2nd Symposium on Languages, Applications and Technologies

SLATE'13, June 20-21, 2013, Porto, Portugal

Edited by

José Paulo Leal
Ricardo Rocha
Alberto Simões

OASICS

Editors

José Paulo Leal	Ricardo Rocha	Alberto Simões
CRACS & INESC TEC	CRACS & INESC TEC	CCTC & CEHUM
Faculdade de Ciências	Faculdade de Ciências	Instituto de Letras e Ciências Humanas
Universidade do Porto	Universidade do Porto	Universidade do Minho
`zp@fcc.fc.up.pt`	`ricroc@fc.up.pt`	`ambs@ilch.uminho.pt`

ACM Classification 1998
D.3 Programming Languages; D.2.12 Interoperability; I.2.7 Natural Language Processing;

ISBN 978-3-939897-52-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-52-1.

Publication date
June, 2013

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: OASIcs.SLATE.2013.i

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

Preface

The success of the humankind relies on our ability to communicate and transform the world. For ages we developed tools and technologies that allowed us to thrive and prosper. As we expanded to every corner of the planet we created languages that enabled us to communicate and record knowledge, even if they also become barriers to communication in themselves.

Technology and language have always been interconnected. Technologies to record language gave birth to history and the written language allowed us to preserve knowledge, including knowledge on technologies. Technology reshaped language as books, radio shows or motion pictures made us aware of how other people communicate. But technologies and language were not completely blend together until computers and networks become our favourite tool to communicate and transform the world.

The goal of the Symposium on Languages, Applications and Technologies (SLATE) is to be a forum to discuss the different ways in which language and technology interplay in computer science, and they are many. The symposium is divided into three main tracks, each one focusing a specific aspect of languages, from natural languages to compilers.

- The HHL (Human-Human Languages) track is dedicated to the discussion of research projects and ideas involving natural language processing and their industrial application.
- The HCL (Human-Computer Languages) track is where researchers, developers and educators exchange ideas and information on the latest academic or industrial work on language design, processing, assessment and applications.
- The CCL (Computer-Computer Languages) track main goal is to provide a broad space for discussion about the XML markup language, examples of usage and associated technologies.

SLATE follows the footsteps of two former conferences: CoRTA, the Conference on Compilers, Related Technologies and Applications; and XATA, the conference on XML, Applications and Applied Technologies, both with more than a decade of history.

This volume contains the proceedings of the 2nd edition of SLATE, held in the Department of Computer Science, Faculty of Sciences, University of Porto, Portugal, during June 20-21, 2013.

This year, SLATE received a total of 26 paper submissions for the three tracks. Each submission was reviewed by at least three Program Committee members, which included 55 researchers (counting sub-reviewers). At the end, 19 papers were selected for publication and presentation at the symposium, resulting in a 27% rejection rate. The set of accepted papers present a variety of contributions and were divided into the following five sessions for presentation at the symposium:

Software Development Tools, includes four articles on programming languages compilation and analysis;

XML and Applications, includes four articles on the usage of XML in different areas, ranging from the annotation of documents to its use on the semantic web;

Languages on Learning Environments, includes three articles that focus the automation on exercises generation and evaluation;

Domain Specific Languages, includes four articles on languages for specific languages, from music, robots or graphical user interfaces;

Natural Language Processing, includes four articles related to processing and teaching natural languages.

In addition to these sessions, the program also included two keynote presentations, one on the PICAT system, a scalable logic-based language, by Neng-Fa Zhou (Brooklyn College, New York), and another on software languages and their history, by Jean-Marie Favre (University of Grenoble, France).

The organizers of SLATE 2013 are in debt to many people without whom this event would never been possible. We wish to thank to our sponsors for making this event possible and to the EasyChair conference management system for simplifying our task. Thanks must go also to the authors of all submitted papers for their contribution and interest in the symposium and to the participants for making the event a meeting point for a fruitful exchange of ideas and feedback on recent developments. Finally, we want to express our gratitude to the Program Committee members and sub-reviewers, as the symposium would not have been possible without their dedicated time and knowledge in evaluating and ranking so many submissions from so many different topics.

To all, our deepest thanks!

José Paulo Leal
Ricardo Rocha
Alberto Simões

List of Authors

José João Almeida
Departamento de Informática
Universidade do Minho
Braga, Portugal
jj@di.uminho.pt

Bruno Azevedo
Departamento de Informática
Universidade do Minho
Braga, Portugal
azevedo.252@gmail.com

Michaela Bačíková
Department of Computers and Informatics
Technical University of Košice
Košice, Slovakia
michaela.bacikova@tuke.sk

Jorge Baptista
Universidade do Algarve
FCHS/CECL
Faro, Portugal
jbaptis@ualg.pt

Fernando Batista
Laboratório de Sistemas de Língua Falada
INESC-ID, and ISCTE,
Instituto Universitário de Lisboa, Portugal
fernando.batista@iscte.pt

Mario Berón
Department of Informatics
Universidad Nacional de San Luis
Ejército de los Andes, Argentina
mberon@unsl.edu.ar

José Campos
Lusíada University
Vila Nova de Famalicão
Portugal
jjscampos@eu.ipp.pt

João M. P. Cardoso
Faculty of Engineering
University of Porto, Portugal
jmpc@acm.org

Sergej Chodarev
Department of Computers and Informatics
Technical University of Košice
Košice, Slovakia
sergej.chodarev@tuke.sk

Teresa Costa
CRACS & INESC-Porto LA
Faculty of Sciences
University of Porto, Portugal
up200101764@alunos.dcc.fc.up.pt

Daniela da Cruz
Departamento de Informática
Universidade do Minho
Braga, Portugal
danieladacruz@di.uminho.pt

Pedro C. Diniz
INESC-ID, Lisbon, Portugal
pedro@esda.inesc-id.pt

Jean-Marie Favre
Université Joseph Fourier
Grenoble, France
jean-marie.favre@megaplanet.org

Diogo R. Ferreira
Instituto Superior Técnico
Universidade Técnica de Lisboa
Lisboa, Portugal
diogo.ferreira@ist.utl.pt

Daniela Fonte
Departamento de Informática
Universidade do Minho
Braga, Portugal
danielamoraisfonte@gmail.com

Georgios Fourtounis
School of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
gfour@softlab.ntua.gr

Tiago Freitas
IST – Instituto Superior Técnico
L^{2}F – Spoken Language Systems Laboratory
INESC ID, Lisboa, Portugal
tiago.freitas@ist.utl.pl

Alda Lopes Gançarski
Institute Telecom
Telecom SudParis
Paris, France
alda.gancarski@telecom-sudparis.eu

Xavier Gómez Guinovart
TALG Group
Universidade de Vigo
Galiza, Spain
xgg@uvigo.es

Ivan Halupka
Department of Computers and Informatics
Technical University of Košice
Košice, Slovakia
ivan.halupka@tuke.sk

Pedro Rangel Henriques
Departamento de Informática
Universidade do Minho
Braga, Portugal
prh@di.uminho.pt

Ján Kollár
Department of Computers and Informatics
Technical University of Košice
Košice, Slovakia
jan.kollar@tuke.sk

Dominik Lakatoš
Department of Computers and Informatics
Technical University of Košice
Košice, Slovakia
dominik.lakatos@tuke.sk

László János Laki
MTA-PPKE Lang. Techn. Research Group
Pázmány Péter Catholic University
Faculty of Information Technology, Hungary
laki.laszlo@itk.ppke.hu

José Paulo Leal
CRACS & INESC-Porto LA
Faculty of Sciences
University of Porto, Portugal
zp@dcc.fc.up.pt

Nuno Mamede
IST – Instituto Superior Técnico
L^{2}F – Spoken Language Systems Laboratory
INESC ID, Lisboa, Portugal
nuno.mamede@ist.utl.pl

Henrique Medeiros
Laboratório de Sistemas de Língua Falada
INESC-ID, and ISCTE
Instituto Universitário de Lisboa, Portugal
hrbmedeiros@hotmail.com

Enrique Miranda
Department of Informatics
Universidad Nacional de San Luis
Ejército de los Andes, Argentina
eamiranda@unsl.edu.ar

Helena Moniz
Laboratório de Sistemas de Língua Falada
INESC-ID, and FLUL/CLUL
Universidade de Lisboa, Portugal
helena.moniz@inesc-id.pt

German Montejano
Department of Informatics
Universidad Nacional de San Luis
Ejército de los Andes, Argentina
gmonte@unsl.edu.ar

Milan Nosáľ
Department of Computers and Informatics,
Technical University of Košice
Košice, Slovakia
milan.nosal@tuke.sk

Attila Novák
MTA-PPKE Lang. Techn. Research Group
Pázmány Péter Catholic University
Faculty of Information Technology, Hungary
novak.attila@itk.ppke.hu

Luis Nunes
Instituto de Telecomunicações, and
ISCTE - Instituto Universitário de Lisboa
Lisboa, Portugal
luis.nunes@iscte.pt

Nuno Oliveira
Departamento de Informática
Universidade do Minho
Braga, Portugal
nunooliveira@di.uminho.pt

György Orosz
MTA-PPKE Lang. Techn. Research Group
Pázmány Péter Catholic University
Faculty of Information Technology, Hungary
oroszgy@itk.ppke.hu

Nikolaos S. Papaspyrou
School of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
nickie@softlab.ntua.gr

Maria João Varanda Pereira
Polytechnic Institute of Bragança
Bragança, Portugal
mjp@ipb.pt

Emília Pietriková
Department of Computers and Informatics
Technical University of Košice
Košice, Slovakia
emilia.pietrikova@tuke.sk

Jaroslav Porubän
Department of Computers and Informatics
Technical University of Košice
Košice, Slovakia
jaroslav.poruban@tuke.sk

Ricardo Queirós
CRACS & INESC-Porto LA, and
DI-ESEIG/IPP
Porto, Portugal
ricardo.queiros@eu.ipp.pt

Ricardo Rocha
CRACS & INESC TEC, and
Faculty of Sciences, University of Porto
Porto, Portugal
ricroc@dcc.fc.up.pt

André C. Santos
INESC-ID, and
IST, Technical University of Lisbon,
Lisbon, Portugal
acoelhosantos@ist.utl.pt

João Santos
CRACS & INESC TEC, and
Faculty of Sciences
University of Porto, Portugal
jsantos@dcc.fc.up.pt

Borbála Siklósi
Pázmány Péter Catholic University
Faculty of Information Technology
Budapest, Hungary
siklosi.borbala@itk.ppke.hu

Alberto Simões
Centro de Estudos Humanísticos
Universidade do Minho
Campus de Gualtar, Braga, Portugal
ambs@ilch.uminho.pt

Isabel Trancoso
Laboratório de Sistemas de Língua Falada
INESC-ID, and Instituto Superior Técnico
Lisboa, Portugal
isabel.trancoso@inesc-id.pt

Neng-Fa Zhou
Brooklyn College
The City University of New York
United States of America
zhou@sci.brooklyn.cuny.edu

Committees

Program Chairs

José Paulo Leal
Universidade do Porto, Portugal

Ricardo Rocha
Universidade do Porto, Portugal

Alberto Simões
Universidade do Minho, Portugal

Publication Chair

Alberto Simões
Universidade do Minho, Portugal

Program Committee

Salvador Abreu
Universidade de Évora, Portugal

Ademar Aguiar
Universidade do Porto, Portugal

José João Almeida
Universidade do Minho, Portugal

Jorge Baptista
Universidade do Algarve, Portugal

María Inés Torres Barañano
Universidad del País Vasco, Spain

Fernando Batista
ISCTE-IUL & INESC-ID, Portugal

Mario Berón
Universidad Nacional de San Luis, Argentina

João Paiva Cardoso
Universidade do Porto, Portugal

Nuno Ramos Carvalho
Universidade do Minho, Portugal

Bastian Cramer
Universität Paderborn, Germany

Matej Crepinsek
Univerza v Mariboru, Slovenia

Daniela da Cruz
Universidade do Minho, Portugal

Gabriel David
Universidade do Porto & INESC TEC, Portugal

Ricardo Dias
Universidade Nova de Lisboa, Portugal

Brett Drury
Universidade do Porto, Portugal

Jean-Marie Favre
Université Joseph Fourier, Grenoble, France

Luis Ferreira
Instituto Politécnico do Cávado e Ave, Portugal

Miguel Ferreira
Universidade do Minho, Portugal

Jean-Cristophe Filliâtre
CNRS & Université Paris Sud, France

Mikel Forcada
Universitat d'Alacant, Spain

Pablo Gamallo
Universidade de Santiago de Compostela, Spain

Alda Lopes Gançarski
Institut Mines-Télécom/Télécom SudParis, France

Marcos Garcia
Universidade de Santiago de Compostela, Spain

Xavier Gómez Guinovart
Universidade de Vigo, Spain

Pedro Rangel Henriques
Universidade do Minho, Portugal

David Insa
Universitat Politècnica de València, Spain

Mirjana Ivanovic
University of Novi Sad, Serbia

Tomaz Kosar
Univerza v Mariboru, Slovenia

José Paulo Leal
Universidade do Porto, Portugal

António Menezes Leitão
Universidade Técnica de Lisboa, Portugal

Giovani Librelotto
Universidade Federal Santa Maria, Brazil

João Correia Lopes
Universidade do Porto & INESC TEC, Portugal

João Lourenço
Universidade Nova de Lisboa, Portugal

Ivan Lukovic
University of Novi Sad, Serbia

Claude Marché
Inria & Université Paris-Sud, France

Marjan Mernik
Univerza v Mariboru, Slovenia

Hugo Gonçalo Oliveira
Universidade de Coimbra, Portugal

Nuno Oliveira
Universidade do Minho, Portugal

Alexander Paar
TWT GmbH Science and Innovation, Germany

Lluís Padró
Universitat Politècnica de Catalunya, Spain

Maria João Varanda Pereira
Instituto Politécnico de Bragança, Portugal

Alberto Proença
Universidade do Minho, Portugal

Ricardo Queirós
Instituto Politécnico do Porto, Portugal

José Carlos Ramalho
Universidade do Minho, Portugal

Cristina Ribeiro
Universidade do Porto & INESC TEC, Portugal

Ricardo Ribeiro
ISCTE-IUL & INESC-ID, Portugal

Ricardo Rocha
Universidade do Porto, Portugal

Casiano Rodriguez-Leon
Universidad de La Laguna, Spain

Josep Silva
Universitat Politècnica de València, Spain

Alberto Simões
Universidade do Minho, Portugal

Boštjan Slivnik
Univerza v Ljubljani, Slovenia

Simão Melo de Sousa
Universidade da Beira Interior, Portugal

António Teixeira
Universidade de Aveiro, Portugal

Jörg Tiedemann
Uppsala University, Sweeden

Pedro Vasconcelos
Universidade do Porto, Portugal

Organization Committee

Miguel Areias
Universidade do Porto, Portugal

Nuno Ramos Carvalho
Universidade do Minho, Portugal

José Paulo Leal
Universidade do Porto, Portugal

Ricardo Queirós
Instituto Politécnico do Porto, Portugal

Ricardo Rocha
Universidade do Porto, Portugal

João Santos
Universidade do Porto, Portugal

Alberto Simões
Universidade do Minho, Portugal

Part I

Keynotes

2^{nd} Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Software Languages: The Lingusitic Continuum (Invited talk)

