X \in W$, and $(\theta|_W)(X) = X$ otherwise; we use $\theta|_{\setminus D}$ as a shortcut for $\theta|_{(\mathscr{V}\setminus D)}$. *C-substitutions* $\theta \in CSubst$ verify that $X\theta \in CTerm$ for all $X \in dom(\theta)$. We say a substitution σ is ground iff $vran(\sigma) = \emptyset$, i.e. $\forall X \in dom(\sigma)$ we have that $\sigma(X)$ is ground. The sets $Subst_\perp$ and $CSubst_\perp$ of partial substitutions and partial c-substitutions are the sets of finite mappings from variables to partial expressions and partial c-terms, respectively.

Example 3 *Using the signature from Example 1, we can define the C-substitutions* $\theta_1 \equiv X/woman$, $\theta_2 \equiv X/man$, *and* $\theta_3 \equiv Y/pilar$. *We can define the restrictions* $\theta_1|_{\{X\}} = \theta_1$ *and* $\theta_1|_{\setminus\{X\}} = \varepsilon$. *Finally, given the expression* $p(X,Y)$ *we have* $p(X,Y)\theta_1\theta_2 = p(woman,Y)$ *and* $p(X,Y)\theta_1\theta_3 = p(X,Y)\theta_3\theta_1 = p(woman,pilar)$.

A left-linear constructor-based term rewriting system or just *constructor system* or *program* $\mathscr{P}$ (*CS*) is a set of c-rewrite rules of the form $f(\bar{t}) \to r$ where $f \in FS^n$, $r \in Exp$ and $\bar{t}$ is a linear n-tuple of c-terms, where linearity means that variables occur only once in $\bar{t}$. Notice that we allow r to contain so called *extra variables*, i.e., variables not occurring in $f(\bar{t})$. To be precise, we say that $X \in \mathscr{V}$ is an extra variable in the rule $l \to r$ iff $X \in var(r) \setminus var(l)$, and by $vExtra(R)$ we denote the set of extra variables in a program rule R. We assume that every CS contains the rules $\mathscr{Q} = \{X\ ?\ Y \to X, X\ ?\ Y \to Y\}$, defining the behavior of $? \in FS^2$, used in infix mode, and that those are the only rules for ?. Besides, ? is right-associative so $e_1\ ?\ e_2\ ?\ e_3$ is equivalent to $e_1\ ?\ (e_2\ ?\ e_3)$. For the sake of conciseness we will often omit these rules when presenting a CS. A consequence of this is that we only consider non-confluent programs. Given a TRS $\mathscr{P}$, its associated *term rewriting relation* $\to_{\mathscr{P}}$ is defined as: $\mathscr{C}[l\sigma] \to_{\mathscr{P}} \mathscr{C}[r\sigma]$ for any context $\mathscr{C}$, rule $l \to r \in \mathscr{P}$ and $\sigma \in Subst$. We write $\overset{*}{\to}_{\mathscr{P}}$ for the reflexive and transitive closure of the relation $\to_{\mathscr{P}}$. We will usually omit the reference to $\mathscr{P}$ or denote it by $\mathscr{P} \vdash e \to e'$ and $\mathscr{P} \vdash e \to^* e'$.

Example 4 *Using the signature from Example 1, we can describe the following constructor-based term rewriting system:*

$$\begin{array}{lcl} branches & \to & madrid\ ?\ vigo \\ employees(madrid) & \to & e(pepe,men) \\ employees(madrid) & \to & e(maria,men) \\ employees(vigo) & \to & e(pilar,women)\ ?\ e(luis,men) \\ search(e(N,S)) & \to & p(N,N) \end{array}$$

In this example, the function symbol branches defines the different branches of the company, employees defines the employees in each branch (built with the constructor symbol e), and search returns a pair of names, built with the constructor symbol p. Note that several different notations are possible; for example, it is possible to define the employees of one branch by using just one rule and the ? *operator or just several different rules with the same lefthand side.*

2.1 A proof calculus for constructor systems with extra variables

In [LRS09a] an adequate semantics for reachability of c-terms by term rewriting in CS's was presented. The key idea there was using a suitable notion of value, in this case the notion of s-cterm. *SCTerm* is the set of s-cterms, which are *finite* sets of elemental s-cterms, while the set *ESCTerm* of elemental s-cterms is defined as $ESCTerm \ni est ::= X \mid c(st_1,\ldots,st_n)$ for $X \in \mathscr{V}$, $c \in CS^n$, $st_1,\ldots,st_n \in SCTerm$. We extend this idea to expressions obtaining the sets *SExp* of s-expressions or just s-exp, and *ESExp* of elemental s-expressions, which are defined the same but now using any symbol in Σ in applications instead of just constructor symbols. Note that the s-expression $\emptyset$ corresponds to $\perp$, so s-exps are partial by default. The approximation preorder $\sqsubseteq$ is defined for s-exps as the least preorder such that $se \sqsubseteq se'$ iff

E	$se \twoheadrightarrow \emptyset$	
RR	$\{X\} \twoheadrightarrow \{X\}$	if $X \in \mathscr{V}$
DC	$\dfrac{se_1 \twoheadrightarrow st_1 \ \ldots \ se_n \twoheadrightarrow st_n}{\{c(se_1,\ldots,se_n)\} \twoheadrightarrow \{c(st_1,\ldots,st_n)\}}$	if $c \in CS$
MORE	$\dfrac{se \twoheadrightarrow st_1 \ \ldots se \twoheadrightarrow st_n}{se \twoheadrightarrow st_1 \cup \ldots \cup st_n}$	
LESS	$\dfrac{\{esa_1\} \twoheadrightarrow st_1 \ \ldots \ \{esa_m\} \twoheadrightarrow st_m}{\{ese_1,\ldots,ese_n\} \twoheadrightarrow st_1 \cup \ldots \cup st_m}$	if $n \geq 2, m > 0$, for any $\{esa_1,\ldots,esa_m\} \subseteq \{ese_1,\ldots,ese_n\}$
ROR	$\dfrac{se_1 \twoheadrightarrow \widetilde{p_1}\theta \ \ldots \ se_n \twoheadrightarrow \widetilde{p_n}\theta \ \ \widetilde{r}\theta \twoheadrightarrow st}{\{f(se_1,\ldots,se_n)\} \twoheadrightarrow st}$	if $(f(p_1,\ldots,p_n) \to r) \in \mathscr{P}$, $\theta \in SCSubst$

Figure 1: A proof calculus for constructor systems

$\forall ese \in se.\exists ese' \in se'$ such that $ese \sqsubseteq ese'$, $X \sqsubseteq X$ for any $X \in \mathscr{V}$, and $h(se_1,\ldots,se_n) \sqsubseteq h(se'_1,\ldots,se'_n)$ iff $\forall i.se_i \sqsubseteq se'_i$.

Example 5 *Using the signature from Example 1, and given the s-cterm* $sct \equiv e(\{pepe,pilar\}, \{men, women\})$*, we have* $sct \in ESCTerm$*, while* $\{sct\} \in SCTerm$*. Similarly, given the es-exp* $esex \equiv employees(\{madrid,vigo\})$ *we have* $esex \in ESExp$ *and* $esex \notin ESCTerm$*. Finally, we have that* $\{esex\} \in SExp$*.*

The sets *SSubst* and *SCSubst* of s-substitutions and s-csubstitutions (or just s-csubst) consist of finite mappings from variables to s-exps or s-cterms, respectively. We extend s-substs to be applied to *ESExp* and *SExp* as $\sigma : ESExp \to SExp$ defined by $X\sigma = \sigma(X)$, $h(\overline{se})\sigma = \{h(\overline{se\sigma})\}$; and $\sigma : SExp \to SExp$ defined by $se\sigma = \bigcup_{ese \in se} ese\sigma$. The approximation preorder $\sqsubseteq$ is defined for s-substs as $\sigma \sqsubseteq \theta$ iff $\forall X \in \mathscr{V}.\sigma(X) \sqsubseteq \theta(X)$. For any nonempty and finite set $\{\theta_1,\ldots,\theta_n\} \subseteq SCSubst$ we define $\bigcup\{\theta_1,\ldots,\theta_n\} \in SCSubst$ as $\bigcup\{\theta_1,\ldots,\theta_n\}(X) = \theta_1(X) \cup \ldots \cup \theta_n(X)$.

Example 6 *Using the signature from Example 1, we can define the s-csubstitution* $\sigma \equiv \{X/\{pepe,pilar\}, Y/\{men,women\}\} \in SCSubst$*. Hence, given* $esex \equiv e(\{X\},\{Y\}) \in ESExp$ *we have that* $esex\sigma \equiv e(\{pepe, pilar\}, \{men,women\})$*.*

We obtain the denotation of an expression as the denotation of its associated s-expression, assigned by the operator $\widetilde{\ }: Exp_\perp \to SExp$, defined as $\widetilde{\perp} = \emptyset$; $\widetilde{X} = \{X\}$ for any $X \in \mathscr{V}$; $\widetilde{h(e_1,\ldots,e_n)} = \{h(\widetilde{e_1},\ldots,\widetilde{e_n})\}$ for any $h \in \Sigma^n$. The operator $\widetilde{\ }$ is extended to s-substitutions as $\widetilde{\sigma}(X) = \widetilde{\sigma(X)}$, for $\sigma \in Subst_\perp$. It is easy to check that $\widetilde{e\sigma} = \widetilde{e}\widetilde{\sigma}$ (see [LRS09a]). Conversely, we can flatten an s-expression *se* to obtain the set $flat(se)$ of expressions "contained" in it, so $flat(\emptyset) = \{\perp\}$ and $flat(se) = \bigcup_{ese \in se} flat(ese)$ if $se \neq \emptyset$, where the flattening of elemental s-exps is defined as $flat(X) = \{X\}$; $flat(h(se_1,\ldots,se_n)) = \{h(e_1,\ldots,e_n) \mid e_i \in flat(se_i)$ for $i = 1..n\}$.

Example 7 *Using the signature from Example 1, we have that* $\widetilde{p(X,Y)} = \{p(\{X\},\{Y\})\}$ *and* $flat(\{p(\{X\}, \{Y,Z\})\}) = \{p(X,Y), p(X,Z)\}$

In Figure 1 we can find the proof calculus that defines the semantics of s-expressions. Our proof calculus proves reduction statements of the form $se \twoheadrightarrow st$ with $se \in SExp$ and $st \in SCTerm$, expressing that *st* represents an approximation to one of the possible structured sets of values for *se*. We refer the interested reader to [LRS09a] for detailed explanations about the calculus.We write $\mathscr{P} \vdash se \twoheadrightarrow st$ to

express that $se \twoheadrightarrow st$ is derivable in our calculus under the CS $\mathscr{P}$. We say that a proof for a statement $\mathscr{P} \vdash se \twoheadrightarrow st$ is ground iff se, st and all the s-exp in the premises are ground. The *denotation* of an s-expression se under a CS $\mathscr{P}$ is defined as $[\![se]\!]^{\mathscr{P}} = \{st \in SCTerm \mid \mathscr{P} \vdash se \twoheadrightarrow st\}$, so $[\![e]\!]^{\mathscr{P}} = [\![\widetilde{e}]\!]^{\mathscr{P}}$. In the following we will usually omit the reference to $\mathscr{P}$. The denotation of $\sigma \in SSubst$ is defined as $[\![\sigma]\!] = \{\theta \in SCSubst \mid \forall X \in \mathscr{V}, \sigma(X) \twoheadrightarrow \theta(X)\}$, so for $\theta \in Subst_\bot$ we define $[\![\theta]\!] = [\![\widetilde{\theta}]\!]$.

Example 8 *Using the signature from Example 1 and the rules from Example 4, we have employees*$(\{X\})$ $\twoheadrightarrow \{e(pepe,men)\}$*, given the substitution* $X/\{madrid\}$.

The setting presented in [LRS09a] was not able to deal with extra variables. As programs with extra variables are very common when using narrowing, for this work we decided to extend the setting to deal with them. But then we realized that the semantics was already prepared to deal with extra variables, as the rule **ROR** from Figure 1 allows to instantiate extra variables freely with s-cterms: therefore all that was left was proving the adecuacy of the semantics in this extended scenario. Nevertheless, as a consequence of the freely instantiation of extra variables in **ROR**, then every program with extra variables turns into non-deterministic. For example consider a program $\{f \to (X,X)\}$ for which the constructors $0,1 \in CS^0$ are available, then we can do:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{0\} \twoheadrightarrow \{0\}}\,\textbf{DC}}{\{0,1\} \twoheadrightarrow \{0\}}\,\textbf{LESS} \quad \dfrac{\ldots}{\{0,1\} \twoheadrightarrow \{1\}}}{\widetilde{(X,X)}[X/\{0,1\}] = \{(\{0,1\},\{0,1\})\} \twoheadrightarrow \{(\{0\},\{1\})\}}\,\textbf{DC}}{\widetilde{f} = \{f\} \twoheadrightarrow \{(\{0\},\{1\})\} = \widetilde{(0,1)}}\,\textbf{ROR}$$

But in fact this is not very surprising, and it has to do with the relation between non-determinism and extra variables [AH06], but adapted to the run-time choice semantics [Hus93, Rod08] induced by term rewriting. As a consequence of this we assume that all the programs contain the function ?, so we only consider non-confluent TRS's. We admit that this is a limitation of our setting, but we also conjecture that for confluent TRS's a simpler semantics could be used, for which the packing of alternatives of c-terms would not be needed. Anyway, the point is that having ? at one's disposal is enough to express the non-determinism of any program [Han05], so we can use it to define the transformation $\widehat{\ }$ from s-exp and elemental s-exp to partial expressions that, contrary to *flat*, now takes care of the keeping the nested set structure by means of uses of the ? function. Then $\widehat{\ }: ESExp \to Exp_\bot$ is defined by $\widehat{X} = X$, $\widehat{h(se_1,\ldots,se_n)} = h(\widehat{se_1},\ldots,\widehat{se_n})$; and $\widehat{\ }: SExp \to Exp_\bot$ is defined by $\widehat{\emptyset} = \bot$, $\widehat{\{ese_1,\ldots,ese_n\}} = \widehat{ese_1}$? … ? $\widehat{ese_n}$ for $n > 0$, where in the case for $\widehat{\{ese_1,\ldots,ese_n\}}$ we use some fixed arbitrary order on terms in the line of Prolog [SS86] for arranging the arguments of ?. This operator is also overloaded for substitutions as $\widehat{\ }: SSubst \to Subst_\bot$ as $(\widehat{\sigma})(X) = \widehat{\sigma(X)}$. Thanks to the power of ? to express non-determinism, that transformation preserves the semantics from Figure 1, and we can use it to prove the following new result about the adequacy of the semantics for programs with extra variables—see [RR12a] for a detailed proof.

Theorem 1 (Adequacy of $[\![\_]\!]$) *For all* $e,e' \in Exp, t \in CTerm_\bot, st \in SCTerm$:
Soundness *$st \in [\![\widetilde{e}]\!]$ and $t \in flat(st)$ implies $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$. Therefore, $\widetilde{t} \in [\![\widetilde{e}]\!]$ implies $e \to^* e'$ for some $e' \in Exp$ such that $t \sqsubseteq |e'|$. Besides, in any of the previous cases, if t is total then $e \to^* t$.*
Completeness *$e \to^* e'$ implies $\widetilde{|e'|} \in [\![\widetilde{e}]\!]$. Hence, if t is total then $e \to^* t$ implies $\widetilde{t} \in [\![\widetilde{e}]\!]$.*

We refer the interested reader to [LRS09a] and [LRS09b] (Theorems 2 and 3) for more properties of $[\![\_]\!]$ like compositionality or monotonicity, some of which are used in the proofs for the results in the

$\_ \vdash \_ \lessdot \_ \subseteq CS \times SCTerm \times Exp$	$\_ \vdash \_ \lessdot \_ \subseteq CS \times ESCTerm \times Exp$
$\mathscr{P} \vdash st \lessdot e$ if $\forall est \in st, \mathscr{P} \vdash est \lessdot e$	$\mathscr{P} \vdash X \lessdot e$ if $\mathscr{P} \vdash e \to^* X$
	$\mathscr{P} \vdash c(\overline{st}) \lessdot e$ if $\mathscr{P} \vdash e \to^* c(\overline{e})$ for some $\overline{e}$ such that $\forall e_i \in \overline{e}, \mathscr{P} \vdash st_i \lessdot e_i$

Figure 2: Domination relation

present paper. There is another characterization of $[\![\_]\!]$ closer to term rewriting which is based of the *domination relation* $\_ \lessdot \_$ presented in Figure 2 (we will omit the prefix "$\mathscr{P} \vdash$" when it is implied by the context). With this relation we try to transfer to the rewriting world the finer distinction between sets of values that the structured representation of *SCTerm* allows us to perform. We extend the relation $\_ \lessdot \_$ to $\_ \vdash \_ \lessdot \_ \subseteq CS \times SCSubst \times Subst$ by $\theta \lessdot \sigma$ iff $\forall X \in \mathscr{V}, \theta(X) \lessdot \sigma(X)$. As can be seen in [LRS09b], this relation is a key ingredient to prove the soundness of $[\![\_]\!]$, and its equivalence to $[\![\_]\!]$ is stated in the following result.

Lemma 1 (Domination) *For all $e \in Exp, st \in SCTerm$, $st \in [\![\widetilde{e}]\!]$ iff $st \lessdot e$. Besides, regarding substitutions, for all $\sigma \in Subst$, $\theta \in SCSubst$ we have that $\theta \in [\![\widetilde{\sigma}]\!]$ iff $\theta \lessdot \sigma$.*

3 The generators approach

In this section we will show a proposal for adapting the generators technique from the field of functional-logic programming [DL07, AH06] to the lifting of term rewriting derivations from arbitrary instances of expressions. This technique consists in replacing free and extra variables by a call to a *generator function* that can be reduced to any ground c-term. The generator function *gen* is defined as follows:

Definition 1 (Generator function) *For any program $\mathscr{P}$ we can define a fresh function gen as follows: for each $c \in CS^n$ we add a new rule $gen \to c(gen, \ldots, gen)$ to the program. By $\mathscr{G}$ we denote the program that consists of the set of rules for gen.*

Example 9 *Given the system in Example 4, the rules for gen are $\mathscr{G} \equiv \{gen \to madrid, gen \to vigo, gen \to pepe, gen \to luis, gen \to maria, gen \to pilar, gen \to men, gen \to women,\ gen \to e(gen,\ gen),\ gen \to p(gen, gen)\}$.*

The point with *gen* is that we can use it to compute any *ground value*:

Proposition 1 *For all $t \in CTerm$, $st \in SCTerm$ and $\theta \in SCSubst$ such that those are ground we have $gen \to^* t$, $st \in [\![gen]\!]$ and $\theta \in [\![[\overline{X/gen}]]\!]$ for $\overline{X} = dom(\theta)$.*

Then the main idea with generators is that given some $e \in Exp$ with $var(e) = \overline{X}$, we can simulate narrowing with e by performing term rewriting with $e[\overline{X/gen}]$. As *gen* can be reduced to any ground s-cterm, then Lemma 1 from [LRS09a] suggests that this procedure will be able to lift derivations $e\sigma \to^* t$ with an arbitrary $\sigma \in Subst$, even those which are not normalized: e.g. we can easily apply this technique to the example in Section 1, getting $f(X,X)[X/gen] \to^* f(0,1) \to 2$. Sadly, on the other hand, only derivations reaching a ground c-term will be lifted, and the reason for that is that *gen* can be reduced to an arbitrary ground c-term, but it cannot be reduced to any c-term with variables. Thus, under the program $\{g(c(X)) \to X\}$ the term rewriting derivation $g(Y)[Y/c(X)] \to X$ cannot be lifted by using generators, as $g(Y)[Y/gen] \to g(c(gen)) \to gen \not\to^* X$, even though $[Y/c(X)]$ is a normalized substitution.

In order to prove the completeness of the generators technique for the reachability of ground c-terms, we rely on the following modification of Lemma 1 from [LRS09a].

Lemma 2 *For all $\sigma \in SSubst$, $se \in SExp$, $st \in SCTerm$, if st is ground then $se\sigma \twoheadrightarrow st$ implies $\exists \theta \in [\![\sigma]\!]$ such that $se\theta \twoheadrightarrow st$, θ is ground and $dom(\theta) = dom(\sigma)$.*

Note the restriction to ground s-cterms in Lemma 2 is crucial, and that it reflects the lack of completeness for reaching non-ground c-terms of the generators technique: e.g. under the program $\{f \to c(X)\}$ using $se = \{Y\}$, $\sigma = [Y/\{f\}]$ and $st = \{c(\{X\})\}$ the only $\theta \in [\![[Y/\{f\}]]\!]$ fulfilling the first condition is $\theta = [Y/\{c(\{X\})\}]$, which is not ground. On the other hand those s-csubst obtained by Lemma 2 are ground, and so they are in the denotation of an appropriate substitution with only generators in its range.

Generators can be introduced in programs systematically in order to eliminate extra variables from program rules using a program transformation in the line of those from [DL07, AH06]. In those works the usual call-time choice semantics for functional-logic programming [GHLR99] was adopted, therefore we use a different transformation that is adapted to the use of term rewriting, which leads to a different set of reachable c-terms than that obtained with call-time choice [Rod08]. The point in eliminating extra variables is that in this way we eliminate the "oracular guessing" that is performing in a term rewriting step using extra variables: by this guessing we refer for example to the instantiation performed under the program $\{f \to g(X), g(0) \to 1\}$ in the first step of the derivation $f \to g(0) \to 1$ for the extra variable X, that has to be instantiated with 0 in order for the derivation to continue. That, combined with a suitable on-demand evaluation strategy like natural rewriting [Esc04], turns term rewriting with generators into an effective mechanism for lifting term rewriting derivations. We formalize our extra variable elimination transformation through the following definition.

Definition 2 (Generators program transformation)
Given a program $\mathscr{P}$ its transformation $\hat{\mathscr{P}}$ consists of the rules $\mathscr{G}$ for gen together with the transformation of each rule in $\mathscr{P}$, defined as $\widehat{f(p_1,\ldots,p_n) \to r} = f(p_1,\ldots,p_n) \to r[\overline{X/gen}]$, where $\overline{X} = vExtra(f(p_1,\ldots,p_n) \to r)$.

Then it is clear that for any program $\mathscr{P}$ its transformation $\hat{\mathscr{P}}$ does not have any extra variable in its rules. Note that, contrary to the proposals from [DL07, AH06], this transformation destroys the sharing that normally appears when there are several occurrences of the same variable, in procedures that instantiate variables like narrowing or SLD resolution. In our transformation, however, once instantiated with *gen* every occurrence of the same variable evolves independently. This is needed to ensure completeness under the transformed program, which can be seen considering the program $\mathscr{P} = \{f \to (g(X), h(X), g(0) \to 1, h(1) \to 2\}$ and the derivation $\mathscr{P} \vdash f \to (g(0\ ?\ 1), h(0\ ?\ 1)) \to^* (g(0), h(1)) \to^* (1,2)$: as extra variables can be instantiated with arbitrary expressions that implies that in particular those can be instantiated with "alternatives" of expressions built using the ? function, which can evolve independently after the alternative between them is resolved. We can lift that derivation with our transformation as $\hat{\mathscr{P}} \vdash f \to (g(gen), h(gen)) \to^* (g(0), h(1)) \to^* (1,2)$. The adequacy of the transformation is formulated in the following result, in the same terms as the variable elimination result from [DL07].

Theorem 2 *For any program $\mathscr{P}$, $se \in SExp$, $st \in SCTerm$ if st is ground then $\mathscr{G} \uplus \mathscr{P} \vdash se \twoheadrightarrow st$ iff $\hat{\mathscr{P}} \vdash se \twoheadrightarrow st$.*

After eliminating extra variables with the proposed program transformation, we can then emulate the instantiation of variables performed by a narrowing procedure by just replacing free variables with *gen*, thus lifting any term rewriting derivation starting from an arbitrary instance of an expression to a ground c-term.

Theorem 3 (Lifting) *For any program $\mathscr{P}$, $e, e' \in Exp$ such that e' is ground:*
Soundness *$\hat{\mathscr{P}} \vdash e[\overline{X/gen}] \to^* e'$ implies $\exists \sigma \in Subst$ such that $\mathscr{P} \vdash e\sigma \to^* e''$ for some $e'' \in Exp$ such that $|e'| \sqsubseteq |e''|$ with $dom(\sigma) = \overline{X}$. As a consequence, if $e' = t \in CTerm$ then $\mathscr{P} \vdash e\sigma \to^* t$.*
Completeness *For any $\sigma \in Subst$ we have that $\mathscr{P} \vdash e\sigma \to^* e'$ implies $\hat{\mathscr{P}} \vdash e[\overline{X/gen}] \to^* e''$ for some $e'' \in Exp$ such that $|e'| \sqsubseteq |e''|$ with $\overline{X} = dom(\sigma)$. As a consequence, if $e' = t \in CTerm$ then $\hat{\mathscr{P}} \vdash e[\overline{X/gen}] \to^* t$.*

4 Maude prototype

We present in this section our prototype; much more information can be found at `http://gpd.sip.ucm.es/snarrowing`. The prototype is started by typing `loop init-s .`, that initiates an input/output loop where programs and commands can be introduced. These programs have syntax `smod NAME is STMNTS ends`, where `NAME` is the identifier of the program and `STMNTS` is a sequence of constructor-based left-linear rewrite rules. For instance, Example 4 would be written as follows:

```
(smod CLERKS is
  branches -> madrid ? vigo .
  employees(madrid) -> e(pepe, men) .
  employees(madrid) -> e(maria, men) .
  employees(vigo) -> e(pilar, women) ? e(luis, men) .
  search(e(N,S)) -> p(N,N) .
ends)
```

where upper-case letters are assumed to be variables. We can evaluate terms with variables with the command `eval-gen`, that transforms each variable in the term into the `gen` constant described above and evaluates the thus obtained expression in the module extended with the `gen` rules:

```
Maude> (eval-gen search(X,X) .)
Result: p(madrid, madrid)
```

That is, the tool first finds a result with the same value for the two elements of the pair. We can ask the system for more solutions with the `next` command until no more solutions are found, which will reveal pairs with different values:

```
Maude> (next .)
Result: p(madrid,vigo)
```

Finally, the system combines the on-demand strategy with two different search strategies: depth-first and breadth-first, and allows the user to check the trace in order to see how the generators are instantiated. We will show in the following section how to use these commands.

4.1 Looking for alternatives

We present here a more complex example, which introduces how to use our tool to search for different paths leading to the solution. This example presents a simplified version of the intruder protocol introduced in [RR12b], which is also executable with the generators approach presented here and is available at `http://gpd.sip.ucm.es/snarrowing`.

The module `PARTY` below describes the specification of a party. Our goal in this party is to have fun, so we define the function `success`, which receives a set of friends `F` and a set of elements that we already have. It is reduced to the function `haveFun` applied to the set obtained after calling to our friends:

```
(smod PARTY is

  success(F, S) -> haveFun?(makeCalls(F, S)) .
```

The function `haveFun` is reduced to `tt` (standing for the value `true`) when it receives the constant `fun`:

```
haveFun(fun) -> tt .
```

The function `makeCalls` combines the current items with the ones obtained by making further calls using the new items obtained by offering your items to your friends:

```
  makeCalls(F, S) -> S ? makeCalls(F, makeAnOffer(F, S)) .
```

We can reach different results by using `makeAnOffer`. First, it is possible to combine the current items to obtain a new one:

```
  makeAnOffer(F, S) -> combine(S, S) .
```

This combination, achieved by the `combine` function, generates a `burger` from `bread` and `meat`, and `fun` when a `burger` and a `videogame` are found:

```
  combine(bread, meat) -> burger .
  combine(burger, videogames) -> fun .
```

Another possibility is to call a friend and show him the items we have obtained so far:

```
  makeAnOffer(F, S) -> call(F, S) .
```

This `call` depends on the friend we call. We present below the different possibilities:

```
  call(enrique, drink) -> music .
  call(adri, meat) -> bread .
  call(rober, music) -> videogames .
  call(nacho, videogames) -> music .
  call(juan, food) -> drink .
ends)
```

Once this module is loaded into the interpreter, we indicate that we want to activate the path. In this way, we can explore the different ways to reach the values:

```
Maude> (path on.)
Path activated.
```

We also set the exploration strategy to *breadth first*, so the tool finds the shortest solutions first:

```
Maude> (breadth-first .)
Breadth-first strategy selected.
```

We can now look for solutions to the `success` function, using a variable as argument:

```
Maude> (eval-gen success(F, S) .)
Result: tt
```

We can now examine the path traversed by the tool to reach the result as follows:

```
Maude> (show path .)
haveFun(makeCalls(gen,gen))
--->
haveFun(gen ? makeCalls(gen,makeAnOffer(gen,gen)))
--->
haveFun(gen)
--->
haveFun(fun)
--->
tt
```

It shows how it simply requires start with `fun` to obtain `fun` at the end. Since this answer is not useful we look for the next one:

```
Maude> (next .)
Result: tt

Maude> (show path .)
haveFun(makeCalls(gen,gen))
--->
haveFun(gen ? makeCalls(gen,makeAnOffer(gen,gen)))
--->
haveFun(makeCalls(gen,makeAnOffer(gen,gen)))
--->
haveFun(makeAnOffer(people,gen) ?
        makeCalls(makeAnOffer(people,makeAnOffer(people,gen))))
--->
haveFun(makeAnOffer(people,gen))
--->
haveFun(combine(gen,gen))
--->
haveFun(combine(burger,gen))
--->
haveFun(combine(burger,videogames))
--->
haveFun(fun)
--->
tt
```

In this case we would require to start having a `burger` and `videogames`, so they can be combined in order to reach the `fun`. In this case no friends were required. However, the next search (where we just show the last steps) requires a `burger`, `music`, and our friend `rober`:

```
...
haveFun(combine(gen,call(gen,gen)))
--->
haveFun(combine(burger,call(gen,gen)))
--->
haveFun(combine(burger,call(rober,gen)))
--->
haveFun(combine(burger,call(rober,music)))
--->
haveFun(combine(burger,videogames))
```

```
--->
haveFun(fun)
--->
tt
```

We can keep looking for more results until we find the one we are looking for or we reach the limit on the number of steps (which can be modified by means of the `depth` command).

4.2 Implementation notes

We have implemented our prototype in Maude [CDE+07], a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in *rewriting logic* [Mes92], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude *functional modules* correspond to specifications in *membership equational logic* [BJM00], which, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. This logic is a good semantic framework for formally specifying programming languages as rewrite theories [MR07]; since Maude specifications are executable, we obtain an interpreter for the language being specified.

Exploiting the fact that rewriting logic is reflective [CMP07], an important feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [CDE+07, Chapter 14], a characteristic that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. In this way, we define the syntax of the modules introduced by the user, manipulate them, direct the evaluation of the terms (by using the on-demand strategy natural narrowing [Esc04]), and implement the input/output interactions in Maude itself.

5 Concluding remarks and ongoing work

In this work we have proposed and formally proved the adequacy of a technique for lifting term rewriting derivations from an arbitrary instance of an expression to a constructed term—or the outer constructed part of any expression—using constructor systems. It is based on the generator technique from the field of functional-logic programming [DL07, AH06], but adapted to the different semantic context of term rewriting [Rod08]. For proving the adequacy of the proposed technique we have employed the semantics for constructor systems defined in [LRS09a] as the main technical tool. This way we have put the semantics in practice by using it for solving a technical problem that wasn't stated in the original paper. Along the way we have extended the semantics to support extra variables in rewriting rules, as those are very frequent when using narrowing, which is the context of the present paper. To do that we have made the necessary adjustments to the formulation of the semantics and to the proofs for its properties.

A fundamental limitation of the generators is that it can be only used for reaching ground c-terms or the outer constructed part of expressions. This limitation can be somewhat mitigated by reducing the reachability to a non-ground value to the reachability of a ground value: for example to test for $e \to^* c(X)$ we can define a new function f by the rule $f(c(X)) \to \mathit{true}$ and then test for $f(e) \to^* \mathit{true}$. Anyway this is a partial solution, and moreover the instantiation of free variables corresponding to the evaluation of *gen* cannot be obtained by a transformation in that line, for example by evaluating $(f(X), X)$ in the previous

example, due to the aforementioned loss of sharing between different occurrences of the same variable. This latter limitation could only be possibly overcomed by using some metaprogramming capabilities of the rewriting engine used to implement this technique. The generators technique has been used in practical systems, for example as the basis for an implementation of the functional-programming language Curry [BHPR11]. There the information provided by a Damas-Milner like type system is used to improve the efficiency, because instead of just one universal generator, like in our proposal, several generators are used, one for each type, which results in a great shrink of the search space for the evaluation of generators. One could argue that our generators are fundamentally equivalent to defining a generator *genE* that could be reduced to any expression, and then replacing each free or extra variable with *genE*, which would be trivially complete. Nevertheless, in our approach the search space for generators is significantly smaller, especially when combined with type information.

The system has been implemented in a Maude prototype that allows us to study their expressivity and possible applications. This prototype uses the on-demand strategy natural rewriting [Esc04], thus providing an efficient implementation.

Regarding future work, we plan to improve our implementation by using the reflection capabilities of Maude to collect the evaluation of generators, in order to be able to present a computed answer for generators derivations, instead of relying on the trace to extract this information.

References

[AH06] S. Antoy, M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In Etalle and Truszczynski (eds.), *ICLP*. Lecture Notes in Computer Science 4079, pp. 87–101. Springer, 2006.

[BHPR11] B. Braßel, M. Hanus, B. Peemöller, F. Reck. KiCS2: A New Compiler from Curry to Haskell. In Kuchen (ed.), *Proceedings of the 20th international conference on Functional and constraint Logic Programming, WFLP 2011*. Lecture Notes in Computer Science 6816, pp. 1–18. Springer, 2011.

[BJM00] A. Bouhoula, J.-P. Jouannaud, J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science* 236:35–132, 2000.

[BN98] F. Baader, T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BS01] F. Baader, W. Snyder. Unification Theory. In Robinson and Voronkov (eds.), *Handbook of Automated Reasoning*. Pp. 445–532. Elsevier and MIT Press, 2001.

[CDE+07] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott. *All About Maude: A High-Performance Logical Framework*. Lecture Notes in Computer Science 4350. Springer, 2007.

[CMP07] M. Clavel, J. Meseguer, M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science* 373(1-2):70–91, 2007.

[DEE+11] F. Durán, S. Eker, S. Escobar, J. Meseguer, C. L. Talcott. Variants, Unification, Narrowing, and Symbolic Reachability in Maude 2.6. In Schmidt-Schauß (ed.), *RTA*. LIPIcs 10, pp. 31–40. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[DL07] J. Dios-Castro, F. López-Fraguas. Extra Variables can be Eliminated from Functional Logic Programs. *Electronic Notes in Theoretical Computer Science 188*, pp. 3–19, 2007.

[Esc04] S. Escobar. Implementing Natural Rewriting and Narrowing Efficiently. In Kameyama and Stuckey (eds.), *Proceedings of the 7th International Symposium on Functional and Logic Programming, FLOPS 2004*. Lecture Notes in Computer Science 2998, pp. 147–162. Springer, 2004.

[GHLR99] J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, M. Rodríguez-Artalejo. An Approach to Declarative Programming Based on a Rewriting Logic. *Journal of Logic Programming* 40(1):47–87, 1999.

[Han05] M. Hanus. Functional Logic Programming: From Theory to Curry. Technical report, Christian-Albrechts-Universität Kiel, 2005.

[Hul80] J. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*. Lecture Notes in Computer Science 87, pp. 318–334. Springer, 1980.

[Hus93] H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[LRS09a] F. López-Fraguas, J. Rodríguez-Hortalá, J. Sánchez-Hernández. A Fully Abstract Semantics for Constructor Systems. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications, RTA 2009*. Lecture Notes in Computer Science 5595, pp. 320–334. Springer, 2009.

[LRS09b] F. López-Fraguas, J. Rodríguez-Hortalá, J. Sánchez-Hernández. A Fully Abstract Semantics for Constructor Systems (Extended version). Technical report SIC-2-09, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2009.

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

[MH94] A. Middeldorp, E. Hamoen. Completeness Results for Basic Narrowing. *Appl. Algebra Eng. Commun. Comput.* 5:213–253, 1994.

[MR07] J. Meseguer, G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science* 373(3):213–237, 2007.

[MT07] J. Meseguer, P. Thati. Symbolic Reachability Analysis Using Narrowing and its Application to Verification of Cryptographic Protocols. *Higher-Order and Symbolic Computation* 20(1-2):123–160, 2007.

[Rod08] J. Rodríguez-Hortalá. A Hierarchy of Semantics for Non-deterministic Term Rewriting Systems. In Hariharan et al. (eds.), *Proceedings Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008*. Leibniz International Proceedings in Informatics 2, pp. 328–339. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008.

[RR12a] A. Riesco, J. Rodríguez-Hortalá. Generators: Detailed proofs. Technical report 07/12, Departamento de Sistemas Informáticos y Computación, 2012. `http://gpd.sip.ucm.es/snarrowing`.

[RR12b] A. Riesco, J. Rodríguez-Hortalá. S-Narrowing for Constructor Systems. In Roychoudhury and D'Souza (eds.), *Proceedings of the 9th International Colloquium on Theoretical Aspects of Computing, ICTAC 2012*. Lecture Notes in Computer Science 7521, pp. 136–150. Springer, 2012.

[SS86] L. Sterling, E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

Towards an Incremental and Modular Termination Analysis of Context-Sensitive Rewriting Systems (Work in Progress)*

Raúl Gutiérrez
DSIC, Universitat Politècnica de València, Spain
rgutierrez@dsic.upv.es

Salvador Lucas
DSIC, Universitat Politècnica de València, Spain
slucas@dsic.upv.es

Modularity is essential in software development, where a piece of software is often designed and implemented as a composition of simpler modules. So, if we want to prove that a program satisfies a given property, a modular approach becomes natural. With the development and successful use of the Dependency Pair Framework, which rather focuses on the *decomposition* of termination problems, less attention has been payed to modularity issues (which rather require the opposite approach). But the modular analysis of termination is still paramount for software developers. In this paper, we analyze modularity of context-sensitive rewrite systems. A modularity analysis was carried out by Gramlich and Lucas in 2002, but a correct notion of context-sensitive dependency pair (CS-DP) was not obtained until 2006. In this paper, we analyze modularity using CS-DPs.

1 Introduction

Term rewriting systems (TRSs) with many rules are frequently specified following a modular and incremental pattern and using well-known constructions such as *if-then-else* or *while* statements or generic modules (mathematical operands, functions that operate with lists, etc.) that are combined and reused many times to obtain the final program. When we try to prove computational properties on these systems with many rules, it is helpful to get use of the modular decomposition given by the developer to check properties by decomposition.