Jean-Marie Favre

University of Grenoble, SIGMA-LIG, France
`jean-marie.favre@megaplanet.org`

Abstract

While computers are linguistic machines moving symbols around, Informatics is BY and FOR people. I claim here that the gap between Computer Languages and Human Languages is, as a matter of fact, filled by a wide spectrum of Software Languages. My point is that the notion of Software Language goes far beyond Programming Languages; just like Informatics is indeed much more than Computer Science. After a very brief retrospective on the history of languages and Information Technologies, I show that nowadays nearly all kinds of languages are indeed amenable to be represented as software; at least to some certain extent. Software Languages include not only the languages used typically in Software Engineering (e.g. Modeling Languages, Specification Languages, Architecture Description Languages, Query Languages, and so on), but also all kinds of Domain Specific Languages that originate from all other areas of human activities. As a matter of fact, although Scientific Languages, Engineering Languages and Business Languages existed long before Computers we all witness today the progressive transformation of these languages into their counterpart as Software Languages. Software Languages can take many different incarnations such as grammars, ontologies, schemas or metamodels. Moreover, these descriptions are often missing as many languages remain "implicit" or just exist in the form of proto-languages. I do not claim here that the notion of "Software Language" is clear cut or well understood. I just advocate that since these languages could reveal to be fundamental in the context of the Information Age they should be (1) studied from a scientific point of view in an integrative approach, and (2) developed and evolved in principled ways. This leads the emerging fields of Software Linguistics and Software Language Engineering respectively.

1998 ACM Subject Classification D.3.2 Language Classifications, F.4.3 Formal Languages, H3.2 Languages, K.2 History of Computing

Keywords and phrases Software Languages, Software Linguistics, Software Language Engineering

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.3

Picat: A Scalable Logic-based Language and System (Invited talk)

Neng-Fa Zhou

Brooklyn College, The City University of New York
2900 Bedford Avenue, Brooklyn, New York, USA
`zhou@sci.brooklyn.cuny.edu`

Abstract

This talk will give the design principles of the Picat language (`http://www.picat-lang.org`), highlight the high-level and intuitive abstractions provided by Picat for easy programming, and contemplate why Picat is more robust and scalable than Prolog and could be more accessible than Prolog to ordinary programmers for scripting and modeling tasks.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Functions, Relations, Loops, Constraints, Tabling

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.5

Despite the elegant concepts, new extensions (e.g., tabling and constraints), and successful applications (e.g., knowledge engineering, NLP, and search problems), Prolog has a bad reputation for being old and difficult. Many ordinary programmers find the implicit non-directionality and non-determinism of Prolog to be hard to follow, and the non-logical features, such as cuts and dynamic predicates, are prone to misuses, leading to absurd codes. The lack of language constructs (e.g., loops) and libraries for programming everyday things is also considered a big weakness of Prolog. The backward compatibility requirement has made it hopeless to remedy the language issues in current Prolog systems, and there are urgent calls for a new language.

Several successors of Prolog have been designed, including Mercury, Erlang, Oz, and Curry. The requirement of many kinds of declarations in Mercury has made the language difficult to use; Erlang's abandonment of non-determinism in favor of concurrency has made the language unsuited for many applications despite its success in the telecom industry; Oz has never attained the popularity that the designers sought, probably due to its unfamiliar syntax and implicit laziness; Curry is considered too close to Haskell. All of these successors were designed in the 1990s, and now the time is ripe for a new logic-based language.

Picat aims to be a simple, and yet powerful, logic-based programming language for a variety of applications. Picat incorporates many declarative language features for better productivity of software development, including explicit non-determinism, explicit unification, functions, constraints, and tabling. Picat lacks Prolog's non-logical features, such as the cut operator and dynamic predicates, making Picat more reliable than Prolog. Picat also provides imperative language constructs for programming everyday things. The resulting system will be used for not only symbolic computations, which is a traditional application domain of declarative languages, but also for scripting and modeling tasks.

Picat is a general-purpose language that incorporates features from logic programming, functional programming, and scripting languages. The letters in the name summarize Picat's features:

- **P**attern-matching: A *predicate* defines a relation, and can have zero, one, or multiple answers. A *function* is a special kind of a predicate that always succeeds with *one* answer. Picat is a rule-based language. Predicates and functions are defined with pattern-matching rules.
- **I**mperative: Picat provides assignment and loop statements for programming everyday things. An assignable variable mimics multiple logic variables, each of which holds a value at a different stage of computation. Assignments are useful for computing aggregates and are used with the `foreach` loop for implementing list comprehensions.
- **C**onstraints: Picat supports constraint programming. Given a set of variables, each of which has a domain of possible values, and a set of constraints that limit the acceptable set of assignments of values to variables, the goal is to find an assignment of values to the variables that satisfies all of the constraints.
- **A**ctors: Actors are event-driven calls. Picat provides *action rules* for describing event-driven behaviors of actors. Events are posted through channels. An actor can be attached to a channel in order to watch and to process its events. Picat treats threads as channels, and allows the use of action rules to program concurrent threads.
- **T**abling: Tabling can be used to store the results of certain calculations in memory, allowing the program to do a quick table lookup instead of repeatedly calculating a value. As computer memory grows, tabling is becoming increasingly important for offering dynamic programming solutions for many problems.

Picat is more expressive than Prolog for scripting and modeling. With arrays, loops, and list comprehensions, it is not rare to find problems for which Picat requires an order of magnitude fewer lines of code to describe than Prolog. Picat is more scalable than Prolog. The use of pattern-matching rather than unification facilitates indexing of rules. Picat is more reliable than Prolog. In addition to explicit non-determinism, explicit unification, and a simple static module system, the lack of cuts, dynamic predicates, and operator overloading also improve the reliability of the language. Picat is not as powerful as Prolog for metaprogramming and it's impossible to write a meta-interpreter for Picat in Picat itself. Nevertheless, this weakness can be remedied with library modules for implementing domain-specific languages.

Part II

Software Development Tools

2^{nd} Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Or-Parallel Prolog Execution on Clusters of Multicores

João Santos and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
`{jsantos,ricroc}@dcc.fc.up.pt`

Abstract

Logic Programming languages, such as Prolog, provide an excellent framework for the parallel execution of logic programs. In particular, the inherent non-determinism in the way logic programs are structured makes Prolog very attractive for the exploitation of *implicit parallelism*. One of the most noticeable sources of implicit parallelism in Prolog programs is *or-parallelism*. Or-parallelism arises from the simultaneous evaluation of a subgoal call against the clauses that match that call. Arguably, the most successful model for or-parallelism is *environment copying*, that has been efficiently used in the implementation of or-parallel Prolog systems both on shared memory and distributed memory architectures. Nowadays, multicores and clusters of multicores are becoming the norm and, although, many parallel Prolog systems have been developed in the past, to the best of our knowledge, none of them was specially designed to explore the combination of shared with distributed memory architectures. Motivated by our past experience, in designing and developing parallel Prolog systems based on environment copying, we propose a novel computational model to efficiently exploit implicit parallelism from large scale real-world applications specialized for the novel architectures based on clusters of multicores.

1998 ACM Subject Classification D.1.3 Concurrent Programming

Keywords and phrases Logic Programming, Or-Parallelism, Environment Copying, Scheduling

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.9

1 Introduction

Logic Programming languages, such as Prolog, provide a high-level, declarative approach to programming. In general, logic programs can be seen as executable specifications that despite their simple declarative and procedural semantics allow for designing very complex and efficient applications. The inherent non-determinism in the way logic programs are structured as simple collections of alternative logic clauses makes Prolog very attractive for the exploitation of *implicit parallelism*.

Prolog offers two major forms of implicit parallelism: *and-parallelism* and *or-parallelism* [5]. And-Parallelism stems from the parallel evaluation of subgoals in a clause, while or-parallelism results from the parallel evaluation of a subgoal call against the clauses that match that call. Arguably, or-parallel systems, such as Aurora [7] and Muse [3], have been the most successful parallel logic programming systems so far. Intuitively, the least complexity of or-parallelism makes it more attractive as a first step. However, practice has shown that a main difficulty, when implementing or-parallelism, is how to efficiently represent the *multiple bindings* for the same variable produced by the parallel execution of alternative matching clauses. One of the most successful or-parallel models that solves the multiple bindings problem is *environment copying*, that has been efficiently used in the implementation of or-parallel Prolog systems both on shared memory [3, 10] and distributed memory [16, 9] architectures.

Another major difficulty in the implementation of any parallel system is the design of *scheduling strategies* to efficiently assign computing tasks to idle workers. A parallel Prolog system is no exception as the parallelism that Prolog programs exhibit is usually highly irregular. Achieving the necessary cooperation, synchronization and concurrent access to shared data among several workers during execution is a difficult task. For environment copying, scheduling strategies based on *dynamic scheduling* of work have proved to be very efficient [2]. *Stack splitting* [4, 8] is an alternative scheduling strategy for environment copying that provides a simple and clean method to accomplish work splitting among workers in which all available work is *statically divided beforehand* in complementary sets between the sharing workers. Due to its static nature, stack splitting was thus first introduced aiming at distributed memory architectures [16, 9] but, recent work, also showed good results for shared memory architectures [15, 14].

The increasing availability and popularity of multicore processors have made our personal computers parallel with multiple cores sharing the main memory. Multicores and clusters of multicores are now the norm and, although, many parallel Prolog systems have been developed in the past, most of them are no longer available, maintained or supported. Moreover, to the best of our knowledge, none of them was specially designed to explore the combination of shared with distributed memory architectures. On one hand, the shared memory based models take advantage of synchronization mechanisms that cannot be easily extended to distributed environments and, on the other hand, the distributed memory based models use specialized communication mechanisms that do not take advantage of the fact that some workers can be sharing memory resources.

Motivated by the intrinsic and strong potential that Prolog has for implicit parallelism and by our past experience in designing and developing parallel systems based on environment copying [10, 9, 15, 14], we propose a novel computational model to efficiently exploit parallelism from large scale real-world applications specialized for clusters of low cost multicore architectures. In this new model, we will have two levels of computational units, *single workers* and *teams of workers*, and the ability to exploit different scheduling strategies, for distributing work among teams and among the workers inside a team. Our approach resembles the concept of teams used by some of the models combining and-parallelism with or-parallelism, like the Andorra-I [13] or ACE [6] systems, where a layered approach implements different schedulers to deal with each level of parallelism.

In our model, a team of workers is formed by workers sharing the same memory address space, i.e., two workers executing in different computer nodes cannot belong to the same team, but we can have more than a team executing in the same computer node. For (shared memory) multicores, we can thus have any combination of strategies, teams and workers inside a team can distribute work using both dynamic or static scheduling of work. For (distributed memory) clusters of multicores, we can only have (static) stack splitting for distributing work among teams, but we can still have dynamic or static scheduling of work for distributing work among the workers inside a team. This idea is similar to the MPI/OpenMP hybrid programming pattern, where MPI is usually used to communicate work among workers in different computer nodes and OpenMP is used to communicate work among workers in the same node.

The remainder of the paper is organized as follows. First, we introduce some background about environment copying, stack splitting and work scheduling. Next, we introduce our new model and discuss the major design issues, algorithms and challenges. Last, we advance directions for further work. Throughout the text, we assume the reader will have good familiarity with the general principles of Prolog implementation, and namely with the WAM [18, 1]. When discussing some technical details, we will take as reference the state-of-the-art Yap Prolog system [12], that integrates or-parallelism based on the environment copying model and supports both dynamic and static scheduling of work.

2 Environment Copying

In the environment copying model, each worker keeps a separate copy of its own environment, thus the bindings to shared variables are done as usual (i.e., stored in the private execution stacks of the worker doing the binding) and without conflicts. Every time a worker shares work with another worker, all the execution stacks are copied to ensure that the requesting worker has the same environment state down to the search tree node where the sharing occurs. At the engine level, a search tree node corresponds to a choice point in the local stack [18, 1].

As a result of environment copying, each worker can proceed with the execution exactly as a sequential engine, with just minimal synchronization with other workers. Synchronization is mostly needed when updating scheduling data and when accessing shared nodes in order to ensure that unexplored alternatives are only exploited by one worker. All other WAM data structures, such as the environment frames, the heap, and the trail do not require synchronization.

2.1 Incremental Copying

To reduce the overhead of stack copying, an optimized copy mechanism called *incremental copy* [3] takes advantage of the fact that the requesting worker may already have traversed part of the path being shared. Therefore, it does not need to copy the stacks referring to the whole path from root, but only the stacks starting from the youngest node common to both workers.