Termination is a fundamental property in programming languages, which allows us to know if for every computation the system will return in a finite time. The main problem dealing with termination from a modular perspective is that termination is not modular, even the union of two terminating TRSs that share no function symbol can be a non-terminating TRS, as shown by Toyama in 1987 [16].

Example 1 ([16]) *Toyama's example:*

$$\mathscr{R}_1 = \{\ \mathsf{f}(0,1,x) \to \mathsf{f}(x,x,x)\} \qquad \mathscr{R}_2 = \{\ \begin{array}{lcl}\mathsf{c}(x,y) & \to & x \\ \mathsf{c}(x,y) & \to & y\}\end{array}$$

The TRS $\mathscr{R} = \mathscr{R}_1 \cup \mathscr{R}_2$ resulting from the union of these two terminating TRSs can generate the following infinite rewrite sequence:

$$\mathsf{f}(\mathsf{c}(0,1),\mathsf{c}(0,1),\mathsf{c}(0,1)) \to_{\mathscr{R}_2} \mathsf{f}(0,\mathsf{c}(0,1),\mathsf{c}(0,1)) \to_{\mathscr{R}_2} \mathsf{f}(0,1,\mathsf{c}(0,1)) \to_{\mathscr{R}_1} \mathsf{f}(\mathsf{c}(0,1),\mathsf{c}(0,1),\mathsf{c}(0,1)) \to_{\mathscr{R}_2} \cdots$$

*Partially supported by the EU (FEDER), MINECO projects TIN2010-21062-C02-02 and TIN 2013-45732-C4-1-P, and project PROMETEO/2011/052. Salvador Lucas' research was developed during a sabbatical year at the CS Dept. of the UIUC and was also partially supported by NSF grant CNS 13-19109, Spanish MECD grant PRX12/00214, and GV grant BEST/2014/026. Raúl Gutiérrez is also partially supported by a Juan de la Cierva Fellowship from the Spanish MINECO, ref. JCI-2012-13528.

Submitted to:
PROLE 2014

This problem appears when combining *duplicating* and *collapsing* rules. A rule is duplicating if the number of occurrences of a variable in the right-hand side is greater than in the left-hand side, and a rule is collapsing if its right-hand side is a variable.

To obtain a modular analysis of termination, a more restrictive notion of termination is imposed: $\mathscr{C}_\varepsilon$-termination [10]. A system is $\mathscr{C}_\varepsilon$-terminating if it is still terminating after adding the rules in $\mathscr{R}_2$, called $\mathscr{C}_\varepsilon$-rules. These rules are used to simulate the behavior of the problematic collapsing rules. $\mathscr{C}_\varepsilon$-termination is a modular property for constructor sharing TRSs.

Context-sensitive rewriting (CSR [14, 15]) extends the signature of a rewrite system with a *replacement map*. A *replacement map* is a mapping $\mu : \mathscr{F} \to \wp(\mathbb{N})$ satisfying $\mu(f) \subseteq \{1,\ldots,\mathsf{ar}(f)\}$ for every function symbol f in the signature $\mathscr{F}$ [14], where $\mathsf{ar}(f)$ means the arity of f. We use it to discriminate the argument positions in which the rewriting steps are *allowed*; rewriting at the topmost position is always possible. In this way, a restriction of the rewrite relation is obtained. CSR has shown useful to model evaluation strategies in programming languages and also to achieve a terminating behaviour by pruning (all) infinite rewrite sequences. In particular, it is an essential ingredient to analyze the *termination behavior* of programming languages like CafeOBJ, Maude, OBJ, etc. [8, 12].

Example 2 *Consider the following context-sensitive term rewriting system (CS-TRS) from [9] that computes the list of all prime numers using the sieve of Eratosthenes in a lazy way:*

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{if}(\mathsf{true},x,y) & \to & x & \mathsf{sieve}(x{:}y) & \to & x{:}\mathsf{sieve}(\mathsf{filter}(x,y)) \\
\mathsf{if}(\mathsf{false},x,y) & \to & y & \mathsf{from}(x) & \to & x{:}\mathsf{from}(\mathsf{s}(x)) \\
\mathsf{filter}(x,y{:}z) & \to & \mathsf{if}(\mathsf{divides}(x,y),\mathsf{filter}(x,z),y{:}\mathsf{filter}(x,z)) & \mathsf{primes} & \to & \mathsf{sieve}(\mathsf{from}(\mathsf{s}(\mathsf{s}(0))))
\end{array}$$

together with $\mu(\mathsf{if}) = \mu(:) = \{1\}$ and $\mu(f) = \{1,\ldots,\mathsf{ar}(f)\}$ for all other symbols f. Function from *is used to generate an infinite list of natural numbers and function* sieve *filters those that are primes. The replacement restriction on the second argument of $(:)$ avoids an infinite computation. This system was proved μ-terminating in [2] showing that the dependency graph has no cycles. The reason is that* divides *rule was not defined and the term* $\mathsf{divides}(\mathsf{s}(\mathsf{s}(x)),y)$ *could not be rewritten to* true *or* false*. But, if we define* divides *in a standard way:*

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{zero}(0) & \to & \mathsf{true} & x-0 & \to & x \\
\mathsf{zero}(\mathsf{s}(x)) & \to & \mathsf{false} & \mathsf{s}(x)-\mathsf{s}(y) & \to & x-y \\
\mathsf{mod}(0,\mathsf{s}(x)) & \to & 0 & \mathsf{s}(x)\leq 0 & \to & \mathsf{false} \\
\mathsf{mod}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{if}(y\leq x,\mathsf{mod}(x-y,\mathsf{s}(y)),\mathsf{s}(x)) & 0\leq x & \to & \mathsf{true} \\
\mathsf{divides}(x,y) & \to & \mathsf{zero}(\mathsf{mod}(y,x)) & \mathsf{s}(x)\leq\mathsf{s}(y) & \to & x\leq y
\end{array}$$

There is no tool for proving termination that can prove this system μ-terminating although we know that these new rules are μ-terminating.

In [11], a modularity analysis of termination of CSR was carried out, but since there was no correct definition of CS-DP until [1], a modular analysis of termination based on CS-DPs was not possible. In this paper, we exploit the modular behaviour of CS-DPs to obtain modularity results from a different perspective of the obtained by Gramlich and Lucas.

2 Preliminaries

See [7] and [14] for basics on term rewriting and CSR, respectively. Throughout the paper, $\mathscr{X}$ denotes a countable set of variables and $\mathscr{F}$ denotes a signature, i.e., a set of function symbols each having a fixed

arity given by a mapping $\mathsf{ar} : \mathscr{F} \to \mathbb{N}$. The set of terms built from $\mathscr{F}$ and $\mathscr{X}$ is $\mathscr{T}(\mathscr{F},\mathscr{X})$. Terms are viewed as labelled trees in the usual way. The symbol labeling the root of the term s is denoted as $root(s)$. Positions $p,q,\ldots$ are represented by chains of positive natural numbers used to address subterms of s. Given positions p,q, we denote its concatenation as $p.q$. We denote the empty chain by $\wedge$. The set of positions of a term s is $\mathscr{P}os(s)$. For a replacement map μ, the set of *active positions* $\mathscr{P}os^{\mu}(s)$ of $s \in \mathscr{T}(\mathscr{F},\mathscr{X})$ is: $\mathscr{P}os^{\mu}(s) = \{\wedge\}$, if $s \in \mathscr{X}$ and $\mathscr{P}os^{\mu}(s) = \{\wedge\} \cup \bigcup_{i \in \mu(root(s))} i.\mathscr{P}os^{\mu}(s|_i)$, if $t \notin \mathscr{X}$. Let $\mathscr{V}ar(s) = \{x \in \mathscr{X} \mid \exists p \in \mathscr{P}os(s), s|_p = x\}$, $\mathscr{V}ar^{\mu}(s) = \{x \in \mathscr{V}ar(s) \mid \exists p \in \mathscr{P}os^{\mu}(s), s|_p = x\}$ and $\mathscr{V}ar^{\not\mu}(s) = \{x \in \mathscr{V}ar(s) \mid s \rhd_{\not\mu} x\}$. We say that $s \trianglerighteq_{\mu} t$ if there is $p \in \mathscr{P}os^{\mu}(s)$ such that $t = s|_p$. We write $s \rhd_{\mu} t$ if $s \trianglerighteq_{\mu} t$ and $s \neq t$. Moreover, $s \rhd_{\not\mu} t$ if there is a *frozen position* p, i.e. $p \in \mathscr{P}os(s) - Pos^{\mu}(s)$, such that $t = s|_p$. A rewrite rule is an ordered pair (ℓ, r), written $\ell \to r$, with $\ell, r \in \mathscr{T}(\mathscr{F},\mathscr{X})$, $\ell \notin \mathscr{X}$ and $\mathscr{V}ar(r) \subseteq \mathscr{V}ar(\ell)$. A TRS is a pair $\mathscr{R} = (\mathscr{F}, R)$ where R is a set of rewrite rules. Given $\mathscr{R} = (\mathscr{F}, R)$, we consider $\mathscr{F}$ as the disjoint union $\mathscr{F} = \mathscr{C} \uplus \mathscr{D}$ of symbols $\mathsf{c} \in \mathscr{C}$, called *constructors* and symbols $\mathsf{f} \in \mathscr{D}$, called *defined functions*, where $\mathscr{D} = \{\mathsf{root}(\ell) \mid \ell \to r \in R\}$ and $\mathscr{C} = \mathscr{F} - \mathscr{D}$. We say that $t \in \mathscr{T}(\mathscr{F},\mathscr{X}) - \mathscr{X}$ is a *hidden term* of s, $t \in \mathsf{HT}(s,\mu)$, if $s \rhd_{\not\mu} t$ and $\mathsf{root}(t)$ is defined. Given a CS-TRS $(\mathscr{R},\mu)$, we have $s \hookrightarrow_{\mathscr{R},\mu} t$ (alternative $s \overset{p}{\hookrightarrow}_{\mathscr{R},\mu} t$ if we want to make the position explicit) if there are $\ell \to r \in \mathscr{R}$, $p \in \mathscr{P}os^{\mu}(s)$ and a substitution σ with $s|_p = \sigma(\ell)$ and $t = t[\sigma(r)]_p$. We write $s \overset{>q}{\hookrightarrow}_{\mathscr{R},\mu} t$ if $s \overset{p}{\hookrightarrow}_{\mathscr{R},\mu} t$ and $q > p$. A rule $\ell \to r$ is *conservative* if $\mathscr{V}ar^{\mu}(r) \subseteq \mathscr{V}ar^{\mu}(\ell)$. A rule $\ell \to r$ is *strongly conservative* if it is conservative and $\mathscr{V}ar^{\mu}(\ell) \cap \mathscr{V}ar^{\not\mu}(\ell) = \mathscr{V}ar^{\mu}(r) \cap \mathscr{V}ar^{\not\mu}(r) = \varnothing$. $\mathscr{R}$ is conservative (resp. strongly conservative) if all rules in $\mathscr{R}$ are. A CS-TRS $(\mathscr{R},\mu)$ is *terminating* if $\hookrightarrow_{\mathscr{R},\mu}$ is well-founded. A CS-TRS $(\mathscr{R},\mu)$ is $\mathscr{C}_{\varepsilon}$-terminating [11] iff $(\mathscr{R} \uplus \mathscr{C}_{\varepsilon}, \mu \uplus \mu_{\mathscr{C}_{\varepsilon}})$ is terminating, where $\mathscr{C}_{\varepsilon} = (\{\mathsf{c}\}, \{\mathsf{c}(x,y) \to x, \mathsf{c}(x,y) \to y\})$ (with c being a fresh symbol) and $\mu_{\mathscr{C}_{\varepsilon}}(\mathsf{c}) = \{1,2\}$.

2.1 Context Sensitive Dependency Pairs

Dependency pairs [6] describe the *propagation* of *function calls* in rewrite sequences. In CSR, we have two kind of (potential) function calls: *direct* calls, i.e., calls at *active* (*replacing*) positions and *delayed* calls, i.e., calls at *frozen* (*non-replacing*) positions that can be activated in forthcoming reduction steps. These function calls are captured in two different ways. For rules $\ell \to r$ such that r contains some *defined* symbol g at an active position, the function call to g is represented as a new rule $u \to v$ (called *dependency pair*) where $u = \mathsf{f}^{\sharp}(\ell_1,\ldots,\ell_k)$ if $\ell = \mathsf{f}(\ell_1,\ldots,\ell_k)$ and $v = \mathsf{g}^{\sharp}(s_1,\ldots,s_m)$ if $\mathsf{g}(s_1,\ldots,s_m)$ is an active subterm of r and g is defined. The notation $\mathsf{f}^{\sharp}$ for a given symbol f means that f is *marked*. In practice, we often capitalize f and use F instead of $\mathsf{f}^{\sharp}$ in our examples. Function calls to g which are at frozen positions of r cannot be issued 'immediately', but could be activated 'in the future'. This situation is carried out by the *migrating variables* and modeled by *collapsing* DPs. Given a rule $\ell \to r$, x is a *migrating variable* if x is at an active position in r but not in ℓ [1]. For rules $\ell \to r$, *collapsing* DPs are pairs of the form $u \to x$ where $u = \mathsf{f}^{\sharp}(\ell_1,\ldots,\ell_k)$ if $\ell = \mathsf{f}(\ell_1,\ldots,\ell_k)$ and x is a *migrating variable*. The idea is that calls which can eventually be activated are subterms of $\sigma(x)$ for σ being the matching substitution of the rewriting step involving the rule $\ell \to r$. Formally, $\mathsf{DP}(\mathscr{R},\mu) = \mathsf{DP}_{\mathscr{F}}(\mathscr{R},\mu) \cup \mathsf{DP}_{\mathscr{X}}(\mathscr{R},\mu)$ where $\mathsf{DP}_{\mathscr{F}}(\mathscr{R},\mu) = \{\ell^{\sharp} \to s^{\sharp} \mid \ell \to r \in R, r \trianglerighteq_{\mu} s, root(s) \in \mathscr{D}\}$, $\mathsf{DP}_{\mathscr{X}}(\mathscr{R},\mu) = \{\ell^{\sharp} \to x \mid \ell \to r \in R, x \in \mathscr{V}ar^{\mu}(r) - \mathscr{V}ar^{\mu}(l)\}$ and $\mu^{\sharp}(f) = \mu(f)$ if $f \in \mathscr{F}$, and $\mu^{\sharp}(f^{\sharp}) = \mu(f)$ if $f \in \mathscr{D}$.

Example 3 *In Example 2, we obtain the following set of CS-DPs:*

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{PRIMES} & \to & \mathsf{SIEVE}(\mathsf{from}(\mathsf{s}(\mathsf{s}(0)))) & \mathsf{IF}(\mathsf{true},x,y) & \to & x \\
\mathsf{PRIMES} & \to & \mathsf{FROM}(\mathsf{s}(\mathsf{s}(0))) & \mathsf{IF}(\mathsf{false},x,y) & \to & y \\
\mathsf{FILTER}(x,y{:}z) & \to & \mathsf{IF}(\mathsf{divides}(x,y),\mathsf{filter}(x,z),y{:}\mathsf{filter}(x,z)) & \mathsf{DIVIDES}(x,y) & \to & \mathsf{ZERO}(\mathsf{mod}(y,x)) \\
\mathsf{FILTER}(x,y{:}z) & \to & \mathsf{DIVIDES}(x,y) & \mathsf{DIVIDES}(x,y) & \to & \mathsf{MOD}(y,x) \\
\mathsf{MOD}(\mathsf{s}(x),\mathsf{s}(y)) & \to & y \leq^\sharp x & \mathsf{s}(x) -^\sharp \mathsf{s}(y) & \to & x -^\sharp y \\
\mathsf{MOD}(\mathsf{s}(x),\mathsf{s}(y)) & \to & \mathsf{IF}(y \leq x, \mathsf{mod}(x-y,\mathsf{s}(y)),\mathsf{s}(x)) & \mathsf{s}(x) \leq^\sharp \mathsf{s}(y) & \to & x \leq^\sharp y
\end{array}$$

together with $\mu(\mathsf{IF}) = \mu(:) = \{1\}$ *and* $\mu(f) = \{1,\ldots,\mathsf{ar}(f)\}$ *for all other symbols* f.

To prove termination, we have to show that there is no infinite *chain* of CS-DPs. A sequence $u_1 \to v_1$, $u_2 \to v_2$, ... of CS-DPs is a chain if there is a substitution such that for all $i \geq 1$, (1) if $u_i \to v_i$ is not collapsing, then $\sigma(v_i) \hookrightarrow^*_{\mathscr{R},\mu} \sigma(u_{i+1})$ or (2) if $u_i \to v_i$ is collapsing, then there is a term w_i such that $(2a)$ $\sigma(v_i) \trianglerighteq_\mu w_i$ and $(2b)$ $w_i^\sharp \hookrightarrow^*_{\mathscr{R},\mu} \sigma(u_{i+1})$. From now on, we assume that all CS-TRSs are finite.

3 Rewriting Modules

In programming, the idea of module comes in a natural way. A programmer groups in a module definitions of functions having something in common (not necessarily *among* them; often as a set of services provided to *external* users –i.e., other modules–). Then new modules which use these functions are written. In term rewriting, modules arise in a natural way, when rules defining a given function symbol f (by means of rules $f(\ell_1,\ldots,\ell_k) \to r$) are collected together, and they are used by other rules from other modules. This modular and hierarchical approach is exploited in [17] to prove termination in a modular and incremental way. Although termination is not modular (in general), the author succeeded thanks to imposing a harder termination condition for modules: the $\mathscr{C}_\varepsilon$-termination. The notion of module is introduced by Urbain using the following notation.

Definition 1 [17] *Let* $\mathscr{R}_1 = (\mathscr{F}_1, R_1)$ *be a TRS. A module extending* $\mathscr{R}_1$ *is a pair* $[\mathscr{F}_2 \mid \mathscr{R}_2]$ *such that:*

1. $\mathscr{F}_1 \cap \mathscr{F}_2 = \varnothing$;
2. $\mathscr{R}_2$ *is a TRS over* $\mathscr{F}_1 \cup \mathscr{F}_2$;
3. *For all* $\ell \to r \in R_2$, $\mathsf{root}(\ell) \in \mathscr{F}_2$.

Then, $\mathscr{R}_1 \cup \mathscr{R}_2$ *over* $\mathscr{F}_1 \cup \mathscr{F}_2$ *is a* hierarchical extension *of* $\mathscr{R}_1$ *with module* $[\mathscr{F}_2 \mid \mathscr{R}_2]$, *written:*

$$\mathscr{R}_1 \longleftarrow [\mathscr{F}_2 \mid \mathscr{R}_2]$$

Note that $\mathscr{D}_2 \subseteq \mathscr{F}_2$. Roughly speaking, the notation $[\mathscr{F} \mid \mathscr{R}]$ behaves as an interface of $\mathscr{R}$ where $\mathscr{F}$ represents the symbols that can be imported by other modules. Context-sensitive rewriting extends the signature of TRSs with a replacement map. Then, if we want to extend the previous modular approach to CSR, we impose an agreement among the replacement maps of the shared symbols between modules. For that reason we require the replacement maps for the modules to be compatible in the following sense.

Definition 2 (Compatibility [11]) *A replacement map* μ_1 *on* $\mathscr{F}_1$ *is compatible with a replacement map* μ_2 *on* $\mathscr{F}_2$ *if they have the same replacement restrictions for shared function symbols, i.e., if* $\mu_1(f) = \mu_2(f)$ *for every* $f \in \mathscr{F}_1 \cap \mathscr{F}_2$.

Now, we are going to extend Definition 1 for taking into account the replacement restrictions.

Definition 3 *Let $\mathscr{R}_1 = (\mathscr{F}_1, R_1)$ be a TRS and μ_1 a replacement map on $\mathscr{F}_1$. A module extending $(\mathscr{R}_1, \mu_1)$ is a pair $[\mathscr{F}_2 \mid (\mathscr{R}_2, \mu_2)]$ such that:*

1. *$\mathscr{F}_1 \cap \mathscr{F}_2 = \varnothing$;*
2. *$\mathscr{R}_2$ is a TRS over $\mathscr{F}_1 \cup \mathscr{F}_2$ and μ_2 is a replacement map on $\mathscr{F}_1 \cup \mathscr{F}_2$;*
3. *μ_1 are μ_2 are compatible;*
4. *for all $\ell \to r \in R_2$, $\mathsf{root}(\ell) \in \mathscr{F}_2$.*

System $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ over $\mathscr{F}_1 \cup \mathscr{F}_2$ is a hierarchical extension of $(\mathscr{R}_1, \mu_1)$ with module $[\mathscr{F}_2 \mid (\mathscr{R}_2, \mu_2)]$ and we write it like:

$$(\mathscr{R}_1, \mu_1) \longleftarrow [\mathscr{F}_2 \mid (\mathscr{R}_2, \mu_2)]$$

Note that symbols from $\mathscr{F}_1$ can appear in rules from $\mathscr{R}_2$, but not as root symbols on the left-hand side of the rules. With this notation, we can also describe the union of composable systems. For the sake of readability, we denote $[\mathscr{F}_0 \mid (\mathscr{R}_0, \mu_0)] \longleftarrow [\mathscr{F}_1 \mid (\mathscr{R}_1, \mu_1)]$ the hierarchical extension with $[\mathscr{F}_1 \mid (\mathscr{R}_1, \mu_1)]$ of the whole hierarchy extended with $[\mathscr{F}_0 \mid (\mathscr{R}_0, \mu_0)]$.

Definition 4 *We say that a module $[\mathscr{F}_2 \mid (\mathscr{R}_2, \mu_2)]$ extends a hierarchy headed by $[\mathscr{F}_0 \mid (\mathscr{R}_0, \mu_0)]$* independently *of a module $[\mathscr{F}_1 \mid (\mathscr{R}_1, \mu_1)]$ if:*

1. *$\mathscr{F}_1 \cap \mathscr{F}_2 = \varnothing$;*
2. *$[\mathscr{F}_0 \mid (\mathscr{R}_0, \mu_0)] \longleftarrow [\mathscr{F}_1 \mid (\mathscr{R}_1, \mu_1)]$ and*
3. *$[\mathscr{F}_0 \mid (\mathscr{R}_0, \mu_0)] \longleftarrow [\mathscr{F}_2 \mid (\mathscr{R}_2, \mu_2)]$.*

3.1 Modular Decomposition

As for TRSs, we show how to decompose a CS-TRS into a 'canonical' modular hierarchy, a hierarchy of minimal modules which cannot be split up further. In order to do that, we follow the graph of purely syntactical dependency relation between symbols given in [17].

Definition 5 (Dependency) *Given a TRS $\mathscr{R} = (\mathscr{F}, R)$, we say that $f \in \mathscr{F}$ directly depends on $g \in \mathscr{F}$, written $f \rhd g$, if and there is a rule $\ell \to r \in R$ with*

- *$f = \mathsf{root}(\ell)$ and*
- *g occurs in ℓ or r.*

Besides the original dependency relation in [17], our dependency relation also considers the symbols in the left-hand side of the rule. Using this relation, the decomposition is done in two steps:

Definition 6 (Modular Decomposition of a TRS) *For a TRS $\mathscr{R} = (\mathscr{F}, R)$:*

1. *we build a graph $\mathscr{G}$ the nodes of which are symbols of $\mathscr{F}$ and such that there is an arc from a node x to a node y if and only if $x \rhd y$*
2. *we pack together symbols of strongly connected components of $\mathscr{G}$, i.e., symbols f and g such that:*

$$f \rhd^* g \text{ and } g \rhd^* f$$

In the obtained hierarchy there is no cycle because symbols of mutually recursive functions appear in the same module. Thus, they belong to the same modules. In the dependency pair framework, a similar graph is constructed for decomposing a DP problem into smaller DP problems, but the dependency relation is more involved because the idea is to capture possible infinite chains between pairs. Dealing with CSR, the replacement restrictions do not change the natural decomposition of the modules.

Example 4 *Figure 1 shows the modular decomposition of Example 2.*

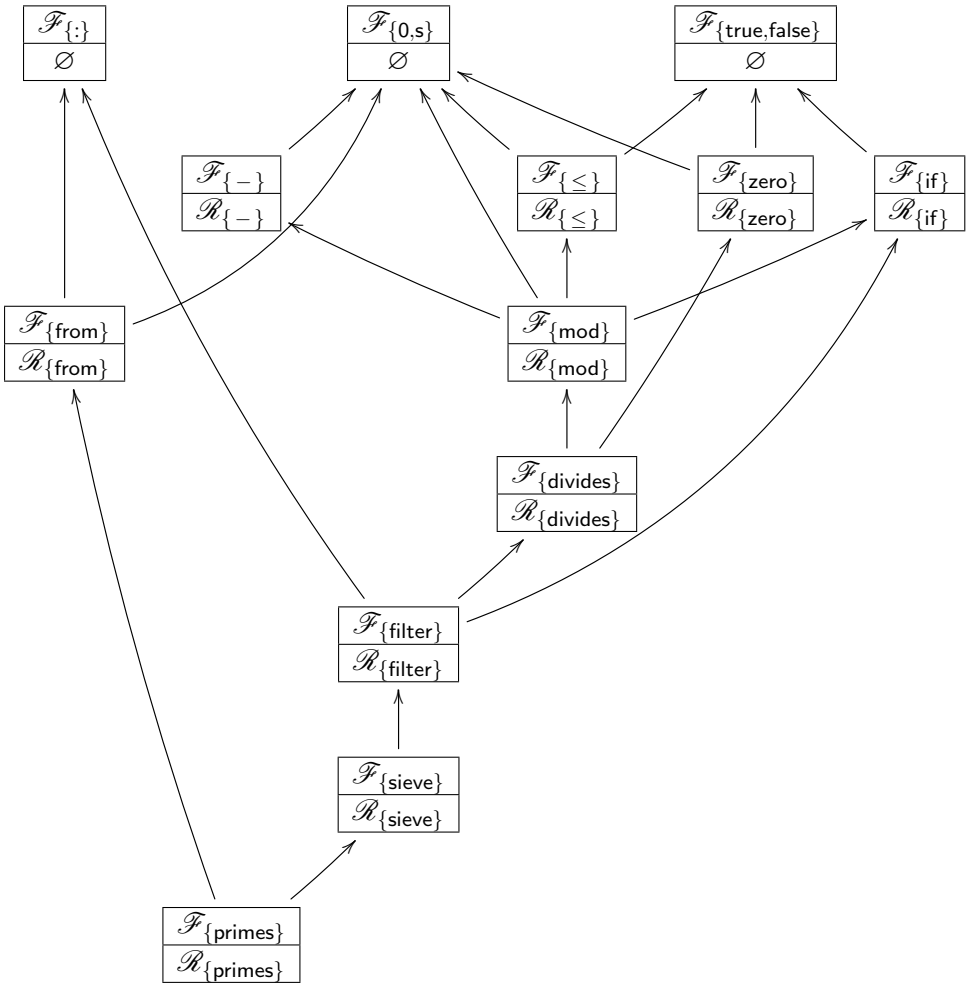


Figure 1: Modular decomposition of Example 2

4 Incremental and Modular Termination

Modular decomposition is quite natural, but from a CS-DP and CS-DP chain point of view, dependencies between modules differ. In this section, we define the notions of CS-DP of a module and relative CS-DP chain.

4.1 CS-DPs of Modules

In CSR, we have to consider two kinds of dependency pairs, the DPs that represent direct calls and the DPs that represent activations of function calls. When dealing with modules, the notion of collapsing DP is still important.

Example 5 [18] (Example 5 modified) *Let us consider the following example:*

$$\mathscr{R}_1 = \begin{array}{lll} \{\mathsf{if}(\mathsf{true},x,y) & \to & x \\ \mathsf{if}(\mathsf{false},x,y) & \to & y\} \end{array} \qquad \mathscr{R}_2 = \{\mathsf{f}(x) \to \mathsf{if}(x,\mathsf{c},\mathsf{f}(\mathsf{false}))\}$$

where $\mu_2(\mathsf{f}) = \{1\}$, $\mu_1(\mathsf{if}) = \mu_2(\mathsf{if}) = \{1,2\}$, *and* $\mu_2(\mathsf{c}) = \mu_1(\mathsf{true}) = \mu_1(\mathsf{false}) = \mu_2(\mathsf{false}) = \varnothing$. *We can see* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *over* $\mathscr{F}_1 \cup \mathscr{F}_2$ *as a hierarchichal extension of* $(\mathscr{R}_1, \mu_1)$ *with module* $[\mathscr{F}_2 \mid (\mathscr{R}_2, \mu_2)]$ *(i.e.,* $(\mathscr{R}_1, \mu_1) \leftarrow [\mathscr{F}_2 \mid (\mathscr{R}_2, \mu_2)]$*). The set of CS-DPs of* $(\mathscr{R}_1, \mu_1)$ *is* $\mathsf{DP}_1 = \{\mathsf{IF}(\mathsf{false}, x, y) \to y\}$ *and the set of CS-DPs of* $(\mathscr{R}_2, \mu_2)$ *is* $\mathsf{DP}_2 = \varnothing$ *(*if *is not a defined function in* $\mathscr{R}_2$*). Both CS-TRSs are* $\mathscr{C}_\varepsilon$*-terminating independently, but if we consider the union of these two CS-TRSs* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$*, the set of CS-DPs of* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *is* $DP_3 = \{\mathsf{IF}(\mathsf{false}, x, y) \to y, \mathsf{F}(x) \to \mathsf{IF}(x, \mathsf{c}, \mathsf{f}(\mathsf{false}))\}$ *(a new CS-DP is considered) and an infinite CS-DP chain exists:*

$$\mathsf{F}(\mathsf{false}) \to_{\mathsf{DP}_3} \mathsf{IF}(\mathsf{false}, \mathsf{c}, \mathsf{f}(\mathsf{false})) \to_{\mathsf{DP}_3} \mathsf{F}(\mathsf{false}) \to_{\mathsf{DP}_3} \cdots$$

where the new CS-DP appeared by the union is relevant to capture the infinite computation.

The original notion of DP of module only considers DPs appeared in the module, but as we have seen in the example this is not the case when dealing with CS-DPs. To obtiain a similar notion of DP of module, we have to work with *conservative* CS-TRSs, i.e., CS-TRSs without function call activations (collapsing CS-DPs).

Definition 7 *Let* $\mathscr{M} = [\mathscr{F}_\mathscr{M} \mid (\mathscr{R}, \mu)]$ *be a module where* $\mathscr{R} = (\mathscr{F}, R) = (\mathscr{C} \uplus \mathscr{D}, R)$ *is a TRS,* $\mathscr{F}_\mathscr{M} \subseteq \mathscr{F}$*,* μ *a replacement map on* $\mathscr{F}$ *and* $(\mathscr{R}, \mu)$ *is conservative. We define* $\mathsf{MDP}(\mathscr{M}) = \mathsf{MDP}_\mathscr{F}(\mathscr{M})$ *to be the set of* conservative context-sensitive dependency pairs[1] *of module* $\mathscr{M}$ *where:*

$$\mathsf{MDP}_\mathscr{F}(\mathscr{M}) \quad = \quad \{\ell^\sharp \to s^\sharp \mid \quad \ell \to r \in R, r \trianglerighteq_\mu s, \mathsf{root}(s) \in \mathscr{D} \cap \mathscr{F}_\mathscr{M}\}$$

We extend μ *into* $\mu^\sharp$ *by* $\mu^\sharp(f) = \mu(f)$ *if* $f \in \mathscr{F}$ *and* $\mu^\sharp(f^\sharp) = \mu(f)$ *if* $f \in \mathscr{D} \cap \mathscr{F}_\mathscr{M}$*.*

4.2 Relative CS-DP Chains

From the definition of CS-DPs of a module, we define CS-DP chains *relative* to some CS-TRS.

Definition 8 *Let* $\mathscr{M} = [\mathscr{F}_1 \mid (\mathscr{R}_1, \mu_1)]$ *be a module where* $(\mathscr{R}_1, \mu_1)$ *is conservative and* $(\mathscr{R}_2, \mu_2)$ *an arbitrary CS-TRS where* μ_1 *and* μ_2 *are compatible. A* CS-DP chain *of* $\mathsf{MDP}(\mathscr{M})$ *relative to* $(\mathscr{R}_2, \mu_2)$ *is a sequence of pairs* $u_i \to v_i \in \mathsf{MDP}(\mathscr{M})$ *together with a substitution* σ *such that for all* $i \geq 1$*, We assume that different occurrences of pairs do not share any variable. A CS-DP chain is* minimal *iff* $\sigma(v_i)$ *is* $(\mathscr{R}_2, \mu_2)$*-terminating.*

Most recent notion of chain, the $(\mathscr{P}, \mathscr{R}, \mathscr{S}, \mu)$-chain, contains three TRSs: $\mathscr{P}$ models the behaviour of CS-DPs; $\mathscr{R}$ models the behaviour of the rules; and $\mathscr{S}$ models the subterm and marking in CS-DP chains. But the given definition is enough for the purpose of the paper because we are dealing with conservative CS-TRSs.

Proposition 1 *A conservative CS-TRS* $(\mathscr{R}, \mu)$ *where* $\mathscr{R} = (\mathscr{F}, R)$ *is* $\mathscr{C}_\varepsilon$*-terminating if and only if there is no infinite minimal chain of* $\mathsf{MDP}([\mathscr{F} \mid (\mathscr{R}, \mu)])$ *relative to* $(\mathscr{R} \cup \mathscr{C}_\varepsilon, \mu \cup \mu_{\mathscr{C}_\varepsilon})$*.*

Proof 1 *Since* c *does not belong to* $\mathscr{F}$*, and since* $\mu_{\mathscr{C}_\varepsilon}(\mathsf{c}) = \{1,2\}$ *then* $\mathsf{DP}(\mathscr{R} \cup \mathscr{C}_\varepsilon, \mu \cup \mu_{\mathscr{C}_\varepsilon}) = \mathsf{DP}(\mathscr{R}, \mu)$*, where* DP *is a function that obtains the CS-DPs of a CS-TRS. So,* $\mathsf{MDP}([\mathscr{F} \mid (\mathscr{R}, \mu)]) = \mathsf{DP}(\mathscr{R} \cup \mathscr{C}_\varepsilon, \mu \cup \mu_{\mathscr{C}_\varepsilon})$*. Therefore, proving the termination of* $\mathsf{MDP}([\mathscr{F} \mid (\mathscr{R}, \mu)])$ *relative to* $(\mathscr{R} \cup \mathscr{C}_\varepsilon, \mu \cup \mu_{\mathscr{C}_\varepsilon})$ *is the same as proving termination of* $\mathsf{DP}(\mathscr{R} \cup \mathscr{C}_\varepsilon, \mu \cup \mu_{\mathscr{C}_\varepsilon})$ *relative to* $(\mathscr{R} \cup \mathscr{C}_\varepsilon, \mu \cup \mu_{\mathscr{C}_\varepsilon})$*, that is proving the* $\mathscr{C}_\varepsilon$*-termination of* $(\mathscr{R}, \mu)$*.* ■

[1] In [3], the notion of CS-DP includes some extra conditions to discard CS-DPs that, by construction, are not involved in infinite chains (narrowability and subterm conditions). To ease readability, we do not include these extra conditions, but the results obtained in the paper are still applicable adding these conditions.

4.3 Termination with modules

In contrast to the unrestricted approach (pure term rewriting), in CSR a chain is possible in a hierarchical extension even being impossible for the specific component (Example 5). The following modularity result can be extracted when the pairs are not collapsing.

Lemma 1 *Let $(\mathscr{R}_1,\mu_1)$ be a CS-TRS where $\mathscr{R}_1=(\mathscr{F}_1,R_1)$ and $[\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)]$ be a module such that $[\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)] \longleftarrow [\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)]$. Then, for any two pairs $u_1 \to v_1 \in \mathsf{MDP}_{\mathscr{F}_1}([\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)])$ and $u_2 \to v_2 \in \mathsf{MDP}([\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)])$, there is no substitution σ such that:*

$$\sigma(v_1) \xhookrightarrow[\mathscr{R}_1\cup\mathscr{R}_2]{>\Lambda}{}^{*} \sigma(u_2)$$

Proof 2 *Since $\mathsf{root}(\sigma(u_2))=\mathsf{root}(u_2)\in\mathscr{D}_2^\sharp\subseteq\mathscr{F}_2^\sharp$, $\mathsf{root}(\sigma(v_1))=\mathsf{root}(v_1)\in\mathscr{D}_1^\sharp\subseteq\mathscr{F}_1^\sharp$ and $\mathscr{D}_1^\sharp\cap\mathscr{D}_2^\sharp=\varnothing$, we obtain that $\mathsf{root}(v_1)\neq\mathsf{root}(u_2)$. Hence, $\sigma(v_1)$ cannot be rewritten* below the root *to $\sigma(u_2)$.* ■

In [17] the termination of modules is based on the following two theorems:

Theorem 1 [17, Theorem 1] *Let $[\mathscr{F}_1 \mid \mathscr{R}_1] \longleftarrow [\mathscr{F}_2 \mid \mathscr{R}_2]$ be a hierarchical extension of $\mathscr{R}_1=(\mathscr{F}_1,R_1)$; if*

- *$\mathscr{R}_1$ is $\mathscr{C}_\varepsilon$-terminating, and*
- *there is no infinite dependency chain of $[\mathscr{F}_2 \mid \mathscr{R}_2]$ relative to $\mathscr{R}_1\cup\mathscr{R}_2$,*

then $\mathscr{R}_1\cup\mathscr{R}_2$ is terminating.

Theorem 2 [17, Theorem 2] *Let $[\mathscr{F}_1 \mid \mathscr{R}_1] \longleftarrow [\mathscr{F}_2 \mid \mathscr{R}_2]$ be a hierarchical extension of $\mathscr{R}_1=(\mathscr{F}_1,R_1)$, and $[\mathscr{F}_3 \mid \mathscr{R}_3]$ be a module extending $\mathscr{R}_1$ independently of $\mathscr{R}_2$. If*

- *$\mathscr{R}_1\cup\mathscr{R}_2$ is $\mathscr{C}_\varepsilon$-terminating, and*
- *there is no infinite dependency chain of $[\mathscr{F}_3 \mid \mathscr{R}_3]$ relative to $\mathscr{R}_1\cup\mathscr{R}_3\cup\mathscr{C}_\varepsilon$,*

then $\mathscr{R}_1\cup\mathscr{R}_2\cup\mathscr{R}_3$ is $\mathscr{C}_\varepsilon$-terminating.