For example, consider that worker Q asks worker P for sharing and that worker P decides to share its private nodes with Q. To implement incremental copying, Q should start by backtracking to the youngest common node with P, therefore becoming partially consistent with part of P. Then, if Q receives a positive answer from P, it only needs to copy the differences between P and Q. These differences can be easily calculated through the information stored in the common node found by Q and in the top registers of the execution stacks of P. Care must be taken about variables older than the youngest common node that were instantiated by P, as incremental copying does not copy these bindings. Worker Q thus needs to explicitly *install* the bindings for such variables. This process, called the *adjustment of cells outside the increments*, is implemented by searching the trail stack for bindings to variables older than the youngest common node [3].

2.2 Or-Frames

Deciding which workers to ask for work and how much work should be shared is a function of the scheduler. A fundamental task when sharing work is to turn *public* the private choice points, so that backtracking to these choice points can be synchronized between different workers. Public choice points are treated differently because we need to synchronize workers in such a way that we avoid executing twice the same alternative.

Strategies based on dynamic scheduling of work, use *or-frames* to implement such synchronization [3]. A worker sharing work adds an or-frame data structure to each private choice point made public. Each or-frame stores the pointer to the next available alternative, as previously stored in the corresponding private choice point, and supports a mutual exclusion mechanism that guarantees atomic updates to the or-frame data. Shared nodes thus become represented by or-frames, a data structure that workers must access, with mutual exclusion, to obtain the unexplored alternatives. The set of all or-frames form a tree that represents the public search tree.

2.3 Stack Splitting

Stack splitting was first introduced to target distributed memory architectures, thus aiming to reduce the mutual exclusion requirements of the or-frames when accessing shared nodes of the search tree. It accomplishes this by defining simple and clean work splitting strategies in which all available work is statically divided beforehand in two complementary sets between the sharing workers. In practice, with stack splitting the synchronization requirement is removed by the preemptive split of all unexplored alternatives at the moment of sharing. The splitting is such that both workers will proceed, each executing its branch of the computation, without any need for further synchronization when accessing shared nodes.

The original stack splitting proposal [4] introduced two strategies for dividing work: *vertical splitting*, in which the available choice points are alternately divided between the two sharing workers, and *horizontal splitting*, which alternately divides the unexplored alternatives in each available choice point. *Diagonal splitting* [9] is a more elaborated strategy that achieves a precise partitioning of the set of unexplored alternatives. It is a kind of mix between horizontal and vertical splitting, where the set of all unexplored alternatives in the available choice points is alternately divided between the two sharing workers. Another splitting strategy [17], which we named *half splitting*, splits the available choice points in two halves. Figure 1 illustrates the effect of these strategies in a work sharing operation between a busy worker P and an idle worker Q.

Figure 1(a) shows the initial configuration with the idle worker Q requesting work from a busy worker P with 7 unexplored alternatives in 4 choice points. Figure 1(b) shows the effect of vertical splitting, in which P keeps its current choice point and alternately divides with Q the remaining choice points up to the root choice point. Figure 1(c) illustrates the effect of half splitting, where the bottom half is for worker P and the half closest to the root is for worker Q. Figure 1(d) details the effect of horizontal splitting, in which the unexplored alternatives in each choice point are alternately split between both workers, with workers P and Q owning the first unexplored alternative in the even and odd choice points, respectively. Figure 1(e) describes the diagonal splitting strategy, where the unexplored alternatives in all choice points are alternately split between both workers in such a way that, in the worst case, Q may stay with one more alternative than P. For all strategies, the corresponding execution stacks are first copied to Q, next both P and Q perform splitting, according to the splitting strategy at hand, and then P and Q are set to continue execution.

2.4 The Yap Prolog System

The Yap Prolog system implements or-parallelism based on the environment copying model and supports both dynamic and static scheduling of work. To implement dynamic scheduling, Yap follows the original Muse approach which uses or-frames to synchronize the access to the open alternatives. To implement static scheduling, two different approaches were

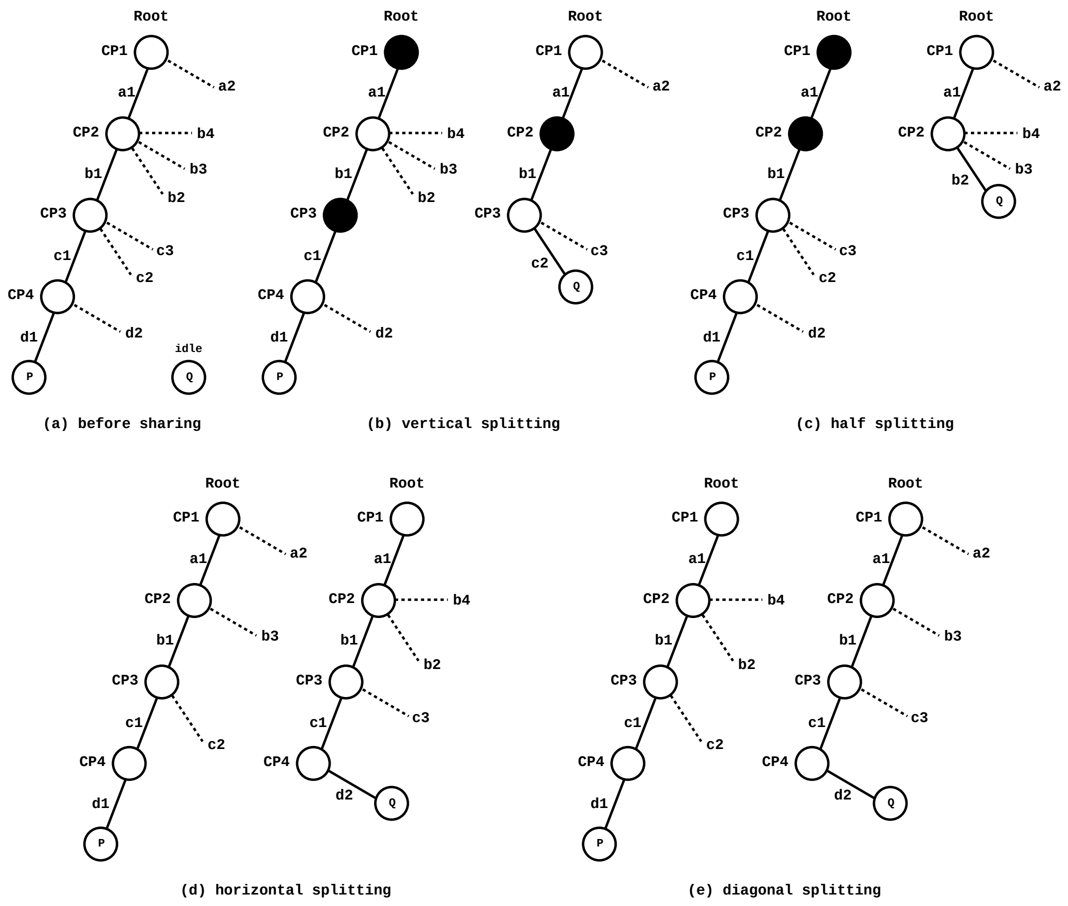


Figure 1 Alternative stack splitting strategies.

followed. In the first approach, the engine was designed to run in Beowulf clusters [9]. More recently, a second approach was designed to run in multicores and it has shown to be very competitive when compared with the original or-frames approach [15, 14].

When running in shared memory architectures, Yap's workers can be either processes (the engine using processes is called YapOr [10]) or POSIX threads (the engine using threads is called ThOr [11]). The memory organization for YapOr/ThOr is quite similar for all the approaches (see Fig. 2(a)). The memory of the system is divided into two major address spaces: the *global space* and a collection of *local spaces*. The global space contains the code area inherited from Yap and all data structures necessary to support parallelism. Among these structures is static information about the execution, such as the number of workers, and dynamic information responsible for determining the end of the execution. Each local space represents one worker and contains the execution stacks inherited form Yap (heap, local, trail and auxiliary stack) and information related to the execution of that worker such as the top shared choice point, share and prune requests or the load of that worker [10, 11].

When running in distributed memory architectures, Yap's workers are processes, each with independent global and local spaces (see Fig. 2(b)). Despite not specially designed for it, this approach also fits in shared memory architectures, i.e., we can have some workers running on the same computer node, but as fully independent processes.

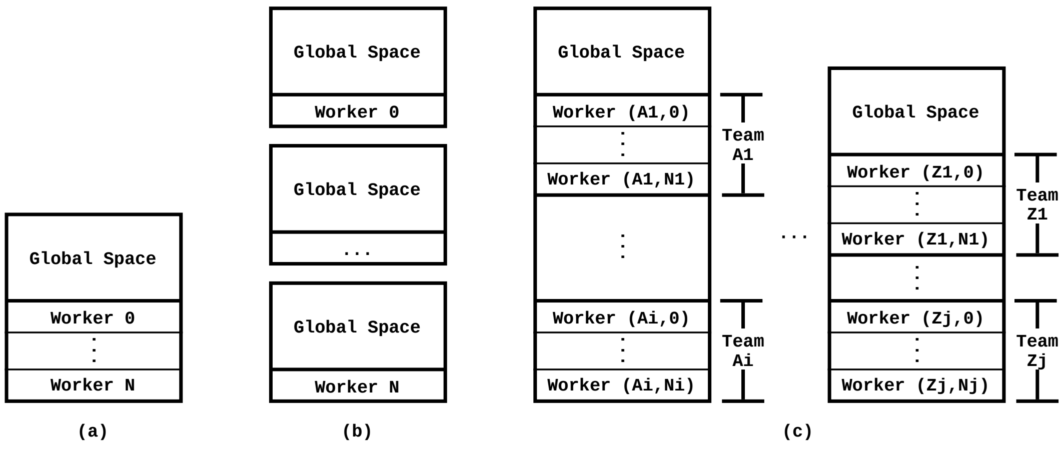


Figure 2 Memory layout for: (a) workers in shared memory; (b) workers in distributed memory; and (c) teams of workers in clusters of multicores.

3 Our Proposal

The goal behind our proposal is to implement the concept of teams trying to reuse, as much as possible, Yap's existing infrastructure. We define a team as a set of workers (processes or threads) who share the same memory address space and cooperate to solve a certain part of the main problem. By demanding that all workers inside a team share the same address space implies that all workers should be in the same computer node. On the other hand, we also want to be possible to have several teams in a computer node or distributed by other nodes.

3.1 Memory Organization

In order to support teams, there are several changes that need to be made, being one of the first, the memory organization. Figure 2(c) shows the new memory layout to support teams of workers. Each team of workers mimics the previous memory layout for a set of workers in shared memory (see Fig. 2(a)), where the memory of the system is divided into a global space, shared among all workers, and a collection of local spaces, each representing one worker's team. In this new memory layout, we can also have several teams sharing the same memory address space and, in particular, sharing the global space. To accomplish that, the information stored in the global space is now related with teams instead of being related with single workers. Moreover, the global space now includes an extra area, named *team space*, where each team stores static information about the team and dynamic information about the execution of the team, such as, to determine if the team is out of work or if it has finished execution. The collection of local spaces maintains its functionality, i.e., it stores the execution stacks and information about the state of the corresponding worker.

Since our aim is to target clusters of multicores, the complete layout for the new memory organization can be seem as a generalization of the previous approach for distributed memory architectures (see Fig. 2(b)), but now instead of single workers with independent global and local spaces, we may have teams, individual teams or collection of teams as described above, sharing the same memory address space.

3.2 Mixed Scheduling

One of the main advantages of using teams is that we can combine the scheduling strategies mentioned before. Therefore we may have teams using static scheduling while others, at the same time, use dynamic scheduling. Figure 3 shows a schematic representation of what we want to achieve with our proposal. In this example, we have a cluster composed by two computers nodes, $N1$ and $N2$. The computer node $N1$ has two teams, team A and team B with 4 workers each. The computer node $N2$ has only one team, team C with 8 workers.

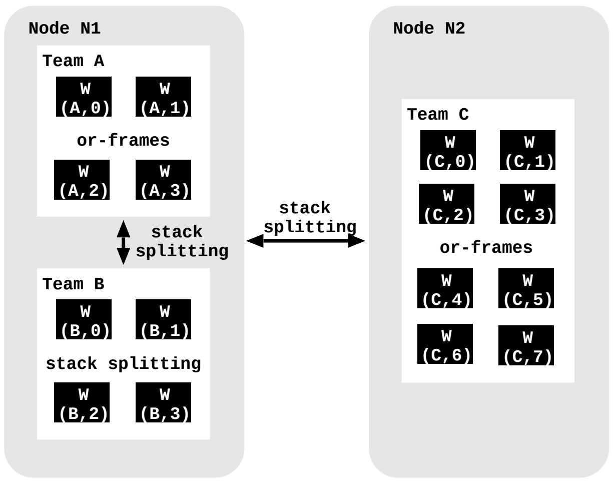


Figure 3 Work scheduling within and among teams.

Regarding the scheduling strategy adopted to distribute work inside the teams, teams A and C are using dynamic scheduling with or-frames, while team B is using stack splitting. To distribute work among teams, we only use stack splitting. This is mandatory since we want to have a single scheduling protocol to distribute work between teams (being they in the same or in different computer nodes) and we want to fully avoid having synchronization data structures, such as the or-frames, being shared between teams. Note that having the access to the open alternatives in data structures shared between teams, not only would have a great impact in the communication overhead required to keep them up-to-date, but would also not clarify the notion of being a team. If two teams are synchronizing the access to the open alternatives, in fact they are not two different teams but only one, because no decision regarding the shared open alternatives can be made without involving both teams.

Independently of the scheduling strategy, teams will have to communicate among them when sharing work or when sending requests to perform a cut or to ensure the termination of the computation. To implement the communication layer, we can use a message passing protocol, for teams physically located in the same or in different computer nodes, or a shared memory synchronization mechanism, for teams in the same computer node. Note that, in this latter case, synchronization is being use to implement communication and not for scheduling purposes, as discussed before.

3.3 Work Sharing

To distribute work inside a team, we can use, with minor adaptations, any of Yap's current dynamic or static schedulers for shared memory. Since these schedulers were developed to deal with workers that are sharing the same memory address space, they can thus be easily extended to support work sharing inside a team. As discussed before, this is not the case for work sharing among teams. To deal with that, our approach is thus to implement a layered approach, similar to the one used by some of the models combining and-parallelism with or-parallelism [13, 6], and for that a second-level scheduler will be used.

Since the concept of a team implies that we must give priority to the exploitation of the work available inside the team, we will only ask for work to other teams when no more work exists in a team. However, even though that it is the entire team that is out of work, the sharing process will still be done between two workers, being the selected worker of the idle team then the responsible for sharing the new work with its teammates.

Figure 4 shows a schematic representation of the sharing process between teams. Consider the cluster configuration in Fig. 3 and assume that team C has run out of work and that team A was selected by C's scheduler to share work with it. Figure 4(a) shows the state of team A before the sharing request from C. The four workers in team A are executing in the private region of the search tree and all share the top three choice points. The top shared choice point is already dead, i.e., without open alternatives, but the second and third shared choice points have two ($b2$ and $b3$) and one ($c3$) open alternatives, respectively.

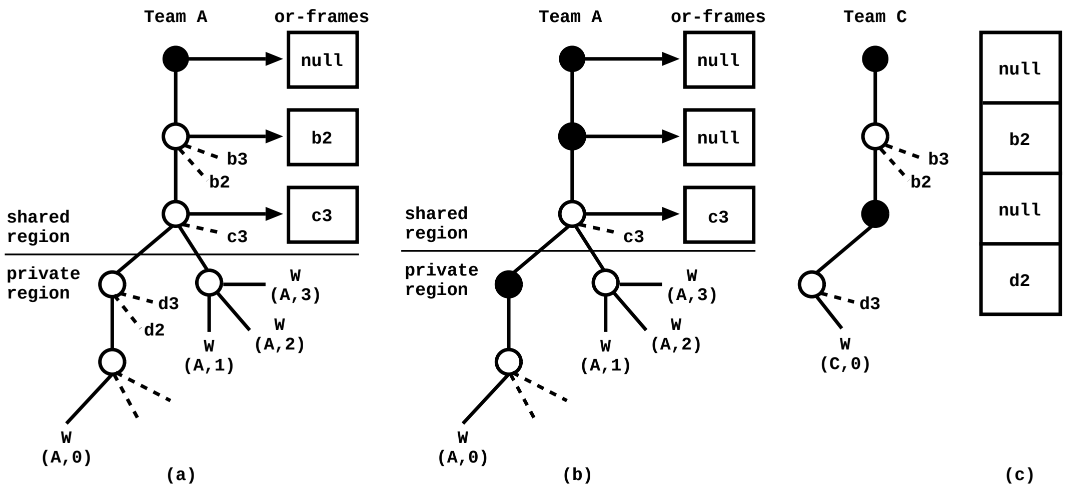


Figure 4 Schematic representation of the sharing process between workers of different teams: in (a) we can see the configuration of team A when team C asks for work and in (b) we can see the configuration of both teams after the sharing process, considering that worker $W(A,0)$ used vertical splitting to share its available work (in (c) we can see the array of open alternatives being shared) with worker $W(C,0)$.

When team A receives the sharing request from team C, one of the workers from A will be selected to share part of its available (private and/or shared) work and manage the sharing process with the requesting worker from C. For the sake of simplicity, here we are considering that this is done by the workers 0 of each team, workers $W(A,0)$ and $W(C,0)$. Since this is a sharing operation between teams, static scheduling is then the strategy adopted to split work. In particular, in this example, we are using the vertical splitting strategy.

To implement vertical splitting, $W(A,0)$ thus needs to alternately divide its choice points with $W(C,0)$. However, since team A is using or-frames to implement dynamic scheduling of work inside the team, we cannot apply the original stack splitting algorithm [15, 14] to split the available work in the shared region of the search tree (please remember that stack splitting avoids the use of or-frames). To solve that problem, $W(A,0)$ constructs an array with the open alternatives per choice point that it will hand over to $W(C,0)$. This array is illustrated in Fig. 4(c). The motivation for using this array is the isolation between the alternatives being shared and the scheduling strategy being used, therefore allowing that two teams can share work, independently of their scheduling strategies. Note that, when splitting work in a shared choice point, first $W(A,0)$ needs to gain (lock) access to the corresponding or-frame, then it moves the next unexplored alternative from the or-frame to the array of open alternatives, updates the or-frame to *null* and unlocks it.

At the end, the array with the open alternatives and the execution stacks of $W(A,0)$ are copied to $W(C,0)$. Figure 4(b) shows the configuration of both teams after the sharing process. In team A, we can see the effect of vertical splitting by observing the new dead nodes in the branch of $W(A,0)$. In team C, we can see that $W(C,0)$ instantiated the work received from $W(A,0)$ as fully private work. $W(C,0)$ will only share its work, and allocate the corresponding or-frames if team C is also using dynamic scheduling, when the scheduler inside the team notifies it to share work with its teammates.

3.4 Algorithms

In this section, we present in more detail the two algorithms that implement the key aspects of our new model.

Algorithm 1 shows the pseudo-code for the $WorkerGetWork()$ procedure that, given an idle worker W belonging to a team T, searches for a new piece of work for W. In a nutshell, we can resume the algorithm as follows. Initially, W starts by selecting a busy worker B from its teammates to potentially share work with (line 3). Next, it sends a share request to B (line 4) and if the request gets accepted, then both workers perform the work sharing procedure, according to the scheduling strategy (dynamic or static) being used in T (line 5). After sharing, W returns to Prolog execution (line 6). Otherwise, if the sharing request gets refused, then W should try another busy worker from T, while there are teammates with available work (line 2).

Algorithm 1 $WorkerGetWork(W, T)$.

```
1:  while $TeamNotFinished(T)$ do
2:    while $TeamWithWork(T)$ do
3:      $B \leftarrow SelectBusyWorker(T)$
4:      if $SendShareRequest(W, B) = ACCEPTED$ then
5:        $ShareWork(W, B)$
6:        return true
7:    if $W = SelectMasterWorker(T)$ then {W will search for work from the other teams}
8:      if $TeamGetWork(W, T)$ then {worker W has obtained work from another team}
9:        return true
10:     else {all teams should finish execution}
11:       $SetTeamAsFinished(T)$
12: return false
```

On the other hand, if all workers in T run out of work (i.e., if all workers are executing the $WorkerGetWork()$ procedure), then one of the workers from T, named the *master worker* W, will be selected to search for work from the other teams (line 7), and for that it executes the $TeamGetWork()$ procedure (line 8), as explained next in Algorithm 2. If the call to $TeamGetWork()$ succeeds, this means that W has obtained a new piece of work from another team and, in such case, W returns to Prolog execution to start exploiting the new available work (line 9). Otherwise, if the call to $TeamGetWork()$ fails, this means that all teams are out of work and, in such case, team T is set as finished (line 11) and all workers in T then finish execution by returning $false$ (line 12).

Next, Algorithm 2 shows the pseudo-code for the $TeamGetWork()$ procedure that, given the master worker W of an idle team T, searches for a new piece of work from the other teams. Initially, W starts by selecting a busy team U from the available set of teams to potentially share work with (line 2). Next, it sends a share request to team U (line 3) and if the request gets accepted, then W performs the work sharing procedure, with the selected sharing worker S from U (lines 4–5), and returns successfully (line 6). Otherwise, if the sharing request gets refused, then W should try another busy team, while there teams with available work (line 1). On the other hand, if all teams run out of work (i.e., if all master workers are executing the $TeamGetWork()$ procedure), then W returns failure (line 7).

Algorithm 2 $TeamGetWork(W, T)$.

```
1: while not $AllTeamsWithoutWork()$ do
2:   $U \leftarrow SelectBusyTeam()$
3:   if $SendShareRequest(T, U) = ACCEPTED$ then
4:     $S \leftarrow GetSharingWorker(U)$
5:     $ShareWork(W, S)$
6:     return true
7: return false
```

4 Conclusions

We have proposed a novel computational model to efficiently exploit implicit or-parallelism from large scale real-world applications specialized for the novel architectures based on clusters of multicores. The main goal behind our proposal is to implement the concept of teams in order to decouple the scheduling of work from the architecture of the system. In particular, we are most interested in the ability of exploiting different scheduling strategies for distributing work among workers and among teams in the same or in different computer nodes.

Currently, we have already started the implementation of the new model in the Yap Prolog system, trying to reuse, as much as possible, the existing infrastructure that supports both dynamic and static scheduling of work for or-parallelism based on the environment copying model. Beyond the implementation of the initial prototype, further work will include: (i) studying load balancing, i.e., how to better distribute work across teams and across workers in a team; (ii) avoid speculative work, i.e., avoid work which would not be done in a sequential system; and (iii) support sequential semantics, i.e., predicate side-effects must be executed by leftmost workers, as otherwise we may change the sequential behavior of the program.

Acknowledgments This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects LEAP (FCOMP-01-0124-FEDER-015008) and PEst (FCOMP-01-0124-FEDER-022701). João Santos is funded by the FCT grant SFRH/BD/76307/2011.

References

1 H. Aït-Kaci. *Warren's Abstract Machine – A Tutorial Reconstruction*. The MIT Press, 1991.

2 K. Ali and R. Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990.

3 K. Ali and R. Karlsson. The Muse Approach to OR-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

4 G. Gupta and E. Pontelli. Stack Splitting: A Simple Technique for Implementing Or-parallelism on Distributed Machines. In *International Conference on Logic Programming*, pages 290–304. The MIT Press, 1999.

5 G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel Execution of Prolog Programs: A Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

6 G. Gupta, E. Pontelli, M. V. Hermenegildo, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–109. The MIT Press, 1994.

7 E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*, pages 819–830. Institute for New Generation Computer Technology, 1988.

8 E. Pontelli, K. Villaverde, Hai-Feng Guo, and G. Gupta. Stack splitting: A technique for efficient exploitation of search parallelism on share-nothing platforms. *Journal of Parallel and Distributed Computing*, 66(10):1267–1293, 2006.

9 R. Rocha, F. Silva, and R. Martins. YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. In *Portuguese Conference on Artificial Intelligence*, number 2902 in LNAI, pages 136–150. Springer-Verlag, 2003.

10 R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System Based on Environment Copying. In *Portuguese Conference on Artificial Intelligence*, number 1695 in LNAI, pages 178–192. Springer-Verlag, 1999.

11 V. Santos Costa, I. Dutra, and R. Rocha. Threads and Or-Parallelism Unified. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 10(4–6):417–432, 2010.

12 V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

13 V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93. ACM, 1991.

14 R. Vieira, R. Rocha, and F. Silva. On Comparing Alternative Splitting Strategies for Or-Parallel Prolog Execution on Multicores. In *Colloquium on Implementation of Constraint and LOgic Programming Systems*, pages 71–85, 2012.

15 R. Vieira, R. Rocha, and F. Silva. Or-Parallel Prolog Execution on Multicores Based on Stack Splitting. In *International Workshop on Declarative Aspects and Applications of Multicore Programming*. ACM Digital Library, 2012.

16 K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. PALS: An Or-Parallel Implementation of Prolog on Beowulf Architectures. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 27–42. Springer-Verlag, 2001.

17 K. Villaverde, E. Pontelli, H. Guo, and G. Gupta. A Methodology for Order-Sensitive Execution of Non-deterministic Languages on Beowulf Platforms. In *International Euro-Par Conference*, number 2790 in LNCS, pages 694–703. Springer-Verlag, 2003.

18 D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

NESSy: a New Evaluator for Software Development Tools[*]

Enrique Miranda[1], Mario Berón[1], German Montejano[1], Maria João Varanda Pereira[2], and Pedro Rangel Henriques[3]

1 **Department of Informatics, Universidad Nacional de San Luis**
Ejército de los Andes 950, Argentina
{eamiranda,mberon,gmonte}@unsl.edu.ar

2 **Department of Informatics, Instituto Politécnico de Bragança**
Quinta de St. Apolónia, Bragança, Portugal
mjoao@ipb.pt

3 **Department of Informatics, Universidade do Minho**
Campus de Gualtar, Braga, Portugal
prh@di.uminho.pt

Abstract

Select the best tool for developing a system is a complex process. There must be considered several aspects corresponding to the domain where the system is going to run. Generally, the domain characteristics only are comprehended by experts. They know very well which are the main characteristics, how they can be combined and which should not be considered. This knowledge is fundamental to select the most appropriate tool for implementing a system that solves problems or automates processes in a specific domain. For this reason, it is difficult to get a tool that allows to establish a ranking of development tools for a particular case. In this paper, NESSy, a system to evaluate software development tools, is presented. This tool implements a multi-criteria evaluation method named LSP (Logic Scoring of Preference). Furthermore, it presents a user-friendly environment for carrying out the evaluation process. LSP uses a set of structures aimed at describing software development tools with the goal of select the best one for a specific problem. The features previously mentioned make NESSy a relevant application to help the software engineer to select the best tool for solving specific problems related to particular domains.

1998 ACM Subject Classification D.2.7. Distribution, Maintenance, and Enhancement

Keywords and phrases Evaluation Method, Elementary Criteria, Aggregation Structure, LSP.

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.21

1 Introduction

During the software development process, the engineer faces several problems. In this context, a common challenge is to select the most appropriate tool to develop software [23, 25, 24, 26]. This problem is not trivial, because the selection is highly dependent on the context and on the application domain [27, 28]. For example, an Integrated Development Environment (IDE) helps the engineer to develop systems when the computer used is powerful.

[*] This work was partially supported by Universidade do Minho and Universidad Nacional de San Luis.