But as we have seen before in the previous examples, the adaptation of these theorems to CSR needs to consider more conditions to safely extend hierarchical extensions to CSR. The key idea behind the results on hierarchical extensions is the possibility of building an infinite CS-DP chain of $\mathsf{MDP}([\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)])$ relative to $(\mathscr{R}_1\cup\mathscr{C}_\varepsilon,\mu_1\cup\mu_{\mathscr{C}_\varepsilon})$ from an infinite minimal CS-DP chain of $\mathsf{MDP}([\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)])$ relative to $(\mathscr{R}_1\cup\mathscr{R}_2,\mu_1\cup\mu_2)$. In [13], a couple of interpretations are presented to simulate rewriting steps on $(\mathscr{R}_2,\mu_2)$ by rewriting steps on $(\mathscr{C}_\varepsilon,\mu_{\mathscr{C}_\varepsilon})$ when $(\mathscr{R}_2,\mu_2)$ is terminating, but none of them are suitable for just conservative CS-TRSs. In order to obtain a result similar to the previous theorems we have to impose a stronger statement (strongly conservative) to get use of the *basic μ-interpretation* in [13]. Basic μ-interpretation simulates rewriting steps on a terminating CS-TRS $(\mathscr{R}_2,\mu_2)$ as rewriting steps using $\mathscr{C}_\varepsilon$-rules.

Definition 9 [13] (Basic μ-interpretation) *Let $(\mathscr{R},\mu)$ be a CS-TRS over $\mathscr{F}$ and $\Delta\subseteq\mathscr{F}$. Let $>$ be an arbitrary total ordering over $\mathscr{T}(\mathscr{F}^\sharp\cup\{\bot,\mathsf{c}\},\mathscr{X})$ where $\bot$ is a new constant symbol and c is a new binary symbol. The interpretation $\Phi_{\Delta,\mu}$ is a mapping from μ-terminating terms in $\mathscr{T}(\mathscr{F}^\sharp,\mathscr{X})$ to terms in $\mathscr{T}(\mathscr{F}^\sharp\cup\{\bot,\mathsf{c}\},\mathscr{X})$ defined as follows:*

$$\Phi_{\Delta,\mu}(t)=\begin{cases} t & \text{if} \quad t\in\mathscr{X} \\ f(\Phi_{\Delta,\mu,f,1}(t_1),\ldots,\Phi_{\Delta,\mu,f,n}(t_n)) & \text{if} \quad t=f(t_1\ldots t_n) \text{ and } f\in\Delta \\ \mathsf{c}(f(\Phi_{\Delta,\mu,f,1}(t_1),\ldots,\Phi_{\Delta,\mu,f,n}(t_n)),t') & \text{if} \quad t=f(t_1\ldots t_n) \text{ and } f\notin\Delta \end{cases}$$

$$\textit{where} \quad \Phi_{\Delta,\mu,f,i}(t) = \begin{cases} \Phi_{\Delta,\mu}(t) & \text{if } i \in \mu(f) \\ t & \text{if } i \notin \mu(f) \end{cases}$$

$$t' = order\left(\{\Phi_{\Delta,\mu}(u) \mid t \hookrightarrow_{\mathscr{R},\mu} u\}\right)$$

$$order(T) = \begin{cases} \bot, & \text{if } T = \varnothing \\ \mathsf{c}(t, order(T - \{t\})) & \text{if } t \text{ is minimal in } T \text{ w.r.t. } > \end{cases}$$

Termination is crucial to obtain a correct approach.

Lemma 2 [13] *For each μ-terminating term s, the term $\Phi_{\Delta,\mu}(s)$ is finite.*

Imposing that all the rules are strongly conservative we ensure that a variable appearing at a frozen position in the left-hand side of the rule never appears at an active position in the right-hand side of the rule (conservative property is not enough to ensure this statement).

Theorem 3 (Strongly Conservative Hierarchical Extension) *Let $[\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)] \longleftarrow [\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)]$ be a hierarchical extension of $(\mathscr{R}_1,\mu_1)$ where $\mathscr{R}_1 = (\mathscr{F}_1, R_1)$; if*

- *$(\mathscr{R}_1,\mu_1)$ is strongly conservative and $\mathscr{C}_\varepsilon$-terminating, and*
- *$(\mathscr{R}_2,\mu_2)$ is strongly conservative, and*
- *there is no infinite minimal CS-DP chain of $\mathsf{MDP}([\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)])$ relative to $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$,*

then $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ is terminating.

Proof 3 (Sketch) *By contradiction. Let us suppose that there is an infinite minimal CS-DP chain of $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$, then:*

- *there is an infinite minimal CS-DP chain of $\mathsf{MDP}([\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)])$ relative to $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$.*
- *$(\mathscr{R}_1,\mu_1)$ is not $\mathscr{C}_\varepsilon$-terminating, contradicting the hypothesis.*

We suppose that $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ is non-terminating, so there is an infinite minimal CS-DP chain of $\mathsf{MDP}([\mathscr{F}_1 \cup \mathscr{F}_2 \mid (\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)])$ relative to $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$. CS-DPs consist of:

1. *CS-DPs from $\mathsf{MDP}([\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)])$;*
2. *CS-DPs from $\mathsf{MDP}([\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)])$;*
3. *CS-DPs $\ell^\sharp \to s^\sharp$ such that $\ell \to r \in \mathscr{R}_2$, $r \trianglerighteq_\mu s$ and $\mathsf{root}(s) \in \mathscr{F}_1$.*

Using Lemma 1, we get an infinite minimal CS-DP chain where CS-DPs are:

i. from (2) only,

ii. from (1) only,

iii. from (2) in a finite number, then one pair from (3) and infinitely many pairs from (1).

- *Case (i): an infinite minimal CS-DP chain of $\mathsf{MDP}([\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)])$ relative to $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ exists, contradicting the hypothesis.*
- *Cases (ii)-(iii): an infinite minimal CS-DP chain of $\mathsf{MDP}([\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)])$ relative to $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$. In a similar way to [13], using Definition 9, we can construct an infinite CS-DP chain of $\mathsf{MDP}([\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)])$ relative to $(\mathscr{R}_1 \cup \mathscr{C}_\varepsilon, \mu_1 \cup \mu_{\mathscr{C}_\varepsilon})$ from an infinite minimal CS-DP chain of $\mathsf{MDP}([\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)])$ relative to $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ where only $\mathscr{F}_2$ symbols are not in Δ.*

■

Independent hierarchical extensions allow us to represent the union of composable CS-TRSs.

Theorem 4 (Independent Strongly Conservative Hierarchical Extension) *Let* $[\mathscr{F}_1 \mid (\mathscr{R}_1,\mu_1)] \longleftarrow [\mathscr{F}_2 \mid (\mathscr{R}_2,\mu_2)]$ *be a hierarchical extension of* $(\mathscr{R}_1,\mu_1)$ *where* $\mathscr{R}_1 = (\mathscr{F}_1, R_1)$*, and* $[\mathscr{F}_3 \mid (\mathscr{R}_3,\mu_3)]$ *be a module extending* $(\mathscr{R}_1,\mu_1)$ *independently of* $(\mathscr{R}_2,\mu_2)$*. If*

- $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *is strongly conservative and* $\mathscr{C}_\varepsilon$*-terminating,*
- $(\mathscr{R}_1 \cup \mathscr{R}_3, \mu_1 \cup \mu_3)$ *is strongly conservative, and*
- *there is no infinite minimal CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_3 \mid (\mathscr{R}_3,\mu_3)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_3 \cup \mathscr{C}_\varepsilon, \mu_1 \cup \mu_3 \cup \mu_{\mathscr{C}_\varepsilon})$*,*

then $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$ *is* $\mathscr{C}_\varepsilon$*-terminating.*

Proof 4 (Sketch) *By contradiction. Let us suppose that* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$ *is non-terminating, so there is an infinite minimal CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_1 \cup \mathscr{F}_2 \cup \mathscr{F}_3 \mid (\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \mu_3)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$*. Using Lemma 1, we know that CS-DP chains are:*

1. *CS-DP chains of* $\mathsf{MDP}([\mathscr{F}_3 \mid (\mathscr{R}_3,\mu_3)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$*;*
2. *CS-DPs chains of* $\mathsf{MDP}([\mathscr{F}_1 \cup \mathscr{F}_2 \mid (\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$*;*
3. *CS-DPs chains relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$ *consisting of a finite number of CS-DPs in* $\mathsf{MDP}([\mathscr{F}_3 \mid (\mathscr{R}_3,\mu_3)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$*, a CS-DP* $\ell^\sharp \to s^\sharp$ *such that* $\ell \to r \in \mathscr{R}_3$*,* $r \trianglerighteq_\mu s$ *and* $\mathsf{root}(s) \in \mathscr{F}_1 \cup \mathscr{F}_2$ *and infinitely many CS-DPs from* $\mathsf{MDP}([\mathscr{F}_1 \cup \mathscr{F}_2 \mid (\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)])$*.*

Using Lemma 1, we get an infinite minimal CS-DP chain where CS-DPs are:

- *Case (1): an infinite minimal CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_3 \mid (\mathscr{R}_3,\mu_3)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$*. In a similar way to [13], using Definition 9, we can construct an infinite CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_3 \mid (\mathscr{R}_3,\mu_3)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_3 \cup \mathscr{C}_\varepsilon, \mu_1 \cup \mu_3 \cup \mu_{\mathscr{C}_\varepsilon})$ *from an infinite minimal CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_3 \mid (\mathscr{R}_3,\mu_3)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$ *where only* $\mathscr{F}_2$ *symbols are not in* Δ*.*
- *Cases (2)-(3): an infinite minimal CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_1 \cup \mathscr{F}_2 \mid (\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$*. In a similar way to [13], using Definition 9, we can construct an infinite CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_1 \cup \mathscr{F}_2 \mid (\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{C}_\varepsilon, \mu_1 \cup \mu_2 \cup \mu_{\mathscr{C}_\varepsilon})$ *from an infinite minimal CS-DP chain of* $\mathsf{MDP}([\mathscr{F}_1 \cup \mathscr{F}_2 \mid (\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)])$ *relative to* $(\mathscr{R}_1 \cup \mathscr{R}_2 \cup \mathscr{R}_3, \mu_1 \cup \mu_2 \cup \mu_3)$ *where only* $\mathscr{F}_3$ *symbols are not in* Δ*.*

■

Without the strongly conservative restriction (even considering only conservative rules) we cannot ensure the previous results.

Example 6 [4] *Consider the following conservative CS-TRS:*

$$\mathscr{R}_1 = \quad \{\mathsf{f}(\mathsf{c}(x),x) \quad \to \quad \mathsf{f}(x,x)\} \qquad \mathscr{R}_2 = \quad \{\mathsf{b} \quad \to \quad \mathsf{c}(\mathsf{b})\}$$

where $\mu_1(\mathsf{f}) = \{1,2\}$*,* $\mu_1(\mathsf{c}) = \mu_2(\mathsf{c}) = \mu_2(\mathsf{b}) = \varnothing$*. We can see* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *as a hierarchichal extension of* $(\mathscr{R}_0,\mu_0) = ((\{\mathsf{c}\},\varnothing),\mu_0)$ *with module* $[\mathscr{F}_1 \mid \mathscr{R}_1]$ *and* $[\mathscr{F}_2 \mid \mathscr{R}_2]$ *is a module extending* $(\mathscr{R}_0,\mu_0)$ *independently of* $(\mathscr{R}_1,\mu_1)$*. The set of CS-DPs of* $(\mathscr{R}_1,\mu_1)$ *is* $\mathsf{DP}_1 = \{\mathsf{F}(\mathsf{c}(x),x) \to \mathsf{F}(x,x)\}$ *and the set of CS-DPs of* $(\mathscr{R}_2,\mu_2)$ *is* $\mathsf{DP}_2 = \varnothing$*. Both conservative CS-TRSs are* $\mathscr{C}_\varepsilon$*-terminating independently, but the union of these two CS-TRSs* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *generates an infinite CS-DP chain:*

$$\mathsf{F}(\mathsf{c}(\mathsf{b}),\mathsf{b}) \to_{\mathsf{DP}_1} \mathsf{F}(\mathsf{b},\mathsf{b}) \hookrightarrow_{\mathscr{R}_2,\mu_2} \mathsf{F}(\mathsf{c}(\mathsf{b}),\mathsf{b}) \to_{\mathsf{DP}_1} \cdots$$

The following example shows an application of our result.

Example 7 [5] *Consider the following example:*

$$\mathscr{R}_1 = \begin{array}{rcl} \{\mathsf{length}(\mathsf{nil}) & \to & \mathsf{0} \\ \mathsf{length}(x{:}y) & \to & \mathsf{s}(\mathsf{length1}(y)) \\ \mathsf{length1}(x) & \to & \mathsf{length}(x)\} \end{array} \qquad \mathscr{R}_2 = \{\mathsf{from}(x) \to x{:}\mathsf{from}(\mathsf{s}(x))\}$$

where $\mu_2(\mathsf{from}) = \mu_1(:) = \mu_2(:) = \mu_1(\mathsf{s}) = \mu_2(\mathsf{s}) = \{1\}$ *and* $\mu_1(\mathsf{length}) = \mu_1(\mathsf{length1}) = \mu_1(\mathsf{nil}) = \mu_1(\mathsf{0}) = \varnothing$. *We can see* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *as a hierarchichal extension of* $(\mathscr{R}_0, \mu_0) = ((\{:, \mathsf{s}\}, \varnothing), \mu_0)$ *with module* $[\mathscr{F}_1 \mid \mathscr{R}_1]$ *and* $[\mathscr{F}_2 \mid \mathscr{R}_2]$ *is a module extending* $(\mathscr{R}_0, \mu_0)$ *independently of* $(\mathscr{R}_1, \mu_1)$. *The set of CS-DPs of* $(\mathscr{R}_1, \mu_1)$ *is* $\mathsf{DP}_1 = \{\mathsf{LENGTH}(x{:}y) \to \mathsf{LENGTH1}(y), \mathsf{LENGTH1}(x) \to \mathsf{LENGTH}(x)\}$ *and the set of CS-DPs of* $(\mathscr{R}_2, \mu_2)$ *is* $\mathsf{DP}_2 = \varnothing$. *Both CS-TRSs are* $\mathscr{C}_\varepsilon$*-terminating independently and we can use the results of the paper to conclude that the union* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *is terminating.*

However, we still cannot deal with the leading example of the paper and we must overcome problems as the one showed in the following example.

Example 8 *Let us consider the following example:*

$$\mathscr{R}_1 = \{\mathsf{take}(x{:}y) \to \mathsf{take}(y)\} \qquad \mathscr{R}_2 = \{\mathsf{from}(x) \to x{:}\mathsf{from}(\mathsf{s}(x))\}$$

where $\mu_1(\mathsf{take}) = \mu_1(:) = \mu_2(:) = \mu_2(\mathsf{from}) = \mu_2(\mathsf{s}) = \{1\}$. *We can see* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *as a hierarchichal extension of* $(\mathscr{R}_0, \mu_0) = ((\{:\}, \varnothing), \mu_0)$ *with module* $[\mathscr{F}_1 \mid \mathscr{R}_1]$ *and* $[\mathscr{F}_2 \mid \mathscr{R}_2]$ *is a module extending* $(\mathscr{R}_0, \mu_0)$ *independently of* $(\mathscr{R}_1, \mu_1)$. *The set of CS-DPs of* $(\mathscr{R}_1, \mu_1)$ *is* $\mathsf{DP}_1 = \{\mathsf{TAKE}(x{:}y) \to \mathsf{TAKE}(y)\}$ *and the set of CS-DPs of* $(\mathscr{R}_2, \mu_2)$ *is* $\mathsf{DP}_2 = \varnothing$. *Both CS-TRSs are* $\mathscr{C}_\varepsilon$*-terminating independently, but the union of these two CS-TRSs* $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *generates an infinite CS-DP chain:*

$$\mathsf{TAKE}(x{:}\mathsf{from}(\mathsf{s}(x))) \to_{\mathsf{DP}_1} \mathsf{TAKE}(\mathsf{from}(\mathsf{s}(x))) \hookrightarrow_{\mathscr{R}_2, \mu_2} \mathsf{TAKE}(\mathsf{s}(x){:}\mathsf{from}(\mathsf{s}(\mathsf{s}(x)))) \to_{\mathsf{DP}_1} \cdots$$

where rules from $(\mathscr{R}_2, \mu_2)$ *are an important actor in the non-termination of the union of CS-TRSs.*

In a hierarchical extension, when we consider a terminating module which is nonterminating without the replacement map, the nonterminating behaviour can be due to new rules in the extended modules. These new rules take an important role in the adaptation of hierarchical extensions to arbitrary CS-TRSs.

5 Related Work

In [11], two results about modularity of CS-TRSs were given. One for the union of CS-TRSs with disjoint signatures and one for the union of CS-TRSs with shared constructors. In our work disjoint signature unions are not considered. For constructor sharing unions, they obtained the following result:

Theorem 5 [11] *Let* $(\mathscr{R}_1, \mu_1)$, $(\mathscr{R}_2, \mu_2)$ *be two constructor sharing, compatible, terminating CS-TRSs:*

1. $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *terminates if* $(\mathscr{R}_1, \mu_1)$ *and* $(\mathscr{R}_2, \mu_2)$ *are layer-preserving.*
2. $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *terminates if* $(\mathscr{R}_1, \mu_1)$ *and* $(\mathscr{R}_2, \mu_2)$ *are non-duplicating.*
3. $(\mathscr{R}_1 \cup \mathscr{R}_2, \mu_1 \cup \mu_2)$ *terminates if* $(\mathscr{R}_1, \mu_1)$ *or* $(\mathscr{R}_2, \mu_2)$ *is both, layer-preserving and non-duplicating.*

Layer preserving means that there is no rule $\ell \to r$ such that r is a variable or rooted by a shared constructor. This condition excludes rules like $\mathscr{R}_1$-rules in Example 8. A rule $\ell \to r$ is non-duplicating if for every $x \in \mathscr{V}ar(\ell)$ the multiset of replacing occurrencies of x in r is contained in the multiset of replacing occurrences of x in ℓ. A rule like $\mathsf{f}(x) \to \mathsf{g}(x,x)$ where $\mu(\mathsf{f}) = \{1\}$ and $\mu(\mathsf{g}) = \{1,2\}$ is strongly conservative but duplicating, hence, our results on strongly conservative modules are complementary to the ones obtained in [11].

6 Conclusions

In this paper we analyze modularity of termination for combinations of context-sensitive rewrite modules from the perspective of CS-DPs. The analysis shows that only in a very restrictive case (strong conservative hierarchical extension), the modularity results for term rewriting extends to CSR. When trying to generalize modularity results to arbitrary CS-TRSs, we find some counterexamples that force us to consider new restrictions in order to obtain a correct result. The main problem comes from modules that are nonterminating when removing the replacement map (those modules contain potential nonterminating μ-rewrite sequences that can appear by means of module hierarchical extensions). These modules with potential nonterminating rules must be considered to obtain a complete incremental and modular termination framework for CSR because these rules cannot be simulated by $\mathscr{C}_\varepsilon$-rules. Future work aims to analyze those problems and obtain a correct hierarchical extension results on arbitrary modules.

References

[1] B. Alarcón, R. Gutiérrez & S. Lucas (2006): *Context-Sensitive Dependency Pairs*. In S. Arun-Kumar & N. Garg, editors: *Proc. of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science, FST&TCS'06*, *LNCS* 4337, Springer-Verlag, pp. 297–308.

[2] B. Alarcón, R. Gutiérrez & S. Lucas (2007): *Improving the Context-Sensitive Dependency Graph*. *Electronic Notes in Theoretical Computer Science* 188, pp. 91–103. Proc. of the 6th Spanish Conference on Programming and Languages, PROLE'06.

[3] B. Alarcón, R. Gutiérrez & S. Lucas (2010): *Context-Sensitive Dependency Pairs*. *Information and Computation* 208, pp. 922–968.