However, it is not true when this last requirement is not met. It is possible to find thousand of examples like the previous one. Unfortunately, the process used by the engineers to select the tools is ad-hoc. This process is based on the engineer's experience and the problem complexity. Both aspects are relevant, nevertheless many other features have to be taken into consideration. In this context, we realized that there is a lack of tools that implement a flexible and configurable evaluation method.

NESSy aims at solving the problem mentioned in the precedent paragraph by implementing LSP (Logic Scoring of Preferences), a multi-criteria evaluation method. In order to simplify the method application, NESSy implements and defines a visual domain specific language. This language is based on graphs and it has several operations to do insert, delete and modify the specification components (nodes and arcs and their corresponding attributes).

To evaluate a development tool using LSP, the following items must be defined: a list of attributes, an aggregation structure and a set of elementary criteria functions [10, 11]. The first component describes all the characteristics that the product must have to simplify the implementation of the problem solution. The second is defined using logical operators and functions that combine the criteria specified. The third maps an attribute value into an elementary preference, i.e. a value into the range [0,100]. This value represents the attribute satisfaction level. Once defined the characteristics, the aggregation structure, and the functions of elementary criterion, an evaluation process is applied to obtain a number that represents a global preference. This preference indicates the satisfaction level of the engineer regarding the Software Development Tool under evaluation. When many tools are evaluated using LSP, it is possible to establish a ranking by sorting the global preference. NESSy implements LSP providing a practical, functional and complete graphical interface This peculiarity makes the selection process easier.

The article is organized as follow. Section 2 explains the Logic Scoring of Preference method. Section 3 describes all the NESSy characteristics, i.e: architecture, environment, the evaluation process, functionalities of its graphical interfaces, etc. Section 4 presents a case study to validate the approach. This case study is concerned with the selection of the best graphical library to build software views using software visualization techniques [29]. Finally, section 5 summarizes the proposal and concludes this article with trends for future work.

2 Logic Scoring Preference

Logic Scoring of Preference (LSP) is a multicriteria evaluation method based in the definition of: a criteria tree, elementary criteria functions and an aggregation structure. LSP is useful to analyze, compare and select, the best alternative from a set of objects being graded and ranked (in our case we are interested in software development tools). In the following subsections, all the LSP components will be explained.

2.1 Criteria Tree

The criteria tree has the characteristics that the tools under evaluation must have. With the goal of developing a complete criteria list, a hierarchical decomposition process is applied. At the end of this process a list of measurable attributes is obtained. In the first instance, the high level characteristics are defined. Then, they are decomposed in sub-characteristics and so on. This process is repeated until obtain the atomic attributes. The result of this task is a tree that describes the main characteristics that the objects under evaluation must meet.

2.2 Elementary Criteria

LSP requires the normalization of the measurable attributes. This normalization is necessary because: i) in several decision contexts the measurement units are different; ii) the values of different attributes may be incomparable.

The LSP attribute normalization is accomplished through the definition of Elementary Criterion Functions. An elementary criterion function maps a value taken by the performance variable in other contained in the interval [0,1] or [0,100]. This value represents the satisfaction level of the performance variable under observation. So, 0 represents a situation where the performance variable does not satisfy the requirements at all, and 1 (or 100) means that the requirement is totally satisfied. The elementary criteria can be classified as: *Absolute* or *Relative*.

An Absolute elementary criterion is used to determine the absolute preference of some attribute. A Relative elementary criterion is employed to establish the relative indicators of the tools under comparison.

Relative elementary criteria are not frequently used for this kind of evaluation. So NESSy only supports the Absolute type of elementary criteria and the Relative one will no more be discussed in this context.

Absolute elementary criteria can belong to different types, as defined below.

- Continuous Variable

  Multivariable: The performance variable is computed by a function. This function receives parameters as its input and returns the value corresponding to the attribute under evaluation. For example, the attribute *Supported Paradigms* can be evaluated by formula 1.

  $$SupportedParadigms = 100 \times \frac{ParadigmsLG}{ParadigmsMax} \tag{1}$$

  In this case, both the *ParadigmsLG* and *ParadigmsMax* are the parameters and the value stored in the variable *SupportedParadigms* is the attribute value.

  Direct: The performance variable has a value that is directly inserted by the evaluator.

- Discrete Variable

  Multilevel: The performance variable can take one value from a set of discrete values. These values are established by the evaluator in the stage of *elementary preference definition*; they correspond to different preference levels. The engineer in the *evaluation* stage must choose a value from that set.

2.3 Aggregation Structure

The elemental preferences, that result from the application of the elementary criteria to the measurable attributes, must be aggregated in order to obtain the global preference. This global preference represents the satisfaction of all the requirements, by the tool under evaluation.

In order to reach the global preference, some aggregation preference functions are used. These functions receive a set of elementary preferences and their corresponding weights as input. The weights represents the relative importance for each preference. The functions return aggregated preferences as their output. All the outputs are aggregated in the next level of the structure. This process is repeated until the global preference is reached. The aggregation function proposed by LSP is presented in formula 2.

$$E = (w_1 e_1^r + w_2 e_2^r + ... + w_k e_k^r)^{\frac{1}{r}} \qquad (2)$$

where:

$$-\infty \leq r \leq +\infty$$
$$0 \leq w_i \leq 1 \text{ and } i = 1..k$$
$$w_1 + ... + w_k = 1$$

E is a general instantiation scheme which produces a continuous spectrum of aggregation functions, depending on the value of r. Table 1 shows the most relevant values for r, taking into account the number, n, of function input values. For example, if the operator under consideration is D- and it receives three input values, then the value of r in the precedent formula is 2.19. To be clearer, r represents the conjunction-disjunction degree of each operator. We say that r generates several functions known as *Conjunctive Disjunctive Generalized functions* (CDG). These functions are the operators used to aggregate the elementary preferences. The formula employed to compute the values in table 1 is explained in [11].

2.4 The Evaluation Process

The evaluation process is carried out defining the values of all performance variables for each tool under evaluation. In this way, for each system, a global preference will be computed and this value is used to elaborate the ranking. Figure 1 shows a representation of the LSP Evaluation Method.

The global preference is obtained from the computation (represented in figure 1 by L($E_1..En$)) of all the elementary preferences.

And these elementary preferences are the result of applying the elementary criteria to the performance variables. Finally, the elementary criteria can be computed because the engineer provides the required values.

2.5 Related Work

Multiple Criteria Decision Methods (MCDMs) are used to evaluate and make decisions regarding some problems that admit a finite number of solutions [23]. Nowadays there

Table 1 Values of r corresponding to each CDG.

Operation Name	Symbol	r			
		n=2	n=3	n=4	n=5
Disjunction	D	$+\infty$	$+\infty$	$+\infty$	$+\infty$
Strong Cuasi Disjunction	D+	9.52	11.09	12.28	13.16
Cuasi Disjunction	DA	3.83	4.45	4.82	5.09
Weak Cuasi Disjunction	D-	2.02	2.19	2.30	2.38
Arithmetic Media	A	1.00	1.00	1.00	1.00
Weak Cuasi Conjunction	C-	0.26	0.20	0.17	0.16
Cuasi Conjunction	CA	-0.72	-0.73	-0.71	-0.67
Strong Cuasi Conjunction	C+	-3.51	-3.51	-2.18	-2.61
Conjuction	C	$-\infty$	$-\infty$	$-\infty$	-∞

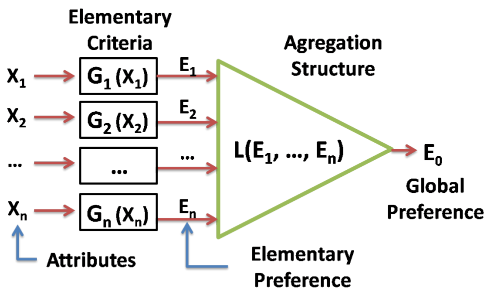


Figure 1 LSP Method Representation.

are a considerable number of MCDMs that are used in decision making in various topics. However, it was difficult to find, in the literature, systems that implement this kind of methods. The MCDMs most recently used and implemented are ELECTRE (ELimination Et Choix Traduisant la REalité) and PROMETHEE (Preference Ranking Organization METHod for Enrichment Evaluations). Both methods use a similar approach than LSP. ELECTRE was proposed by Bernard Roy in 1971 [24]. The tools that implement different versions of ELECTRE [19, 1, 20], generally have some drawbacks, for example: they employ traditional interaction strategies, they do not define a Domain Specific Language (DSL) to be used during the evaluation process (even when it would be very functional), they use complex fuzzy logic that user must deal with, etc. PROMETHEE was developed by Brans and further extended by Vincke and Brans [9]. PROMETHEE is quite simple in conception and application compared with the other MCDMs. Therefore, it is widely used in research and practical contexts. Two of the most used implementations of PROMETHEE are Decision LAB and PROCALC [8]. Nevertheless, both have similar drawbacks comparing to ELECTRE implementations. Other MCDM implementations such as AHP [27], MAUT [17], etc., were studied. However we could not find those implementations available for a deeper comparative analysis. In the case of LSP (Logic Scoring of Preference), there are some tools based on this method, as the one presented in this article. However, these tools have the following drawbacks: i) they frequently are developed for specific cases (e.g. LSPmed [13], webQEM [21]); ii) they do not provide a DSL (even when this kind of language might be clearly useful); iii) some present a poor user interface (e.g ISEE [12]); iv) they do not offer complete documentation; v) most of them are not available to be used or analyzed.

NESSy tries to tackle the problems before mentioned by providing a user-friendly interface, a visual DSL, a simple evaluation process, among other features.

3 NESSy

In this section, NESSy characteristics are described. In first place, and with the goal of providing an overview of NESSy components, the architecture will be explained. Then the interface where the engineer carries out the evaluation process will be presented. Finally, the evaluation process will be in detail explained.

3.1 Architecture

Figure 2 shows NESSy architecture. NESSy is composed of four components: *Criteria Tree Constructor* (CTC), *Aggregation Structure Constructor* (ASC), *Elementary Criteria Specifier* (ECS) and *Evaluator*.

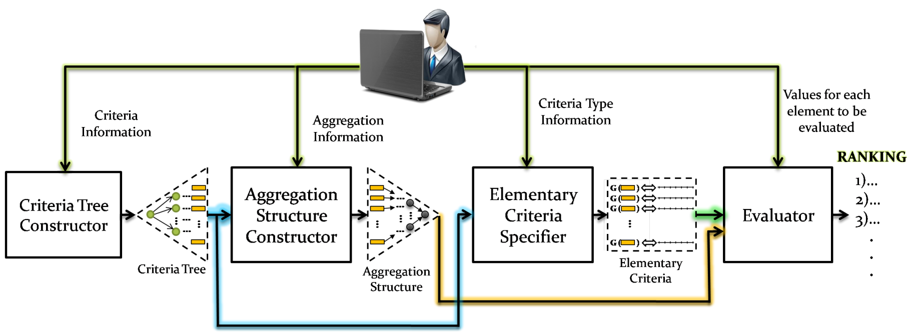

Criteria Information
Aggregation Information
Criteria Type Information
Values for each element to be evaluated
RANKING
Criteria Tree Constructor
Criteria Tree
Aggregation Structure Constructor
Aggregation Structure
Elementary Criteria Specifier
Elementary Criteria
Evaluator
1)...
2)...
3)...

Figure 2 NESSy Architecture.

CTC receives as its input *Criteria Information* (CI) and produces as its output the *Criteria Tree* (CT). CTC allows to define criteria for characterizing the tools to be evaluated. Clearly, this component has functionalities like: *Add Criterion*, *Delete Criterion*, *Modify Criterion*, etc. In this context CI represents the expert's knowledge. It is important because the evaluation process depends on the CT. If CT is not well built the results obtained will not be correct. The CT structure reflects the successive decomposition of the characteristics into sub-characteristics and so on until obtaining the measurable attributes.

ASC adds the logic needed to carry out the evaluation process.

It is important to mention that the aggregation structure is built bottom-up from the leaves (attributes) until the last operator is obtained. This particular operator produces the tool global preference. This component has functionalities such as: *Add Logical Operator*, *Delete Logical Operator*, *Add Weight*, etc.

ECS receives as its input the CT. Like ASC, this component takes into consideration the leaves of the CT, i.e. the measurable attributes. For each attribute, this component selects its type and, according to the type, to define its evaluation function.

Finally, the *Evaluator* receives as its input both the AS and the refined ECs. Then the Evaluator traverses the AS and, using the information provided by the engineer, produces a ranking of the tools under evaluation.

3.2 Interface

NESSy interface is composed of four components: *Menu*, *Top Panel* (TP), *Central Panel* (CP) and *Bottom Panel* (BP) (see figure 3). Menu exhibits a classical set of project management operations. The operations available are: *Load*, *Save*, *New Project*, *Exit* and *Help*.

TP contains the buttons *Load*, *Save*, *Validate* and the field *Current Stage*. The buttons have the same functionality that the options provided in the Menu component. Current stage field indicates the process stage undergoing, i.e. the one carried out in the CP.

CP displays all the components needed to carry out the evaluation process. This process has four stages which are explained in the next subsection. The elements shown in CP depend on the process stage. In the first two stages, the elements exhibited are concerned with the construction of structures needed by the evaluation process. In the other stages, the elements exhibited are related with the presentation of intermediate and final results.

BP has the buttons *Previous* and *Next* and *Contextualization Figures*. The buttons are employed to proceed to the next stage or go back to the previous one. The Contextualization Figures are useful to indicate the current phase and to notify the evolution of the evaluation process (its present level).

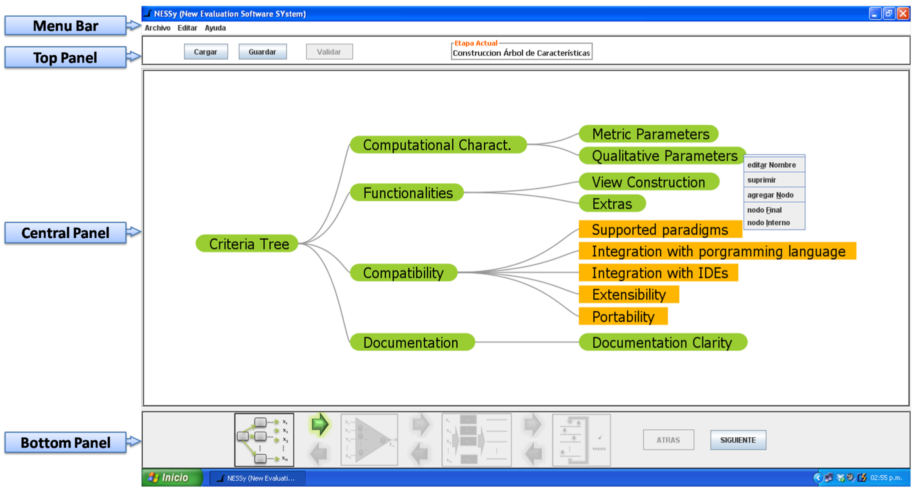


Figure 3 NESSy Environment Screenshot.

3.3 The Process

The evaluation process provided by NESSy has four stages (the same specified in LSP section), they are: **CT Constructor**, **AS Constructor**, **EC Specifier** and **Evaluator**. The following subsections explain in detail each stage.

3.3.1 CT Constructor

In this stage, the main characteristics used to compare the tools to be evaluated will be defined. This set of characteristics is represented using a tree. The leaves of this tree are measurable attributes. It is important to notice that the evaluation process only use the tree leaves. Nevertheless, the progressive elaboration of the tree from its root until its leaves has the advantages mentioned below:

1. Makes easier the Criteria Definition: The creation of internal nodes allows to apply a top-down decomposition process. In this process, the engineer defines high level characteristics and decomposes them in sub-characteristics until obtain the attributes.
2. Improves the Visualization: At the end of the definition process, it is possible to observe a tree structure that shows all the criteria defined. This global view permits to do some reasoning and this particularity helps to improve the structure.

Each time that a new project is created, NESSy shows, in its central panel, the tree root. The nodes are created pressing the mouse right button and selecting the option *Add Node* from the pop-up menu. In order to improve the visualization, the tree nodes are distinguished using different shapes and colors. The *Internal Nodes* (characteristics, sub-characteristics, etc.) have elliptic shape and their background is green. The leaves (attributes) have square shape and their background is yellow. The nodes can be edited just pressing the mouse right button. The following operations are then available:

1. Edit Name: It allows to modify the node name.
2. Delete Node: It is employed to delete a node. This operation is implemented as a cascade deletion, i.e. all the sub-tree corresponding to the node will be deleted.
3. Add Node: This operation is used to add a new node in the tree. The new node is tagged as *New Node* and it is setted as *Internal Node*.

4. Final Node: It is utilized when the engineer wants to change the node type to *Final Node*, i.e. an attribute.
5. Internal Node: It is utilized when the engineer wants to change the node type to *Internal Node*, i.e. a characteristic or sub-characteristic.

A larger number of nodes increases the complexity of visualizing and organizing the tree structure. For this reason, NESSy provides visualization functionalities such as:

1. Zoom in and Zoom out: It is possible to zoom in or zoom out all the structure. Zoom in is carried out holding pressed the mouse right button and moving it down. To do a zoom out the same tasks must be done except that the mouse must be moved up.
2. Drag and Drop: It is employed to move all the structure and to focus on the structure sector under analysis. This functionality is achieved by pressing the mouse left button and moving it to the position wished.
3. Zoom to fit: It is used when the structure is out of focus. NESSy provides the operation *Zoom to fit* to achieve that. This operation puts the structure on the center of the central panel and executes the operations necessary for its total visualizaton. To accomplish a zoom to fit the central mouse button must be pressed.

Along this stage, NESSy guarantees that:

1. It is not possible to delete the root node.
2. The root can not be a final node.
3. The final nodes have different names.
4. The Criteria Tree has at least two final nodes (attributes).
5. An internal node cannot be converted into a final node if it is the root of a sub-tree.

Figure 3 shows a fragment of the Criteria Tree used to compare graphical libraries. Notice that among the four characteristics presented at the first level of the CT, only *Compatibility* is decomposed into its elementary components, the attributes (tree leaves).

3.3.2 AS Constructor

In this stage, the attributes defined in the previous one (the CT leaves) are shown. They are placed in the central panel following the order established in the CT.

The objective of this stage is to build a DAG (Directed Acyclic Graph), such that the initial nodes (the DAG source nodes) are the attributes; these nodes are then aggregated until obtaining just one node (the DAG sink node). The resulting value of this node represents *Global Preference*. Each node, except those that represent attributes, must be associated to a LSP operator. Furthermore, the arcs must be labeled with a number. This number represents a weight. The elements of the DAG are differentiated through their shape and color. In this case, we use a square shape with yellow color for the attributes. The operators, i.e the internal nodes, have elliptic shape with gray color. The arcs are represented by arrows with gray lines and yellow heads.

The aggregation structure is built using four pop up menus.

The first, *Add Node*, is employed to add an LSP operator node. When the left button is pressed on the menu another pop up menu appears. It presents the following options: i) Select Operator, this option permits to assign the corresponding logical operator to the node. ii) Delete, it is utilized to delete the current node.

The second, *Add Arc*, is used to connect the nodes. This connection can be carried out between an attribute and one operator or between two operators. When an arc is pressed another sub-menu is displayed; it offers several options to modify the node label, and to delete the arc.

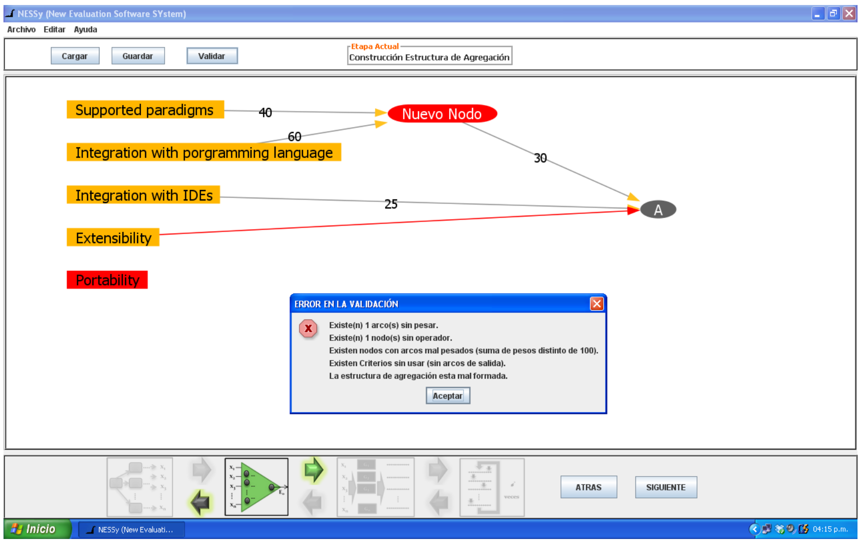
Applying the operations described above, it is possible to build any graph. However, LSP method does not work with any graph, some conditions must be fulfilled. They are listed below:

1. The graph must be a DAG.
2. The attributes have not input degree.
3. All the operator nodes must be defined as LSP operators.
4. All the arcs must have a weight p, $1 \leq p \leq 100$, assigned.
5. The aggregation structure can not contain parallel arcs.
6. The input degree of the operator nodes ranges between two and five.
7. The sum of the weights of the input arcs of an operator node must be equal to 100.
8. There are only one sink node.
9. The *Left Ideal*[1] of the sink node is composed by all the structure nodes.

Before proceeding to the next stage the aggregation structure must be validated. In other words, NESSy must verify that the Aggregation Structure built complies with all the conditions listed above. NESSy carries out this task when the button *Validate* is pressed. If the aggregation structure is not correct, NESSy shows the errors found. Figure 4 illustrates the precedent situation, using an example based in the evaluation of graphical libraries. The errors detected in this example are:

1. There is one arc without weight – the arc with red color has no weight.
2. There is one node without operator – the node with label *New Node* has no LSP operator assigned.
3. There are nodes with wrongly pounded arcs – the weight is not correct (the sum is not equal to 100), as happens with node labeled *a* (one input arc has no weight assigned).
4. There is one criterion without use – *Portability* is an isolated node.
5. The aggregation structure is incorrectly formed – There is one node isolated: *Portability*.

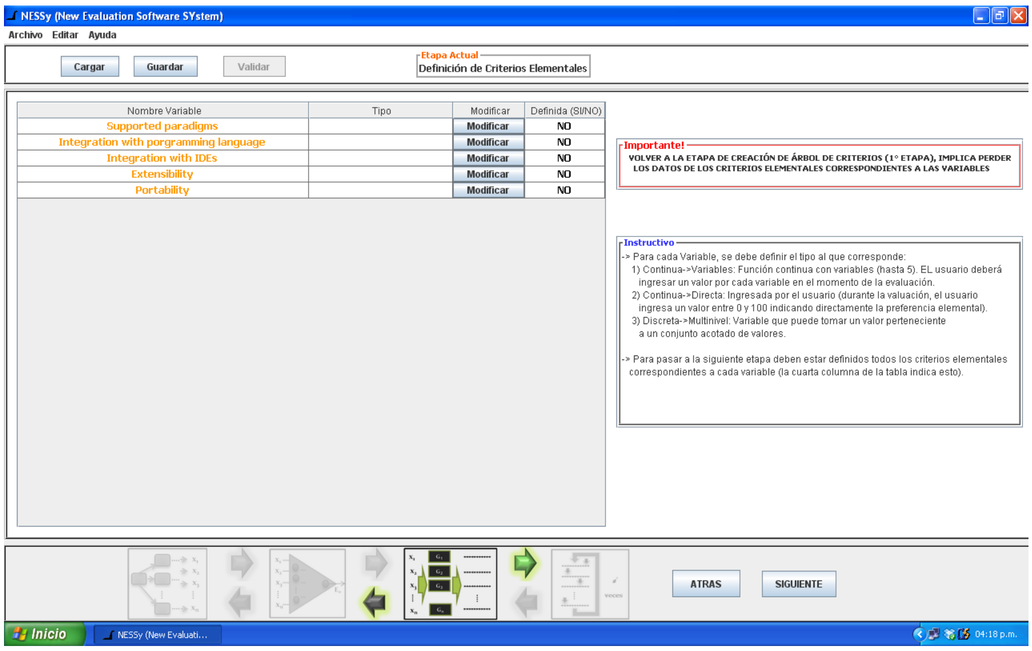
[1] Let G=(P,E) a graph where P is a node set and E is a relation defined on P, $E \subseteq P \times P$, and let x be a node $x \in P$ then LeftIdeal(x)=$\{y \in P/\rho(y,x)\}$ where $\rho(y,x)$ denotes a path from y to x.



ERROR EN LA VALIDACIÓN

Existe(n) 1 arco(s) sin pesar.
Existe(n) 1 nodo(s) sin operador.
Existen nodos con arcos mal pesados (suma de pesos distinto de 100).
Existen Criterios sin usar (sin arcos de salida).
La estructura de agregación esta mal formada.

Figure 4 Error detection in the definition of an Aggregation Structure.



Etapa Actual
Definición de Criterios Elementales

Nombre Variable	Tipo	Modificar	Definida (SI/NO)
Supported paradigms		Modificar	NO
Integration with porgramming language		Modificar	NO
Integration with IDEs		Modificar	NO
Extensibility		Modificar	NO
Portability		Modificar	NO

Importante!
VOLVER A LA ETAPA DE CREACIÓN DE ÁRBOL DE CRITERIOS (1° ETAPA), IMPLICA PERDER LOS DATOS DE LOS CRITERIOS ELEMENTALES CORRESPONDIENTES A LAS VARIABLES

Instructivo
-> Para cada Variable, se debe definir el tipo al que corresponde:
1) Continua->Variables: Función continua con variables (hasta 5). EL usuario deberá ingresar un valor por cada variable en el momento de la evaluación.
2) Continua->Directa: Ingresada por el usuario (durante la valuación, el usuario ingresa un valor entre 0 y 100 indicando directamente la preferencia elemental).
3) Discreta->Multinivel: Variable que puede tomar un valor perteneciente a un conjunto acotado de valores.

-> Para pasar a la siguiente etapa deben estar definidos todos los criterios elementales correspondientes a cada variable (la cuarta columna de la tabla indica esto).

Figure 5 Interface corresponding to the Third Stage.

3.3.3 EC Specifier

At this stage, the type of each elementary criterion (EC) and the formula to evaluate it are specified. NESSy interface to support this stage is shown in figure 5.

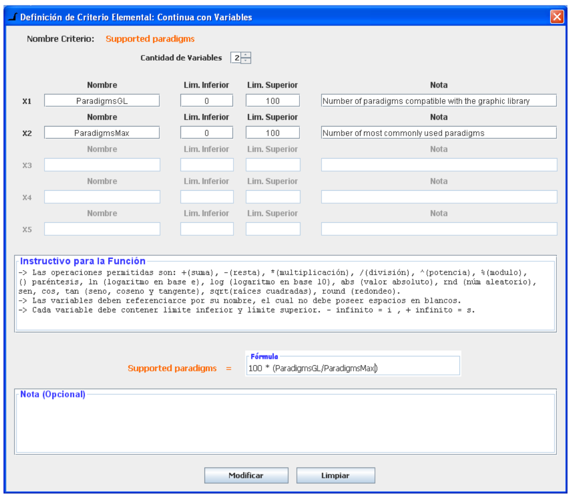
Observing this figure, it is possible to identify the elementary criteria displayed on the left side, and two text areas on the right side. The first contains an advice to inform the user that going back to stage one, all work done will be lost. The second displays some tips regarding the criteria specification.

To specify the ECs, the left side of the screen, shown in figure 5, has three columns: i) `Variable Name` contains the criteria names, ii) `Type` allows to select the type of each criterion, iii) `Modify` is a button that allows to add information concerned with the criterion. A forth column is included, *Defined (yes/no)*, to indicate whether the criterion has been defined, or not.