[4] B. Alarcón & S. Lucas (2007): *Termination of Innermost Context-Sensitive Rewriting Using Dependency Pairs*. In F. Wolter, editor: *Proc. of the 6th International Symposium on Frontiers of Combining Systems, FroCoS'07*, *LNAI* 4720, Springer-Verlag, pp. 73–87.

[5] M. Alpuente, S. Escobar, B. Gramlich & S. Lucas (2010): *On-Demand Strategy Annotations Revisited: An Improved On-Demand Evaluation Strategy*. *Theoretical Computer Science* 411(2), pp. 504–541.

[6] T. Arts & J. Giesl (2000): *Termination of Term Rewriting Using Dependency Pairs*. *Theoretical Computer Science* 236(1–2), pp. 133–178.

[7] F. Baader & T. Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press.

[8] F. Durán, S. Lucas, C. Marché, J. Meseguer & X. Urbain (2008): *Proving Operational Termination of Membership Equational Programs*. *Higher Order Symbolic Computation* 21(1-2), pp. 59–88.

[9] J. Giesl & A. Middeldorp (1999): *Transforming Context-Sensitive Rewrite Systems*. In P. Narendran & M. Rusinowitch, editors: *Proc. of the 10th International Conference on Rewriting Techniques and Applications, RTA'99*, *LNCS* 1631, Springer-Verlag, pp. 271–285.

[10] B. Gramlich (1994): *Generalized Sufficient Conditions for Modular Termination of Rewriting*. *Applicable Algebra in Engineering, Comunication and Computing* 5, pp. 131–151.

[11] B. Gramlich & S. Lucas (2002): *Modular Termination of Context-Sensitive Rewriting*. In: *Proc. of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP'02*, ACM Press, pp. 50–61.

[12] R. Gutiérrez (2010): *Automatic Proofs of Termination of Context-Sensitive Rewriting*. Ph.D. thesis, Dept. de Sistemes Informàtics i Computació, Universitat Politècnica de València, Valencia, Spain.

[13] R. Gutiérrez, S. Lucas & X. Urbain (2008): *Usable Rules for Context-Sensitive Rewrite Systems*. In A. Voronkov, editor: *Proc. of the 19th International Conference on Rewriting Techniques and Applications, RTA'08*, *LNCS* 5117, Springer-Verlag, pp. 126–141.

[14] S. Lucas (1998): *Context-Sensitive Computations in Functional and Functional Logic Programs*. Journal of Functional and Logic Programming 1998(1), pp. 1–61.

[15] S. Lucas (2002): *Context-Sensitive Rewriting Strategies*. Information and Computation 178(1), pp. 293–343.

[16] Y. Toyama (1987): *Counterexamples to Termination for the Direct Sum of Term Rewriting Systems*. Information Processing Letters 25, pp. 141–143.

[17] X. Urbain (2004): *Modular & Incremental Automated Termination Proofs*. Journal of Automated Reasoning 32(4), pp. 315–355.

[18] H. Zantema (1997): *Termination of Context-Sensitive Rewriting*. In H. Comon, editor: *Proc. of the 7th International Conference on Rewriting Techniques and Applications, RTA'97*, *LNCS* 1232, Springer-Verlag, pp. 172–186.

Launchbury's semantics revisited: On the equivalence of context-heap semantics (Work in progress)

Lidia Sánchez-Gil
Facultad de Informática
Universidad Complutense de Madrid
España
lisanche@ucm.es

Mercedes Hidalgo-Herrero
Facultad de Educación
Universidad Complutense de Madrid
España
mhidalgo@ucm.es

Yolanda Ortega-Mallén
Facultad de CC. Matemáticas
Universidad Complutense de Madrid
España
yolanda@ucm.es

Launchbury's natural semantics for lazy evaluation is based on heaps of bindings, i.e., variable-expression pairs, which define the evaluation context. In order to prove the adequacy of the operational semantics with respect to a standard denotational one, Lauchbury defines an alternative natural semantics where updating of bindings is removed and β-reduction is done through indirections instead of variable substitution. We study how context heaps are affected by these changes, and we define several relations between heaps. These relations allow to establish the equivalence between Launchbury's natural semantics and its alternative version. This result is crucial because many authors have based their proofs on its veracity.

1 Motivation

More than twenty years have elapsed since Launchbury first presented in [9] his natural semantics for lazy evaluation, a key contribution to the semantic foundation for non-strict functional programming languages like Haskell or Clean. Throughout these years, Launchbury's natural semantics has been cited frequently and has inspired many further works as well as several extensions like in [2, 10, 18, 8]. The authors have extended in [13] Launchbury's semantics with rules for *parallel application* that creates new processes to distribute the computation; these distributed processes exchange values through communication channels. The success of Lanchbury's proposal resides in its simplicity. Expressions are evaluated with respect to a *context*, which is represented by a heap of *bindings*, that is, (variable, expression) pairs. This heap is explicitly managed to make possible the sharing of bindings, thus, modeling laziness.

In order to prove that this lazy (operational) semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury introduces some variations in the operational semantics. On the one hand, the update of bindings with their computed values is an operational notion without counterpart in the standard denotational semantics, so that the alternative natural semantics does no longer update bindings and becomes a *call-by-name* semantics. On the other hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds to the actual argument. For a closer approach to this mechanism, in the alternative operational semantics applications

Submitted to:
PROLE 2014

are carried out by introducing *indirections*, i.e., variables bound to variables, instead of by performing the β-reduction through substitution. Besides, the denotation "undefined" indicates that there is no value associated to the expression being evaluated, but there is no indication of the reason for that. By contrast, in the operational semantics there are two possibilities for not reaching a value: either the reduction gets blocked if no rule is applicable (*blackhole*), or the reduction never stops. The rules in the alternative semantics guarantee that reductions never reach a blackhole.

Unfortunately, the proof of the equivalence between the natural semantics and its alternative version is detailed nowhere, and a simple induction turns out to be insufficient. The *context-heap* semantics is too sensitive to the changes introduced by the alternative rules. Intuitively, both reduction systems should lead to the same results. However, this cannot be directly established since final values may contain free variables that are dependent on the context of evaluation, which is represented by the heap of bindings. The lack of update leads to the duplication of bindings, but is awkward to prove that duplicated bindings, as well as indirections, do not add relevant information to the context. Therefore, our challenge is to establish a way of relating the heaps and values obtained with each reduction system, and to prove that the semantics are equivalent, so that any reduction of a term in one of the systems has its counterpart in the other. To achieve this goal, indirections and update are considered separately giving place to two intermediate semantics. We focus on the one without update. We aim to proof the equivalence between it and the two semantics proposed by Launchbury. The proof that deals with indirections will soon appear in [17] while the relation involving update is currently in progress.

We want to identify terms up to α-conversion, but dealing with α-equated terms usually implies the use of Barendregt's variable convention [3] to avoid the renaming of bound variables. However, the use of the variable convention is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [19]). Moreover, we intend to formalize our results with the help of some proof assistant like Coq [4] or Isabelle [11]. Looking for a binding system susceptible of formalization, we have chosen a *locally nameless* representation (as presented by Charguéraud in [7]). This is a mixed notation where bound variable names are replaced by de Bruijn indices [6], while free variables preserve their names. This is suitable in our case because context heaps collect free variables whose names we are interested in preserving in order to identify them more easily. A locally nameless version of Launchbury's natural semantics has been presented by the authors in [14] and [15].

Others are revisiting Launchbury's semantics too. For instance, Breitner has formally proven in [5] the correctness of the natural semantics by using Isabelle's nominal package [20], and presently he is working on the formalization of the adequacy. While Breitner is exclusively interested in formalizing the proofs, we have a broader objective: To analyze the effect of introducing indirections in the context heaps, and the correspondence between heap/value pairs obtained with update and those produced without update. Furthermore, we want to prove the equivalence of the two operational semantics.

The paper is structured as follows: In the next section we give an overview of the mentioned locally nameless version of Launchbury's natural semantics and its alternative rules. We define two intermediate semantics: one introducing indirections, and the other eliminating updates and blackholes. Section 3 is dedicated to indirections, while in Section 4 we study the similarities and differences between the reductions proofs obtained with and without update of bindings. In the last section we draw conclusions and outline our future work.

$$\begin{array}{lcll} x & \in & Var & \\ e & \in & Exp & ::= \quad x \mid \lambda x.e \mid (e\ x) \mid \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e \end{array}$$

Figure 1: Named representation of the extended λ-calculus

$$\begin{array}{lll} x \in Id & & i, j \in \mathbb{N} \\ v \in Var & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\ t \in LNExp & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t \end{array}$$

Figure 2: Locally nameless syntax

2 A locally nameless representation

The language described in [9] is a lambda calculus extended with recursive local declarations. The abstract syntax, in the *named representation*, appears in Figure 1. Since there are two name binders, i.e., λ-abstraction and `let`-declaration, a quotient structure respect to α-equivalence is required. We avoid this by employing a *locally nameless representation* [7].

As mentioned above, our locally nameless representation has already been presented in [14] and [15]. Here we give only a brief presentation avoiding those technicalities that are not essential to the contributions of the present work.

2.1 Locally nameless syntax

The locally nameless version of the abstract syntax is shown in Figure 2. *Bound variables* and *free variables* are distinguished. Since `let`-declarations are multibinders, we have followed Charguéraud [7] and bound variables are represented with two natural numbers: the first number is a de Bruijn index that counts how many binders (abstraction or `let`) have been passed through in the syntactic tree to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind one variable, so that the second number should be zero.

Example 1. *Let $e \in Exp$ be the λ-expression given in the named representation*

$$e \equiv \lambda z.\texttt{let}\ \{x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2\}\ \texttt{in}\ (z\ x_2).$$

The corresponding locally nameless term $t \in LNExp$ is:

$$t \equiv \texttt{abs}\ (\texttt{let}\ \{\texttt{abs}\ (\texttt{bvar}\ 0\ 0), \texttt{abs}\ (\texttt{bvar}\ 0\ 0)\}\ \texttt{in}\ \texttt{app}\ (\texttt{bvar}\ 1\ 0)\ (\texttt{bvar}\ 0\ 1)).$$

Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms. The syntactic tree appears en Figure 3. □

This locally nameless syntax allows to build terms that have no corresponding named expression in *Exp* (Figure 1). For instance, when bound variables indices are out of range. The terms in *LNExp* that do match expressions in *Exp* are called *locally-closed*, written `lc` t. The *local closure* predicate is detailed in [15]. We avoid those technicalities that are not essential to the new contributions of this work.

In the following, a list like $\{t_i\}_{i=1}^{n}$ is represented as $\overline{t}$, with length $|\overline{t}| = n$. Later on, we use the notation $[t : \overline{t}]$ to represent a list with head t and tail $\overline{t}$, and $+\!\!+$ for the concatenation of lists.

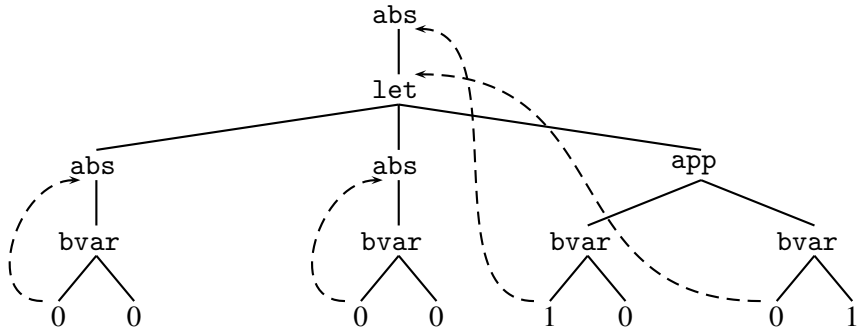


Figure 3: Syntax tree of Example 1.

We denote by $\mathtt{fv}(t)$ the set of *free variables* of a term t. A name $x \in Id$ is *fresh in a term* $t \in LNExp$, written $\mathtt{fresh}\ x\ \mathtt{in}\ t$, if x does not belong to the set of free variables of t, i.e., $x \notin \mathtt{fv}(t)$. Similarly, for a list of names, $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ t$ if $\overline{x} \notin \mathtt{fv}(t)$, where $\overline{x}$ represents a list of pairwise-distinct names in Id.

We say that two terms have the *same structure*, written $t \sim_S t'$, if they differ only in the names of their free variables.

Since there is no danger of name capture, *substitution* of variable names in a term is trivial in the locally nameless representation. We write $t[y/x]$ for replacing the occurrences of x by y in the term t. Clearly, name substitution preserves the structure of a term.

A *variable opening* operation is needed to manipulate locally nameless terms. This operation turns the outermost bound variables into free variables. The opening of a term $t \in LNExp$ with a list of names $\overline{x} \subseteq Id$ is denoted by $t^{\overline{x}}$. For simplicity, we write t^x for the variable opening with a unitary list $[x]$. A formal definition of variable opening can be found in [7] and [15]. Here we just illustrate the concept and its use with an example.

Example 2. *Let* $t \equiv \mathtt{abs}\ (\mathtt{let\ bvar\ 0\ 1, bvar\ 1\ 0\ in\ app}\ (\mathtt{abs\ bvar\ 2\ 0})\ (\mathtt{bvar\ 0\ 1}))$. *Hence, the body of the abstraction is:*

$$u \equiv \mathtt{let\ bvar\ 0\ 1,}\ \boxed{\mathtt{bvar\ 1\ 0}}\ \mathtt{in\ app}\ (\mathtt{abs}\ \boxed{\mathtt{bvar\ 2\ 0}})\ (\mathtt{bvar\ 0\ 1}).$$

But then in u the bound variables referring to the outermost abstraction (shown squared) point to nowhere. Therefore, we consider u^x *instead of u, where*

$$u^x \quad = \quad \mathtt{let\ bvar\ 0\ 1, fvar}\ x\ \mathtt{in\ app}\ (\mathtt{abs\ fvar}\ x)\ (\mathtt{bvar\ 0\ 1}).$$ □

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of a term is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct).

Example 3. *We close the term obtained by opening u in Example 2.*
Let $t \equiv \mathtt{let\ bvar\ 0\ 1, fvar}\ x\ \mathtt{in\ app}\ (\mathtt{abs\ fvar}\ x)\ (\mathtt{bvar\ 0\ 1})$, *then*

$${}^{\backslash x}t \quad = \quad \mathtt{let\ bvar\ 0\ 1, bvar\ 1\ 0\ in\ app}\ (\mathtt{abs\ bvar\ 2\ 0})\ (\mathtt{bvar\ 0\ 1}).$$ □

Notice that in the last example the closed term coincides with u, the body of the abstraction in Example 2 that was opened with x, although this is not always the case. Only under some conditions variable closing and variable opening are inverse operations: If the variables are fresh in t, then ${}^{\backslash\overline{x}}(t^{\overline{x}}) = t$, and if the term is locally closed, then $({}^{\backslash\overline{x}}t)^{\overline{x}} = t$.

$$\text{LNLAM}\quad \frac{\{\mathtt{ok}\ \Gamma\}\qquad \{\mathtt{lc}\ (\mathtt{abs}\ t)\}}{\Gamma : \mathtt{abs}\ t \Downarrow \Gamma : \mathtt{abs}\ t}$$

$$\text{LNVAR}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \cup \mathtt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : \mathtt{fvar}\ x \Downarrow (\Delta, x \mapsto w) : w}$$

$$\text{LNAPP}\quad \frac{\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u \qquad \Theta : u^x \Downarrow \Delta : w \qquad \{x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin \mathtt{dom}(\Delta)\}}{\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x) \Downarrow \Delta : w}$$

$$\text{LNLET}\quad \frac{\begin{array}{c}\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\,.\,[(\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{x}}) : w^{\overline{x}} \wedge {}^{\backslash \overline{x}}(\overline{s}^{\overline{x}}) = \overline{s} \wedge {}^{\backslash \overline{x}}(w^{\overline{x}}) = w] \\ \{\overline{y}^{|\overline{t}|} \notin L\}\end{array}}{\Gamma : \mathtt{let}\ \overline{t}\ \mathtt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{s}^{\overline{y}}) : w^{\overline{y}}}$$

Figure 4: Natural semantics with locally nameless representation

$$\text{ALNVAR}\quad \frac{(\Gamma, x \mapsto t) : t \Downarrow \Delta : w}{(\Gamma, x \mapsto t) : \mathtt{fvar}\ x \Downarrow \Delta : w}$$

$$\text{ALNAPP}\quad \frac{\begin{array}{c}\Gamma : t \Downarrow \Theta : \mathtt{abs}\ u \\ \forall y \notin L \subseteq Id\,.\,[(\Theta, y \mapsto \mathtt{fvar}\ x) : u^y \Downarrow ([y : \overline{z}] \mapsto \overline{s}^y) : w^y \wedge {}^{\backslash y}(\overline{s}^y) = \overline{s} \wedge {}^{\backslash y}(w^y) = w] \\ \{x \notin \mathtt{dom}(\Gamma) \Rightarrow x \notin [z : \overline{z}]\} \qquad \{z \notin L\}\end{array}}{\Gamma : \mathtt{app}\ t\ (\mathtt{fvar}\ x) \Downarrow ([z : \overline{z}] \mapsto \overline{s}^z) : w^z}$$

Figure 5: Alternative rules with locally nameless representation

2.2 Locally nameless semantics

In the natural semantics defined by Launchbury [9] judgements are of the form $\Gamma : t \Downarrow \Delta : w$, that is, the term t in the context of the heap Γ reduces to the value w in the context of the (modified) heap Δ. *Values* ($w \in Val$) are terms in weak-head-normal-form (*whnf*) and *heaps* are collections of *bindings*, i.e., pairs (variable, term). A binding $(\mathtt{fvar}\ x, t)$ with $x \in Id$ and $t \in LNExp$ is represented by $x \mapsto t$. In the following, we represent a heap $\{x_i \mapsto t_i\}_{i=1}^n$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of the locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\mathtt{dom}(\Gamma)$, collects the names that are defined in the heap, so that $\mathtt{dom}(\overline{x} \mapsto \overline{t}) = \overline{x}$. By contrast, the function `names` returns the set of all the names that appear in a heap, i.e., the names occurring either in the domain or in the terms in the right-hand side of the bindings. This is used to define a freshness predicate for heaps: $\mathtt{fresh}\ \overline{x}\ \mathtt{in}\ \Gamma = \overline{x} \notin \mathtt{names}(\Gamma)$.

In a well-formed heap names are defined at most once and terms are locally closed. We write $\mathtt{ok}\ \Gamma$ to indicate that a heap is well-formed.

In Figure 4 we show a locally nameless representation of the rules for the natural semantics for lazy evaluation, given by Launchbury in [9]. For clarity, in the rules we put in braces the side-conditions to better distinguish them from the judgements.

To prove the computational adequacy of the natural semantics (Figure 4) with respect to a standard denotational semantics, Launchbury introduces alternative rules for variables and applications, whose locally nameless version is shown in Figure 5. Observe that the ALNVAR rule does not longer update the binding for the variable being evaluated, namely x. Besides, the binding for x does not disappear from the heap where the term bound to x is to be evaluated; therefore, any further reference to x leads to an infinite reduction. The effect of ALNAPP is the addition of an indirection $y \mapsto \mathtt{fvar}\ x$ instead of

	NS	INS	NNS	ANS
Indirections	✗	✓	✗	✓
Update	✓	✓	✗	✗
Blackholes	✓	✓	✗	✗

Figure 6: The lazy natural semantics and its alternatives

performing the β-reduction by substitution, as in u^x in LNAPP.

In the rules LNLET and ALNAPP we use *cofinite quantification* [1], which is an alternative to "exists-fresh" quantifications that provides stronger induction and inversion principles. Although there are not explicit freshness side-conditions in the rules, the finite set L represents somehow the names that should be avoided during a reduction proof. We use the variable opening to express that the final heap and value may depend on the chosen names. For instance, in LNLET, $w^{\overline{x}}$ indicates that the final value depends on the names $\overline{x}$, but there is a common basis w. Moreover, it is required that this basis does not contain occurrences of $\overline{x}$; this is is expressed by $^{\backslash\overline{x}}(w^{\overline{x}}) = w$. A detailed explanation of these semantic rules can be found in [14, 15].

In the following, the natural semantics (rules in Figure 4) is referred as NS, and the alternative semantics (rules LNLAM, LNLET and those in Figure 5) as ANS. We write $\Downarrow^A$ for reductions in ANS. Launchbury proves in [9] the correctness of NS with respect to a standard denotational semantics, and a similar result for ANS is easily obtained (as the authors of this paper have done in [12]). Therefore, NS and ANS are "denotationally" equivalent in the sense that if an expression is reducible (in some heap context) by both semantics then the obtained values have the same denotation. But this is insufficient for our purposes, because we want to ensure that if for some (heap : term) pair a reduction exists in any of the semantics, then there must exist a reduction in the other too, and the final heaps must be related. The changes introduced by ANS might seem to involve no serious difficulties to prove the latter result. Unfortunately things are not so easy. On the one hand, the alternative rule for variables transforms the original call-by-need semantics into a call-by-name semantics because bindings are not updated and computed values are no longer shared. Moreover, in the original semantics the reduction of a self-reference gets blocked (*blackhole*), while in the alternative semantics self-references yield infinite reductions. On the other hand, the addition of indirections complicates the task of comparing the (heap : value) pairs obtained by each reduction system, as one may need to follow a chain of indirections to get the term bound to a variable. We deal separately with each modification and introduce two intermediate semantics: (1) the *No-update Natural Semantics* (NNS) with the rules of NS (Figure 4) except for the variable rule, that corresponds to the one in the alternative version, i.e., ALNVAR in Figure 5; and (2) the *Indirection Natural Semantics* (INS) with the rules of NS but for the application rule, that corresponds to the alternative ALNAPP rule in Figure 5. We use $\Downarrow^N$ to represent reductions of NNS and $\Downarrow^I$ for those of INS. Figure 6 resumes the characteristics of the four natural semantics explained above.

It is guaranteed that the judgements produced by the locally nameless rules given in Figures 4 and 5 involve only well-formed heaps and locally closed terms. Furthermore, the reduction systems corresponding to these rules verify a number of interesting properties proved in [15]. We just show here the *renaming* lemma, that ensures that the evaluation of a term is independent of the names chosen during the reduction process. Further, any name defined in the context heap can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reductions for (heap : term) pairs are unique up to α-conversion of the names defined in the heap.

Lemma 1. *(Renaming)*

1. $\Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma,\Delta,t,w \Rightarrow \Gamma[y/x] : t[y/x] \Downarrow^K \Delta[y/x] : w[y/x]$;
2. $\Gamma : t \Downarrow^K \Delta : w \wedge \texttt{fresh}\ y\ \texttt{in}\ \Gamma,\Delta,t,w \wedge x \notin \texttt{dom}(\Gamma) \wedge x \in \texttt{dom}(\Delta) \Rightarrow \Gamma : t \Downarrow^K \Delta[y/x] : w[y/x]$,

where $\Gamma[y/x]$ indicates that name substitution is done in the left and right hand sides of the heap Γ, and $\Downarrow^K$ represents $\Downarrow$, $\Downarrow^A$, $\Downarrow^I$ and $\Downarrow^N$.

3 Indirections

The aim in this section is to prove the equivalence of NNS and ANS. After the evaluation of a term in a given context, each semantics yields a different binding heap. It is necessary to analyze their differences, which lie in the indirections introduced by ANS. An *indirection* is a binding of the form $x \mapsto \texttt{fvar}\ y$, that is, it just redirects to another variable name. The set of indirections of a heap Γ is denoted by $\texttt{Ind}(\Gamma)$.

The next example illustrates the situation.

Example 4. *Let us evaluate the term*

$$t \equiv \ \texttt{let abs (bvar 0 0) in app (abs } s\texttt{) (bvar 0 0)},$$

where

$$s \equiv \ \texttt{let abs (bvar 0 0), app (bvar 0 0) (bvar 1 0) in abs (bvar 0 0)}$$

in the empty context $\Gamma = \emptyset$.

$$\begin{array}{lll} \Gamma : t & \Downarrow^N & \{x_0 \mapsto \texttt{abs (bvar 0 0)}, x_1 \mapsto \texttt{abs (bvar 0 0)}, x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } x_0\texttt{)}\} \\ & & : \texttt{abs (bvar 0 0)} \\ \Gamma : t & \Downarrow^A & \{x_0 \mapsto \texttt{abs (bvar 0 0)}, x_1 \mapsto \texttt{abs (bvar 0 0)}, x_2 \mapsto \texttt{app (fvar } x_1\texttt{) (fvar } y\texttt{)}, y \mapsto \texttt{(fvar } x_0\texttt{)}\} \\ & & : \texttt{abs (bvar 0 0)} \end{array}$$

The value produced is the same in both cases. Yet, when comparing the final heap in $\Downarrow^A$ *with the final heap in* $\Downarrow^N$*, we observe that there is an extra indirection,* $y \mapsto \texttt{fvar}\ x_0$*. This indirection corresponds to the binding introduced by* ALNApp *to reduce the application in the term* t*.* □

The previous example gives a hint of how to establish a relation between the heaps that are obtained with NNS and those produced by ANS: Two heaps are related if one can be obtained from the other by eliminating some indirections. For this purpose we define how to remove indirections from a heap, while preserving the evaluation context represented by that heap.

$$\begin{array}{rcl} (\emptyset, x \mapsto \texttt{fvar}\ y) \ominus x & = & \emptyset \\ ((\Gamma, z \mapsto t), x \mapsto \texttt{fvar}\ y) \ominus x & = & ((\Gamma, x \mapsto \texttt{fvar}\ y) \ominus x, z \mapsto t[y/x]) \end{array}$$

This definition can be generalized to remove a sequence of indirections from a heap:

$$\Gamma \ominus [\,] \ = \ \Gamma \qquad\qquad \Gamma \ominus [x : \overline{x}] \ = \ (\Gamma \ominus x) \ominus \overline{x}$$

3.1 Context equivalence

The meaning of a term depends on the meaning of its free variables. However, if a free variable is not defined in the context of evaluation of a term, then the name of this free variable is irrelevant. Therefore, we consider that two terms are equivalent in a given context if they only differ in the names of the free variables that do not belong to the context.

Definition 1. *Let $V \subseteq Id$, and $t, t' \in LNExp$. We say that t and t' are* context-equivalent in V, *written $t \approx^V t'$, when*

$$\text{CE-BVAR}\quad \frac{}{(\texttt{bvar}\ i\ j) \approx^V (\texttt{bvar}\ i\ j)} \qquad \text{CE-FVAR}\quad \frac{x, x' \notin V \vee x = x'}{(\texttt{fvar}\ x) \approx^V (\texttt{fvar}\ x')}$$

$$\text{CE-ABS}\quad \frac{t \approx^V t'}{(\texttt{abs}\ t) \approx^V (\texttt{abs}\ t')} \qquad \text{CE-APP}\quad \frac{t \approx^V t' \quad v \approx^V v'}{(\texttt{app}\ t\ v) \approx^V (\texttt{app}\ t'\ v')}$$

$$\text{CE-LET}\quad \frac{|\overline{t}| = |\overline{t'}| \quad \overline{t} \approx^V \overline{t'} \quad t \approx^V t'}{(\texttt{let}\ \overline{t}\ \texttt{in}\ t) \approx^V (\texttt{let}\ \overline{t'}\ \texttt{in}\ t')}$$

Fixed the set of names V, $\approx^V$ is an equivalence relation on *LNExp*. Based on this equivalence on terms, we define a family of equivalences on heaps, where two heaps are considered equivalent when they have the same domain and the corresponding closures may differ only in the free variables not defined in a given context:

Definition 2. *Let $V \subseteq Id$, and $\Gamma, \Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-context-equivalent in V, *written $\Gamma \approx^V \Gamma'$, when*

$$\text{HCE-EMPTY}\quad \frac{}{\emptyset \approx^V \emptyset} \qquad \text{HCE-CONS}\quad \frac{\Gamma \approx^V \Gamma' \quad t \approx^V t' \quad \texttt{lc}\ t \quad x \notin \texttt{dom}(\Gamma)}{(\Gamma, x \mapsto t) \approx^V (\Gamma', x \mapsto t')}$$

There is an alternative characterization for heap-context-equivalence which expresses that two heaps are context-equivalent whenever they are well-formed, have the same domain, and each pair of corresponding bound terms is context-equivalent.

Lemma 2. $\Gamma \approx^V \Gamma' \Leftrightarrow \texttt{ok}\ \Gamma \wedge \texttt{ok}\ \Gamma' \wedge \texttt{dom}(\Gamma) = \texttt{dom}(\Gamma') \wedge (x \mapsto t \in \Gamma \wedge x \mapsto t' \in \Gamma' \Rightarrow t \approx^V t')$.

Considering context-equivalence on heaps, we are particularly interested in the case where the context coincides with the domain of the heaps:

Definition 3. *Let $\Gamma, \Gamma' \in LNHeap$. We say that Γ and Γ' are* heap-equivalent, *written $\Gamma \approx \Gamma'$, if they are heap-context-equivalent in* $\texttt{dom}(\Gamma)$, *i.e., $\Gamma \approx^{\texttt{dom}(\Gamma)} \Gamma'$.*

If equivalent heaps are obtained by removing different sequences of indirections, then these must be the same up to permutation:

Lemma 3. $\texttt{ok}\ \Gamma \wedge \overline{x}, \overline{y} \subseteq \texttt{Ind}(\Gamma) \Rightarrow (\Gamma \ominus \overline{x} \approx \Gamma \ominus \overline{y} \Leftrightarrow \overline{y} \in \mathscr{S}(\overline{x}))$,
where $\mathscr{S}(\overline{x})$ denotes the set of all permutations of $\overline{x}$.

3.2 Indirection relation

Coming back to the idea of Example 4, where a heap can be obtained from another by just removing some indirections, we define the following relation on heaps:

Definition 4. *Let $\Gamma, \Gamma' \in LNHeap$. We say that Γ is* indirection-related *to Γ', written $\Gamma \gtrsim_I \Gamma'$, when*

$$\text{IR-HE}\quad \frac{\Gamma \approx \Gamma'}{\Gamma \gtrsim_I \Gamma'} \qquad \text{IR-IR}\quad \frac{\texttt{ok}\ \Gamma \quad \Gamma \ominus x \gtrsim_I \Gamma' \quad x \in \texttt{Ind}(\Gamma)}{\Gamma \gtrsim_I \Gamma'}$$

There is an alternative characterization for the relation $\gtrsim_I$ which expresses that a heap is indirection-related to another whenever the later can be obtained from the former by removing a sequence of indirections.

Proposition 1. $\Gamma \gtrsim_I \Gamma' \Leftrightarrow \mathtt{ok}\,\Gamma \wedge \exists\,\overline{x} \subseteq \mathtt{Ind}(\Gamma)\,.\,(\Gamma \ominus \overline{x}) \approx \Gamma'$.

By Lemma 3, the sequence of indirections is unique up to permutations, and it corresponds to the difference between the domains of the related heaps:

Corollary 1. $\Gamma \gtrsim_I \Gamma' \Rightarrow (\Gamma \ominus (\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma'))) \approx \Gamma'$. [1]

The *indirection-relation* is a preorder on the set of well-formed heaps. We extended the relation to (heap : term) pairs:

Definition 5. *Let* $\Gamma, \Gamma' \in$ *LNHeap, and* $t, t' \in$ *LNExp. We say that* $(\Gamma : t)$ *is* indirection-related *to* $(\Gamma' : t')$*, written* $(\Gamma : t) \gtrsim_I (\Gamma' : t')$*, if*

$$\text{IR-HT} \quad \frac{\forall z \notin L \subseteq Id\,.\,(\Gamma, z \mapsto t) \gtrsim_I (\Gamma', z \mapsto t')}{(\Gamma : t) \gtrsim_I (\Gamma' : t')}$$

We use cofinite quantification instead of adding freshness conditions on the new name z.

It is easy to prove that two (heap : term) pairs are indirection-related only if the heaps are indirection related and the terms have the same structure:

Lemma 4. $(\Gamma : t) \gtrsim_I (\Gamma' : t') \Rightarrow \Gamma \gtrsim_I \Gamma' \wedge t \sim_S t'$.

We illustrate these definitions with an example.

Example 5. *Let us consider the following heap and term:*

$$\begin{array}{lcl} \Gamma & = & \{x_0 \mapsto \mathtt{fvar}\,x_1, x_1 \mapsto \mathtt{abs}\,(\mathtt{bvar}\,0\,0), x_2 \mapsto \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_0)\,(\mathtt{bvar}\,0\,0)), \\ & & \quad y_0 \mapsto \mathtt{fvar}\,x_2\} \\ t & = & \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_0)\,\mathtt{bvar}\,0\,0) \end{array}$$

The (heap : term) pairs related with $(\Gamma : t)$ *are obtained by removing the sequences of indirections* $[]$*,* $[y_0]$*,* $[x_0]$*, and* $[x_0, y_0]$*:*

a) $\{x_0 \mapsto \mathtt{fvar}\,x_1, x_1 \mapsto \mathtt{abs}\,(\mathtt{bvar}\,0\,0), x_2 \mapsto \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_0)\,(\mathtt{bvar}\,0\,0)), y_0 \mapsto \mathtt{fvar}\,x_2\}$
$: \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_0)\,(\mathtt{bvar}\,0\,0))$

b) $\{x_0 \mapsto \mathtt{fvar}\,x_1, x_1 \mapsto \mathtt{abs}\,(\mathtt{bvar}\,0\,0), x_2 \mapsto \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_0)\,(\mathtt{bvar}\,0\,0))\}$
$: \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_0)\,(\mathtt{bvar}\,0\,0))$

c) $\{x_1 \mapsto \mathtt{abs}\,(\mathtt{bvar}\,0\,0), x_2 \mapsto \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_1)\,(\mathtt{bvar}\,0\,0)), y_0 \mapsto \mathtt{fvar}\,x_2\}$
$: \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_1)\,(\mathtt{bvar}\,0\,0))$

d) $\{x_1 \mapsto \mathtt{abs}\,(\mathtt{bvar}\,0\,0), x_2 \mapsto \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_1)\,(\mathtt{bvar}\,0\,0))\}$
$: \mathtt{abs}\,(\mathtt{app}\,(\mathtt{fvar}\,x_1)\,(\mathtt{bvar}\,0\,0))$

□

Now we are ready to establish the equivalence between ANS and NNS in the sense that if a reduction proof can be obtained with ANS for some term in a given context heap, then there must exist a reduction proof in NNS for the same (heap : term) pair such that the final (heap : value) is indirection-related to the final (heap : value) obtained with ANS, and vice versa.

Theorem 1. *(Equivalence ANS-NNS).*

1. $\Gamma : t \Downarrow^A \Delta_A : w_A \Rightarrow$
$\exists \Delta_N \in$ *LNHeap*$\,.\,\exists w_N \in$ *Val*$\,.\,\Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_A : w_A) \gtrsim_I (\Delta_N : w_N)$.

[1] Since the ordering of indirections is irrelevant, $\mathtt{dom}(\Gamma) - \mathtt{dom}(\Gamma')$ represents any sequence with the names defined in Γ but undefined in Γ'.

2. $\Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow$
$\exists \Delta_A \in \mathit{LNHeap} . \exists w_A \in \mathit{Val} . \exists \overline{x} \subseteq \mathtt{dom}(\Delta_N) - \mathtt{dom}(\Gamma) . \exists \overline{y} \subseteq \mathit{Id} .$
$|\overline{x}| = |\overline{y}| \wedge \Gamma : t \Downarrow^A \Delta_A : w_A \wedge (\Delta_A : w_A) \succsim_I (\Delta_N[\overline{y}/\overline{x}] : w_N[\overline{y}/\overline{x}])$.

Notice that in the second part of the theorem, i.e., from NNS to ANS, a renaming may be needed. This renaming only affects the names that are added to the heap during the reduction process. This is due to the fact that in NNS names occurring in the evaluation term (that is t in the theorem) may disappear during the evaluation and, consequently, they may be chosen on some application of the rule LNLET and added to the final heap. This cannot happen in ANS due to the alternative application rule.

The proof of Theorem 1 is not straightforward and induction cannot be applied directly. Several intermediate results are needed to prove a generalization of the theorem where instead of evaluating the same term in the same initial context heap, indirection-related initial (heap : term) pairs are considered. This is developed in detail in [16], and a reduced version will soon appear in [17].

4 No update

In this section we compare (heap : term) pairs obtained with NS, where bindings are updated with the values obtained during reduction, with those obtained with NNS, without update, and where infinite reductions may occur due to self-references. We start with an example.

Example 6. *Let us consider the following term:*

$$
\begin{array}{rcll}
t & \equiv & \mathtt{let} & \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \\
 & & & \mathtt{let} \quad \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1) \\
 & & & \mathtt{in} \quad \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1), \\
 & & & \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1) \\
 & & \mathtt{in} & \mathtt{app}\ (\mathtt{app}\ (\mathtt{bvar}\ 0\ 1)\ (\mathtt{bvar}\ 0\ 0))\ (\mathtt{bvar}\ 0\ 2)
\end{array}
$$

When the term t is evaluated in the context of the empty heap the following final (heap : value) pairs are obtained:

$$
\begin{array}{ll}
\Downarrow & \{x_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), x_2 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \\
 & \ y_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ y_0)\ (\mathtt{fvar}\ y_1)\} \\
 & : \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0)
\end{array}
$$

$$
\begin{array}{ll}
\Downarrow^N & \{x_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \\
 & \ x_1 \mapsto \ \mathtt{let} \quad \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1) \\
 & \qquad\ \ \mathtt{in} \quad \mathtt{app}\ (\mathtt{bvar}\ 0\ 0)\ (\mathtt{bvar}\ 0\ 1), \\
 & \ x_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ x_0)\ (\mathtt{fvar}\ x_1), \\
 & \ y_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ y_0)\ (\mathtt{fvar}\ y_1), \\
 & \ y'_0 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y'_1 \mapsto \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0), y'_2 \mapsto \mathtt{app}\ (\mathtt{fvar}\ y'_0)\ (\mathtt{fvar}\ y'_1)\} \\
 & : \mathtt{abs}\ (\mathtt{bvar}\ 0\ 0)
\end{array}
$$

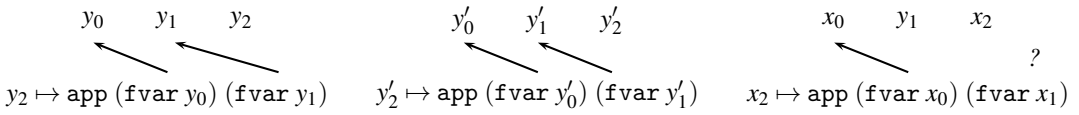
The inner `let`*-declaration, which is bound to the name x_1, is required twice. In the case of NS ($\Downarrow$), the evaluation of x_1 entails the introduction of three new names (y_0, y_1 and y_2), and the binding for x_1 is updated with the value obtained for the body term. Thus, the second time x_1 is required it is not re-evaluated. This is not the case in NNS ($\Downarrow^N$), where a second evaluation of x_1 implies the introduction of duplicated names (y'_0, y'_1 and y'_2) in the heap.* □

In the previous example one can observe the two main differences between the final heaps obtained by evaluating an expression with NS and with NNS. On the one hand, some variables that are bound to *whnf* values in NS remain bound to their initial terms in NNS. On the other hand, when evaluating with NNS, more bindings are obtained with respect to NS. The "extra" bindings are produced by duplicated evaluations of `let`-declarations. Therefore, to relate the final heaps we proceed in two steps: First we remove the extra bindings of the final heap of NNS to obtain a heap with the same domain as the one obtained with NS; second we check that the bindings that have not been updated are "equivalent" to the corresponding updated bindings.