In order to specify the function for each elementary criterion, the engineer must follow the steps described below.

1. Select the criterion type.
2. Press the button *Modify* to open the corresponding pop-up panel.
3. Complete the information required filling the form in that panel.

It is important to remark that NESSy supports the three types of elementary criteria explained in section 2, they are: Continuous Variable – Multivariable (see figure 6), and Direct; and Discrete Variable – Multilevel (see figure 7).


Definición de Criterio Elemental: Continua con Variables

Nombre Criterio: Supported paradigms

Cantidad de Variables 2

	Nombre	Lim. Inferior	Lim. Superior	Nota
X1	ParadigmsGL	0	100	Number of paradigms compatible with the graphic library
X2	ParadigmsMax	0	100	Number of most commonly used paradigms
X3				
X4				
X5				

Instructivo para la Función

```
-> Las operaciones permitidas son: +(suma), -(resta), *(multiplicación), /(división), ^(potencia), %(modulo),
() paréntesis, ln (logaritmo en base e), log (logaritmo en base 10), abs (valor absoluto), rnd (núm aleatorio),
sen, cos, tan (seno, coseno y tangente), sqrt(raíces cuadradas), round (redondeo).
-> Las variables deben referenciarce por su nombre, el cual no debe poseer espacios en blancos.
-> Cada variable debe contener límite inferior y límite superior. - infinito = i , + infinito = s.
```

Supported paradigms = Fórmula: 100 * (ParadigmsGL/ParadigmsMax)

Nota (Opcional)

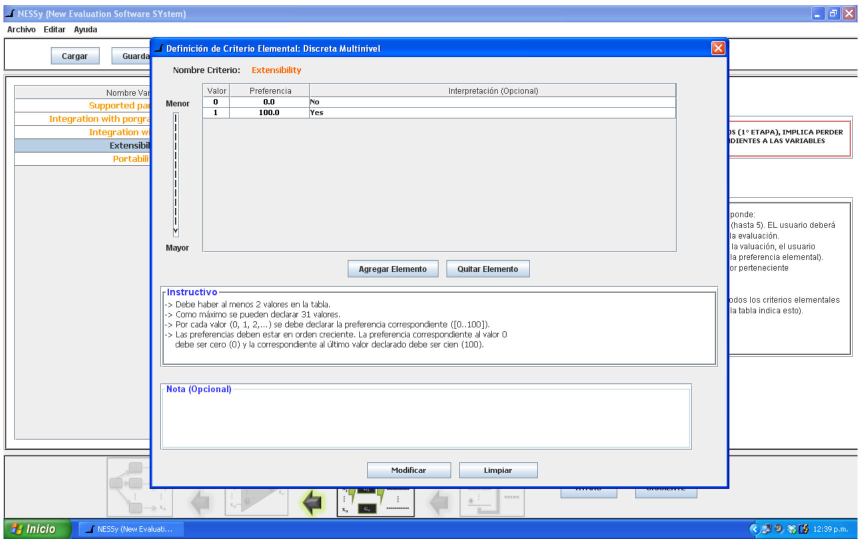
Figure 6 Pop up Panel to specify Criteria of type Continuous Variable (Multivariable).

3.3.4 Evaluator

In this phase, the engineer must:

1. Define the tools to evaluate.
2. Provide, for each tool the information required by the elementary criteria in order to proceed the evaluation.

The evaluation phase also has its pop up panel. In this panel, it is possible to find a table and two buttons (*Add Element* and *Delete Element*). The table allows to visualize the software system to be submitted to the evaluation process. It has four columns, they are: i) Name: It is the name of the tool; ii) Input Values: Each cell in this column has a button. When this button is pressed a pop up panel appears. The format of this panel depend on the type of elementary criterion. If it is a *Continuous Variable* the engineer must fill the


NESSy (New Evaluation Software SYstem)

Definición de Criterio Elemental: Discreta Multinivel

Nombre Criterio: Extensibility

Valor	Preferencia	Interpretación (Opcional)
0	0.0	No
1	100.0	Yes

Instructivo

-> Debe haber al menos 2 valores en la tabla.
-> Como máximo se pueden declarar 31 valores.
-> Por cada valor (0, 1, 2,...) se debe declarar la preferencia correspondiente ([0..100]).
-> Las preferencias deben estar en orden creciente. La preferencia correspondiente al valor 0 debe ser cero (0) y la correspondiente al último valor declarado debe ser cien (100).

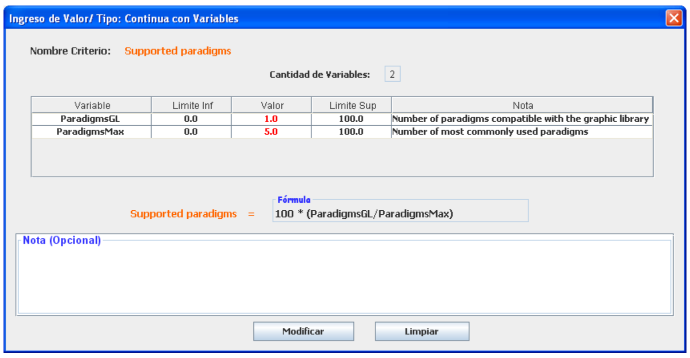
Nota (Opcional)

Nombre Var
Supported pa
Integration with porgr
Integration w
Extensibil
Portabili

OS (1° ETAPA), IMPLICA PERDER
DIENTES A LAS VARIABLES

ponde:
(hasta 5). EL usuario deberá
la evaluación.
la valuación, el usuario
la preferencia elemental).
or perteneciente
odos los criterios elementales
la tabla indica esto).

Figure 7 Pop up Panel to specify Criteria of type Discrete Variable (Multilevel).



Nombre Criterio: Supported paradigms

Cantidad de Variables: 2

Variable	Limite Inf	Valor	Limite Sup	Nota
ParadigmsGL	0.0	1.0	100.0	Number of paradigms compatible with the graphic library
ParadigmsMax	0.0	5.0	100.0	Number of most commonly used paradigms

Fórmula
Supported paradigms = 100 * (ParadigmsGL/ParadigmsMax)

Nota (Opcional)

Figure 8 Panel to Evaluate a Continuous Variable Criterion.

data required by the panel shown in figure 8. If it is *Constant Value* the engineer just needs to provide the preference level. Finally, if it is *Discrete Multi-Level*, all the values defined in the previous steps are shown again. The engineer must select the preference level wished.

The next step is to proceed with the global evaluation. This process uses the Aggregation Structure combining both the criteria and the LSP operators. The Aggregation Structure is traversed and the value for each criterion is computed, following the LSP semantics. At the end of the process, the global preference is computed.

The process described above is applied to all the tools under analysis and the ranking is established taking into consideration the global preference of each tool.

4 Case Study: Visualization Libraries

Software Visualization (SV) is a discipline of Software Engineering aimed at creating and displaying useful static or dynamic views of software [2, 3, 18]. A view is a graphical representation that helps to understand some software aspects.

In order to build a view, many artifacts must be defined. An artefact is a concept used to refer an object belonging to a particular visualization.

Building views and their associated artefacts can be a complex task. For example, to build a graph-based view by implementing the graph from the scratch is a hard task and it consumes much time and efforts. The engineer must consider: the internal representation, the complexity of the operations and different strategies to visualize graphs.

When a complete and tested graph library is used much time is saved and many programming errors are avoided.

The following sections describes how NESSy was used to select the most appropriate tool to rig up software visualizations [4, 5].

4.1 Criteria Tree

The Criteria Tree shown in figure 9 has been built after a deep research in the context of Software Visualization. As it possible to observe, the tree has four main characteristics: *Computational*, *Functional*, *Compatibility* and *Documentation*.

The Computational characteristic is concerned with the computation of two kind of sub-characteristics: *Quantitative* and *Qualitative*. The first analyzes simple metrics that must be taken into account when a software tool is selected. The second is related with properties concerning the current use of the library.

The Functional characteristic contains properties that a visualization library must have. For example, a visualization library must have a large number of visual artifacts, because in the other way it will not be useful.

The Compatibility characteristic describes the possibility of using the library with various paradigms and in different platforms.

Finally, the Documentation characteristic includes a relevant software aspect: the existence of well-formed and organized texts describing the software package. Many times, the engineers reject using a powerful library because it is complex to understand how it works. It is due to the absence of a good and clear documentation.

To finish this section, it is relevant to remark that, the actual tree has more criteria (see [14] for more details) than those here considered. However, many of them need to be disaggregated in order to be used by NESSy (for more details read [14]).

4.2 Aggregation Structure

The aggregation structure was built taken into account: i) The user's experience using graphical libraries; ii) Experts recomendations; iii) The state of the art of graphic libraries in the context of Software Visualization, as exposed by Miranda in [14].

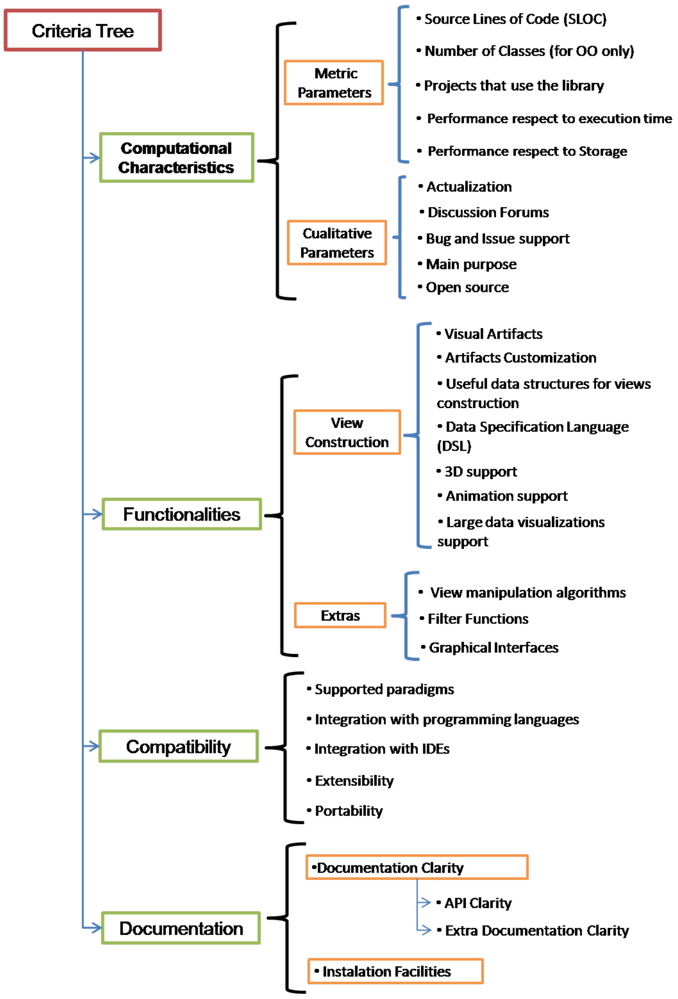

Criteria Tree
Computational Characteristics
Metric Parameters
• Source Lines of Code (SLOC)
• Number of Classes (for OO only)
• Projects that use the library
• Performance respect to execution time
• Performance respect to Storage
Cualitative Parameters
• Actualization
• Discussion Forums
• Bug and Issue support
• Main purpose
• Open source
Functionalities
View Construction
• Visual Artifacts
• Artifacts Customization
• Useful data structures for views construction
• Data Specification Language (DSL)
• 3D support
• Animation support
• Large data visualizations support
Extras
• View manipulation algorithms
• Filter Functions
• Graphical Interfaces
Compatibility
• Supported paradigms
• Integration with programming languages
• Integration with IDEs
• Extensibility
• Portability
Documentation
•Documentation Clarity
• API Clarity
• Extra Documentation Clarity
• Instalation Facilities

Figure 9 Criteria Tree.

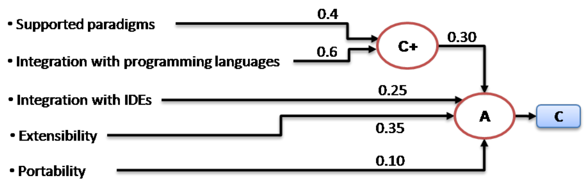


Figure 10 Aggregation Structure corresponding to High Level Characteristic *Compatibility*.

Table 2 $Paradigms_{GL}$ and $Paradigms_{Max}$ description.

Name	Min.	Max.	Note
$Paradigms_{GL}$	0	100	Paradigms supported by the graphic library
$Paradigms_{Max}$	0	100	Maximum number of paradigms supported by a graphic library

Figure 10 shows the aggregation structure corresponding to the high level characteristic *Compatibility*. The Compatibility's partial preference is computed by using two operators.

The first, C+, a quasi-conjunctive function, aggregates *Supported Paradigms* and *Integration with Programming Languages*. This operator is employed when the input requirements are mandatory. Thus if one of the input values is zero, the operation result will be zero. The second operation A (the arithmetic average) is a neutral function (neither conjunctive nor disjunctive). It aggregates the first result with the three criteria *Integration with IDEs*, *Extensibility* and *Portability*. The reader willing to know the full aggregation structure can read [14].

4.3 Elementary Criteria Functions

The approach to specify the Elementary Criteria Functions was described in section 3.3.3. To illustrate how this task is carried out, a simple example is presented: the specification of the elementary criterion *Supported Paradigms*. This criterion is of type Multivariable (a Continuous Variable) and depends on two parameters $Paradigms_{GL}$ and $Paradigms_{Max}$ that are described in table 2. The value of *Supported Paradigms* is determined by formula 1.

Using NESSy to accomplish this task, it is necessary to start specifying the criterion type in the form shown in figure 5. After this action, a new pop-up panel will spring out. Then the description of the two parameters and the evaluation formula, presented above, shall be filled in the form associated to that pop-up panel, as can be seen in figure 6.