4.1 Group relation

The first step is to identify duplicated bindings, i.e., those that correspond to the re-evaluation of a `let`-declaration. The next example illustrates the problem.

Example 7. *We choose three groups of bindings from the heap obtained with the $\Downarrow^N$-reduction in Example 6:* $\overline{y} = [y_0, y_1, y_2]$, $\overline{y'} = [y'_0, y'_1, y'_2]$ *and* $\overline{z} = [x_0, y_1, x_2]$.



We observe that y_0, y'_0 and x_0 are bound to terms with the same structure. *Similarly for y_1, y'_1 and y_1, and for y_2, y'_2 and x_2. But a closer look detects that $[x_0, y_1, x_2]$ is different from the other two groups: If the terms bound in each group are closed with the names of that group, then equal terms are obtained in the first two groups, while a different term is obtained in the third group:*

$$
\begin{array}{lllllllll}
y_0 & \xrightarrow{\backslash \overline{y}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y_1 & \xrightarrow{\backslash \overline{y}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y_2 & \xrightarrow{\backslash \overline{y}} & \texttt{app}\ (\texttt{bvar}\ 0\ 0)\ (\texttt{bvar}\ 0\ 1) \\
y'_0 & \xrightarrow{\backslash \overline{y'}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y'_1 & \xrightarrow{\backslash \overline{y'}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y'_2 & \xrightarrow{\backslash \overline{y'}} & \texttt{app}\ (\texttt{bvar}\ 0\ 0)\ (\texttt{bvar}\ 0\ 1) \\
x_0 & \xrightarrow{\backslash \overline{z}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & y_1 & \xrightarrow{\backslash \overline{z}} & \texttt{abs}\ (\texttt{bvar}\ 0\ 0) & x_2 & \xrightarrow{\backslash \overline{z}} & \texttt{app}\ (\texttt{bvar}\ 0\ 0)\ (\texttt{fvar}\ x_1)
\end{array}
$$

Therefore, groups $\overline{y}$ and $\overline{y'}$ should be related, but not with $\overline{z}$. □

We start by relating terms (with respect to two lists of names) that are equal except for the free variables, and those names that are different occupy the same position in their respective lists.

Definition 6. *Let $t, t' \in LNExp$ and $\overline{x}, \overline{y} \subseteq Id$. We say that t and t' are* context-group-related *in the contexts of $\overline{x}$ and $\overline{y}$, written $t \approx^{(\overline{x},\overline{y})} t'$, when:*

$$\text{CR-BVAR}\quad \frac{|\overline{x}| = |\overline{y}|}{(\texttt{bvar}\ i\ j) \approx^{(\overline{x},\overline{y})} (\texttt{bvar}\ i\ j)} \qquad \text{CR-ABS}\quad \frac{t \approx^{(\overline{x},\overline{y})} t'}{(\texttt{abs}\ t) \approx^{(\overline{x},\overline{y})} (\texttt{abs}\ t')}$$

$$\text{CR-FVAR1}\quad \frac{|\overline{x}| = |\overline{y}| \quad x \notin \overline{x} \cup \overline{y}}{(\texttt{fvar}\ x) \approx^{(\overline{x},\overline{y})} (\texttt{fvar}\ x)} \qquad \text{CR-APP}\quad \frac{t \approx^{(\overline{x},\overline{y})} t' \quad v \approx^{(\overline{x},\overline{y})} v'}{(\texttt{app}\ t\ v) \approx^{(\overline{x},\overline{y})} (\texttt{app}\ t'\ v')}$$

$$\text{CR-FVAR2}\quad \frac{|\overline{x}| = |\overline{y}| \quad x = \texttt{List.nth}\ i\ \overline{x} \quad y = \texttt{List.nth}\ i\ \overline{y}}{(\texttt{fvar}\ x) \approx^{(\overline{x},\overline{y})} (\texttt{fvar}\ y)} \qquad \text{CR-LET}\quad \frac{|\overline{t}| = |\overline{t'}| \quad \overline{t} \approx^{(\overline{x},\overline{y})} \overline{t'} \quad t \approx^{(\overline{x},\overline{y})} t'}{(\texttt{let}\ \overline{t}\ \texttt{in}\ t) \approx^{(\overline{x},\overline{y})} (\texttt{let}\ \overline{t'}\ \texttt{in}\ t')}$$

An alternative to the definition above is to check that the terms are equal under closure in their respective contexts:

Lemma 5. $t \approx^{(\overline{x},\overline{y})} t' \Leftrightarrow [t \sim_S t' \wedge {}^{\setminus \overline{x}}t = {}^{\setminus \overline{y}}t' \wedge |\overline{x}| = |\overline{y}|]$

Next we relate heaps that differ in duplicated groups of bindings.

Definition 7. *Let* $\Gamma,\Gamma' \in$ *LNHeap. We say that* Γ *is* group-related *to* Γ'*, written* $\Gamma \succsim_G \Gamma'$*, when*

$$\text{GR-EQ}\quad \frac{}{\Gamma \succsim_G \Gamma} \qquad \text{GR-GR}\quad \frac{\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s} \qquad \overline{x} \cap \overline{y} = \emptyset \qquad (\Gamma, \overline{x} \mapsto \overline{t})[\overline{x}/\overline{y}] \succsim_G \Gamma'}{(\Gamma, \overline{x} \mapsto \overline{t}, \overline{y} \mapsto \overline{s}) \succsim_G \Gamma'}$$

This relation is a partial order on heaps. We extend it to (heap : term) pairs:

Definition 8. *Let* $\Gamma,\Gamma' \in$ *LNHeap, and* $t,t' \in$ *LNExp. We say that* $(\Gamma : t)$ *is* group-related *to* $(\Gamma' : t')$*, written* $(\Gamma : t) \succsim_G (\Gamma' : t')$*, when*

$$\text{GR\_HT\_EQ}\quad \frac{}{(\Gamma : t) \succsim_G (\Gamma : t)}$$

$$\text{GR\_HT\_GR}\quad \frac{\overline{t} \approx^{(\overline{x},\overline{y})} \overline{s} \qquad \overline{x} \cap \overline{y} = \emptyset \qquad ((\Gamma, \overline{x} \mapsto \overline{t})[\overline{x}/\overline{y}] : t[\overline{x}/\overline{y}]) \succsim_G (\Gamma' : t')}{((\Gamma, \overline{x} \mapsto \overline{t}, \overline{y} \mapsto \overline{s}) : t) \succsim_G (\Gamma' : t')}$$

Group-related (heap : term) pairs are equivalent in the sense that if there exists a NNS-reduction for one of them, then there also exists a NNS-reduction for the other, so that the final (heap : value) pairs are group-related too.

Lemma 6. $(\Gamma : t) \succsim_G (\Gamma' : t') \wedge \Gamma' : t' \Downarrow^N \Delta' : w' \Rightarrow$
$\exists \Delta \in$ *LNHeap*, $w \in$ *Val*$.\, \Gamma : t \Downarrow^N \Delta : w \wedge (\Delta : w) \succsim_G (\Delta' : w')$.

4.2 Update relation

Once all the duplicated groups of names have been detected and eliminated, we have to deal with updating. For this, we check that those bindings in the no-updated heap, which have unevaluated expressions do evaluate to values "equivalent" to those in the updated heap. For a recursive definition, we fix an initial context heap for these evaluations.

Definition 9. *Let* $\Gamma,\Gamma',\Delta \in$ *LNHeap. We say that* Γ *is* update-related *to* Γ' in the context of Δ*, written* $\Gamma \sim_U^\Delta \Gamma'$*, when*

$$\text{UCR\_EQ}\quad \frac{}{\Gamma \sim_U^\Delta \Gamma} \qquad \text{UCR\_VT}\quad \frac{\Gamma \sim_U^\Delta \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma, x \mapsto t) \sim_U^\Delta (\Gamma', x \mapsto w')}$$

Notice that, by definition, update related heaps have the same domain. We are particularly interested in the case where the context coincides with the first heap:

Definition 10. *Let* $\Gamma,\Gamma' \in$ *LNHeap. We say that* Γ *is* update-related *to* Γ'*, written* $\Gamma \sim_U \Gamma'$*, if* $\Gamma \sim_U^\Gamma \Gamma'$.

Once again we extend these definitions to (heap : term) pairs:

Definition 11. *Let* $\Gamma,\Gamma',\Delta \in$ *LNHeap, and* $t,t' \in$ *LNExp. We say that* $(\Gamma : t)$ *is* update-related *to* $(\Gamma' : t')$ in the context of Δ*, written* $(\Gamma : t) \sim_U^\Delta (\Gamma : t')$*, when*

$$\text{UCR\_TT\_HT}\quad \frac{\Gamma \sim_U^\Delta \Gamma'}{(\Gamma : t) \sim_U^\Delta (\Gamma' : t)}$$

$$\text{UCR\_VT\_HT}\quad \frac{\Gamma \sim_U^\Delta \Gamma' \quad \Delta : t \Downarrow^N \Theta : w \quad (\Theta : w) \succsim_G (\Delta : w') \quad t \notin \mathit{Val}}{(\Gamma : t) \sim_U^\Delta (\Gamma' : w')}$$

And $(\Gamma : t)$ *is* update-related *to* $(\Gamma' : t')$*, written* $(\Gamma : t) \sim_U (\Gamma' : t')$*, if* $(\Gamma : t) \sim_U^\Gamma (\Gamma' : t')$.

4.3 Group-update relation

Finally, we combine the group and the update relations to obtain the desired equivalence between heaps in NS and those in NNS.

Definition 12. *Let* $\Gamma, \Gamma' \in LNHeap$. *We say that* Γ *is* group-update-related *to* Γ', *written* $\Gamma \succsim_{GU} \Gamma'$, *when*

$$\text{GUR} \quad \frac{\Gamma \succsim_G \Delta \quad \Delta \sim_U \Gamma' \quad \texttt{ok}\,\Gamma \quad \texttt{ok}\,\Gamma'}{\Gamma \succsim_{GU} \Gamma'}$$

And the extension to (heap : term) pairs:

$$\text{GUR\_HT} \quad \frac{(\Gamma : t) \succsim_G (\Delta : s) \quad (\Delta : s) \sim_U (\Gamma' : t') \quad \texttt{ok}\,\Gamma \quad \texttt{ok}\,\Gamma' \quad \texttt{lc}\,t \quad \texttt{lc}\,t'}{(\Gamma : t) \succsim_{GU} (\Gamma' : t')}$$

We define the equivalence between NS and NNS in similar fashion to the equivalence ANS-NNS (Theorem 1), that is, if a reduction proof can be obtained with NS for some term in a given context heap, then there must exist a reduction proof in NNS for that same (heap : term) pair such that the final (heap : value) is group-update-related to the final (heap : value) obtained with NS, and conversely.

Theorem 2. *(Equivalence NS-NNS).*

1. $\Gamma : t \Downarrow \Delta : w \Rightarrow \exists \Delta_N \in LNHeap\,.\,\exists w_N \in Val\,.\,\Gamma : t \Downarrow^N \Delta_N : w_N \wedge (\Delta_N : w_N) \succsim_{GU} (\Delta : w)$.
2. $\Gamma : t \Downarrow^N \Delta_N : w_N \Rightarrow \exists \Delta \in LNHeap\,.\,\exists w \in Val\,.\,\Gamma : t \Downarrow \Delta : w \wedge (\Delta_N : w_N) \succsim_{GU} (\Delta : w)$.

Likewise to Theorem 1, this result cannot be proved directly by rule induction and a generalization is needed. At present we are working on the proof of this generalization and other intermediate results. This may lead to slight modifications of the relations defined in this section.

Some of the problems that we have found in the proof of the generalization of the theorem are due to the fact that semantics rules for variables are different. Working with NS the variable that is being evaluated is removed from the heap, while it remains in the heap when applying NNS. In order to apply rule induction we have to remove this variable from the heap in NNS. However, several variables may depend on this one, and all of them must be removed from the heap in order not to lose the group relation between heaps.

5 Conclusions and Future Work

The variations introduced by Launchbury in its alternative natural semantics (ANS) do affect two rules: The variable rule (no update / no blackholes) and the application rule (indirections). We have defined two intermediate semantics to deal separately with the effects of each modification: NNS (without update / without blackholes) and INS (with indirections). Subsequently, we have studied the differences between the heaps obtained by the reduction systems corresponding to each semantics.

To begin with we have compared NNS with ANS, that is, substitution vs. indirections. To this purpose we have defined a preorder $\succsim_I$ expressing that a heap can be transformed into another by eliminating indirections. Furthermore, the relation $\succsim_I$ has been extended to (heap : terms) pairs, expressing that two terms can be considered equivalent when they have the same structure and their free variables (only those defined in the context of the corresponding heap) are the same except for some indirections. We have used this extended relation to establish the equivalence between the NNS and the ANS (Theorem 1).

Thereafter we have compared NS with NNS, that is, update vs. no update. The absence of update implies the duplication of evaluation work, that leads to the generation of duplicated bindings. These duplicated bindings come from the evaluation of `let`-declarations, so that they form *groups*. Therefore, we

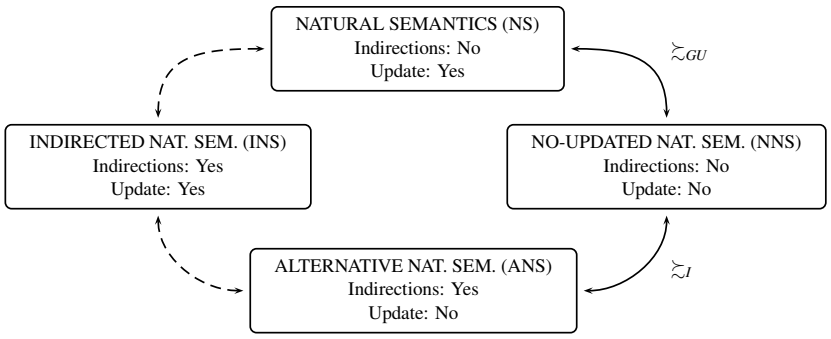


Figure 7: The relations between the semantics

have defined a *group-update*-relation $\succsim_{GU}$ that relates two heaps whenever the first can be transformed into the second by first eliminating duplicated groups of bindings, and then updating the bindings. We have extended $\succsim_{GU}$ for (heap : terms) to formulate an equivalence theorem for NS and NNS (Theorem 2). This closes the path from NS to ANS, and justifies their equivalence. A schema of the semantics and their relations is shown in Figure 7.

As we have mentioned before, we are still working on the proof of Theorem 2. When done we would like to complete the picture by comparing NS with INS, and then INS with ANS. For the first step, we have to define a preorder similar to $\succsim_{I}$, but taking into account that extra indirections may now be updated, thus leading to "redundant" bindings. For the second step, some variation of the group-update-relation will be needed. Dashed lines in Figure 7 indicate this future work.

We have chosen to use a locally nameless representation to avoid the problems with α-equivalence, and we have introduced cofinite quantification (in the style of [1]) in the evaluation rules that introduce fresh names, namely the rule for local declarations (LNLET) and for the alternative application (ALNAPP). Moreover, this representation is more amenable to formalization in proof assistants. In fact we have started to implement the semantic rules given in Section 2.2 using Coq [4], with the intention of obtaining a formal checking of our proofs.

Acknowledgements: This work is partially supported by the projects TIN2012-39391-C04-04 and S2009/TIC-1465.

References

[1] B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack & S. Weirich (2008): *Engineering formal metatheory*. In: *ACM Symposium on Principles of Programming Languages, POPL'08*, ACM Press, pp. 3–15.

[2] C. Baker-Finch, D. King & P. W. Trinder (2000): *An Operational Semantics for Parallel Lazy Evaluation*. In: *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, Montreal, Canada, pp. 162–173.

[3] H. P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*. *Studies in Logic and the Foundations of Mathematics* 103, North-Holland.

[4] Y. Bertot (2006): *Coq in a Hurry*. *CoRR* abs/cs/0603118. Available at `http://arxiv.org/abs/cs/0603118`.

[5] J. Breitner (2013): *The Correctness of Launchbury's Natural Semantics for Lazy Evaluation*. Archive of Formal Proofs. `http://afp.sf.net/entries/Launchbury.shtml`, Formal proof development, Amended version May 2014.

[6] N. G. de Bruijn (1972): *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. Indagationes Mathematicae 75(5), pp. 381–392.

[7] A. Charguéraud (2011): *The Locally Nameless Representation*. Journal of Automated Reasoning, pp. 1–46.

[8] M. van Eekelen & M. de Mol (2007): *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pp. 87–101. Radboud University Nijmegen.

[9] J. Launchbury (1993): *A Natural Semantics for Lazy Evaluation*. In: ACM Symp. on Principles of Programming Languages, POPL'93, ACM Press, pp. 144–154.

[10] K. Nakata & M. Hasegawa (2009): *Small-step and big-step semantics for call-by-need*. Journal of Functional Programming 19(6), pp. 699–722.

[11] T. Nipkow, L. C. Paulson & M. Wenzel (2002): *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283, Springer.

[12] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2010): *Call-by-need, call-by-name, and natural semantics*. Technical Report UU-CS-2010-020, Department of Information and Computing Sciences, Utrech University.

[13] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2010): *Trends in Functional Programming*, chapter An Operational Semantics for Distributed Lazy Evaluation, pp. 65–80. 10, Intellect.

[14] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2012): *A locally nameless representation for a natural semantics for lazy evaluation*. Technical Report 01/12, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid. Http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-1-12.pdf.

[15] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2012): *A Locally Nameless Representation for a Natural Semantics for Lazy Evaluation*. In: Theoretical Aspects of Computing ICTAC 2012, LNCS 7521, Springer, p. 105119.

[16] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2013): *The role of indirections in lazy natural semantics (extended version)*. Technical Report 13/13, Dpt. Sistemas Informáticos y Computación. Univ. Complutense de Madrid. Http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/TR-13-13.pdf.

[17] L. Sánchez-Gil, M. Hidalgo-Herrero & Y. Ortega-Mallén (2014): *The role of indirections in lazy natural semantics*. In: Proceedings of Ershov Informatics Conference (the PSI Conference Series, 9th edition). To appear.

[18] P. Sestoft (1997): *Deriving a lazy abstract machine*. Journal of Functional Programming 7(3), pp. 231–264.

[19] C. Urban, S. Berghofer & M. Norrish (2007): *Barendregt's Variable Convention in Rule Inductions*. In: Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction, LNCS 4603, Springer-Verlag, pp. 35–50.

[20] C. Urban & C. Kaliszyk (2012): *General Bindings and Alpha-Equivalence in Nominal Isabelle*. Logical Methods in Computer Science 8(2:14), pp. 1–35.