4.4 Evaluation

In order to show NESSy usefulness, three graphical libraries were evaluated: Graphviz, JUNG and Prefuse. Graphviz is open source graph visualization software [15]. This library has been used in several scientific projects. Jung provides a common and extensible language for the modeling, analysis and visualization of data that can be represented as a graph or network [16]. Prefuse is a software tool for creating rich interactive data visualization [22].

At this stage all the values required to evaluate the elementary criteria were provided. This task was accomplished for each library. It is important to notice that the values previously mentioned were obtained from: i) Graphic libraries Web Site; ii) Graphic libraries Documentation; iii) Evaluators Experience.

The *Global Preferences* obtained for each library are shown in table 3. In the same table it is possible to see the decomposition of the Global Preference of each tools into its components

Table 3 Final Scores obtained by NESSy.

High-Level Characteristics	Graphviz	JUNG	Prefuse
Computational Characteristics	**69.0892**	51.8395	47.4708
Functionalities	64.5728	63.2473	**76.5484**
Compatibility	**88.1042**	75.2208	75.2208
Documentation	74.7158	**88.6803**	85.2318
Final Scores	**71.7626**	66.1774	67.9197

(the high level characteristic preferences). Graphiz was ranked in the first position (achieve the maximal punctuation for the Global Preference) due to values got for the characteristics *Computational* and *Compatibility*.

5 Conclusion and Future Work

In the context of a bilateral cooperation project devoted to the research of Program Comprehension and Language-based Tools, we decided to adopt Logic Scoring of Preference (LSP)—a multi-criteria Evaluation Method adaptable to several domains, that was being applied by the Argentinean team for a long time in different areas—as a method to compare or select software systems (as discussed for instance in [7] or [6]).

This decision has created the need for a tool that could help in the application of LSP, leading to the development of NESSy, *a new evaluator for software development tools*, that was presented in this paper. In order to correctly follow the LSP approach, NESSy has four stages.

The first stage allows the engineer to define the criteria tree. The second is concerned with the definition of an aggregation structure. The third phase allows to define the elementary criteria types and the respective evaluation functions. Finally, the fourth stage uses the aggregation structure and the elementary criteria to produce a global preference. This preference represents the engineer satisfaction level regarding the tool evaluated.

NESSy provides an easy-to-use interface to support the user work along all these four steps. As a proof of concept, NESSy was used to rank three powerful graphic libraries: *Graphviz*, *JUNG* and *Prefuse*. The main goal was to establish which of them provide more functionalities to build graph-based software views. The results obtained indicate that Graphviz is better (more helpful) than Jung and Prefuse. It is because Graphviz got the best scores concerning the Computational Characteristics and Compatibility. These characteristics were considered more important by the Aggregation Structure designers.

From the experience gained using NESSy in laboratory contexts, we can say: NESSy is user-friendly, has an attractive and easy to learn visual DSL which is employed to specify the aggregation structure, NESSy uses few computational resources and the time consumed to produce the result is acceptable. The *evaluation task* is a difficult process, either using NESSy tool or adopting the traditional manual approach. However, the advantage of using NESSy is that a considerable amount of work (e.g Criteria Tree, Agregation Structure and Elementary Criteria) is already done for future evaluations.

The future work is oriented in four directions. The first is concerned with improving NESSy adding the following characteristics: i) New types of elementary criteria, and ii) More support for project management.

The second is related with the elaboration of strategies to automatize the evaluation process. Currently, all the necessary data is provided manually. However, some elementary criteria can be automatically computed. An example of this assertion is the metric SLOC (Source Lines of Code). For this reason, we intend to study the possibility of using plug-ins in order to automatize the evaluation of some attributes.

The third is related with the improvement of the Criteria Tree for the Software Visualization Domain. As was mentioned along this paper, the CT is wider than the presented in section 4.1. Many criteria were not included because they need to be disaggregated. This problem motivates further research for producing a more complete CT.

Finally, we also plan to explore the application of NESSy to other areas such as: Reverse Engineering, Program Comprehension and Re-Engineering. The goal is to define the Criteria Tree, Aggregation Structure and Elementary Criteria for specific problems in these areas. For example, if a Program Comprehension tool is needed in a specific context, the components previously mentioned, can help to select the best option for this particular situation. Obviously, NESSy is fundamental to carry out this task properly.

References

1 Aicha Aguezzoul, B. Rabenasolo, and A.-M. Jolly-Desodt. Multicriteria decision aid tool for third-party logistics providers' selection. In *Service Systems and Service Management, 2006 International Conference on*, volume 2, pages 912–916, 2006.

2 T. Ball and SG Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

3 Michael Balzer, Andreas Noack, Oliver Deussen, and Claus Lewerentz. Software Landscapes: Visualizing the Structure of Large Software Systems, 2004.

4 S. Bassil and R. Keller. A Qualitative and Quantitative Evaluation of Software Visualization Tools. *Proc. of the IEEE Symposium on Information Visualization*, pages 69–75, 2001.

5 M. Beron, P. Henriques, and R. Uzal. Program Inspection to interconnect Behavioral and Operational Views for Program Comprehension. *Ph.D Thesis Dissertation at University of Minho. Braga. Portugal*, 2010.

6 M. M. Beron, D. Cruz, M. J. Varanda Pereira, P. R. Henriques, and R. Uzal. Evaluation criteria of software visualization system used for program comprehension. *3a Conferencia Nacional em Interacção Pessoa-Máquina*, 03:285, 2008.

7 Mario Marcelo Berón. *Program Inspection to interconnect the Behavioral and Operational Views for Program Comprehension.* PhD thesis, National University of San Luis & University of Minho, Nov 2009.

8 Jean-Pierre Brans and Bertrand Mareschal. Promethee methods. In *Multiple criteria decision analysis: state of the art surveys*, pages 163–186. Springer, 2005.

9 J.P. Brans, Ph. Vincke, and B. Mareschal. How to select and how to rank projects: The promethee method. *European Journal of Operational Research*, 24(2):228 – 238, 1986. <ce:title>Mathematical Programming Multiple Criteria Decision Making</ce:title>.

10 J.J. Dujmovic. A Method for Evaluation and Selection of Complex Hardware and Software Systems. *The 22nd Int'l Conference for the Resource Management and Performance Evaluation of Enterprise CS. CMG 96 Proceedings*, 1:368–378, 1996.

11 J.J. Dujmovic, R. Elnicki, University of Florida, and United States. National Bureau of Standards. *A DMS Cost/benefit Decision Model: Mathematical Models for Data Management System Evaluation, Comparison and Selection (part 1 of Second Deliverable).* National Bureau of Standards, 1981.

12 Jozo Dujmović and Metin Kadaster. A technique and tool for software evaluation. *Evolution*, 374:246, 2002.

13 Jozo J Dujmović, Jeffrey W Ralph, and Leslie J Dorfman. Evaluation of disease severity and patient disability using the lsp method. In *Proceedings of the 12th Information Processing and Management of Uncertainty international conference (IPMU 2008)*, pages 1398–1405.

14 Miranda Enrique. Evaluación de funcionalidades de visualización de software provistas por librerías gráficas. licentiate thesis. 2013.

15 GraphViz-Team. http://www.graphviz.org/, 2011.

16 JUNG-Team. http://jung.sourceforge.net/, 2011.

17 Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, 1993.

18 K. Mens, T. Mens, and M. Wermelinger. Supporting software evolution with intentional software views. *Proceedings of the International Workshop on Principles of Software Evolution*, pages 138–142, 2002.

19 Gholam Ali Montazer, Hamed Qahri Saremi, and Maryam Ramezani. Design a new mixed expert decision aiding system using fuzzy electre iii method for vendor selection. *Expert Syst. Appl.*, 36(8):10837–10847, October 2009.

20 V. Mousseau, R. Slowinski, and P. Zielniewicz. A user-oriented implementation of the electre-tri method integrating preference elicitation support. *Comput. Oper. Res.*, 27(7-8):757–777, June 2000.

21 L. Olsina and G. Rossi. Measuring Web Application Quality with WebQEM. *IEEE MultiMedia, 2002*, 09(4):20–29, 2002.

22 Prefuse-Team. http://prefuse.org/, 2011.

23 Carlos Romero. *Teoría de la Decisión Multicriterio: Conceptos, técnicas y aplicaciones*. Alianza Editorial: Madrid., 1993.

24 B. Roy. Problems and methods with multiple objective functions. *Mathematical Programming*, 1:239–266, 1971.

25 Bernard Roy. The outranking approach and the foundations of electre methods. *Theory and Decision*, 31:49–73, 1991.

26 M.J.; Ríos-Insua S Ríos, S.; Ríos-Insua. *Procesos de Decisión Multicriterio*. 1989.

27 T. Saaty. How to make a decision: The analytic hierarchy process. *European Journal of Operational Research*, 48(1):9–26, September 1990.

28 Herbert Alexander Simon. *The New Science of Management Decision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1977.

29 S. Tilley and S. Huang. On selecting software visualization tools for program understanding in an industrial context. *iwpc*, 00:285, 2002.

Supporting Separate Compilation in a Defunctionalizing Compiler

Georgios Fourtounis and Nikolaos S. Papaspyrou

School of Electrical and Computer Engineering
National Technical University of Athens, Greece
`{gfour, nickie}@softlab.ntua.gr`

Abstract

Defunctionalization is generally considered a whole-program transformation and thus incompatible with separate compilation. In this paper, we formalize a modular variant of defunctionalization which can support separate compilation. Our technique allows modules in a Haskell-like language to be separately defunctionalized and compiled, then linked together to generate an executable program. We provide a prototype implementation of our modular defunctionalization technique and we discuss the experiences of its application in a compiler from a large subset of Haskell to low-level C code, based on the intensional transformation.

1998 ACM Subject Classification D.1.1 Applicative (functional) programming; D.3.3 Language Constructs and Features: Abstract data types, Modules, Packages; F.3.3 Studies of Program Constructs: Functional constructs; D.3.4 Processors: Compilers.

Keywords and phrases Defunctionalization, functional programming, modules, separate compilation.

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.39

1 Introduction

Separate compilation allows programs to be organized in modules that can be compiled separately to produce object files, which the linker can later combine to produce the final executable. Modern compilers support separate compilation for many reasons. It saves development time by avoiding all the source code to be recompiled every time a change is made. Object files can be collected together in the form of libraries, which can be distributed as closed-source code. It is also used by build systems like `make` to tractably recompile big code bases [1].

Defunctionalization [15] is a technique which transforms higher-order programs to first-order programs. It does so by eliminating all closures of the source program, replacing them with simple data types and invocations of special first-order `apply` functions. It has been an important theoretical tool, e.g. used by Ager *et al.* to derive abstract machines and compilers from compositional interpreters [3, 2], but it has also been used as a compilation technique [8].

Defunctionalization has so far been presented as a whole-program transformation, a property that has been frequently cited as its major shortcoming, rendering it unsuitable as a realistic implementation approach for most compilers. Although defunctionalization is used in compilers that run in whole-program mode, such as MLton and UHC, so far it has not been used in compilers that support separate compilation to native code.

In the rest of this paper we give an introduction to defunctionalization and describe the problems that appear when we attempt to combine it with separate compilation. We then demonstrate how these problems can be overcome using *modular defunctionalization*, a variant that supports separate compilation of modules and linking. We give a formalization of our transformation and describe how it has been implemented in a compiler for a subset of Haskell. To our knowledge, this is the first time defunctionalization is implemented in a way that supports separate compilation to native code.

2 Defunctionalization

In this section we introduce the reader to the basics of defunctionalization, a program transformation that takes a higher-order program and produces an equivalent first-order program with additional data types representing function closures.

Assume that we have the following higher-order program written in Haskell:

```
result   = high (add 1) 1 + high inc 2
high g x = g x
inc z    = z + 1
add a b  = a + b
```

There are three higher-order expressions in this program:

1. `add 1` is a partial application of the `add` function yielding a closure of `add` that binds `a` to `1`; the closure has residual type `Int → Int`.
2. `inc` is the name of the `inc` function yielding a (trivial) closure that binds no variables and has residual type `Int → Int`.
3. `g` is a higher-order formal variable of type `Int → Int`.

Defunctionalization will then convert this program to an extensionally equivalent one, using only first-order functions. This is achieved by introducing a data type `Clo` for closures with one constructor for each different type of closure. In addition, a special function `apply` is introduced that recognizes these constructors and does function dispatch:

```
data Clo = Add Int | Inc

result   = high (Add 1) 1 + high Inc 2
high g x = apply g x
inc z    = z + 1
add a b  = a + b

apply c c0 = case c of
               Inc    → inc c0
               Add a0 → add a0 c0
```

Defunctionalization is a well-known technique, first introduced by Reynolds as an implementation technique for higher-order languages in an untyped setting [15]. For applying it to the simply-typed language that we study in this paper, we base our transformation on the type-safe variant of defunctionalization proposed by Bell, Bellegarde, and Hook, which creates different closure dispatching functions for different closure types [4]. For example, assume the following higher-order program:

```
result     = high1 (add 1) 1 1 + high2 inc 2
high1 h i j = h i j
high2 g x  = g x
inc z      = z + 1
add a b c  = a + b + c
```

The types of the closure constructors introduced would be `Int → Int → Int` for `Add1` and `Int → Int` for `Inc`. The example code is then defunctionalized to the following equivalent first-order program:

```
data CloI_I  = Inc | Add2 Int Int
data CloII_I = Add1 Int

result      = high1 (Add1 1) 1 1 + high2 Inc 2
high1 h i j = apply_II_I h i j
high2 g x   = apply_I_I g x
inc z       = z + 1
add a b c   = a + b + c

apply_I_I    clo1 m1    = case clo1 of
                            Inc        → inc m1
                            Add2 a1 b1 → add a1 b1 m1
apply_II_I   clo2 m2 n2 = case clo2 of
                            Add1 a2    → add a2 m2 n2
apply_II_I_I cloC mC    = case cloC of
                            Add1 aC    → Add2 aC mC
```

In this example, the constructors representing closures that can be applied to different types are dispatched by two different functions, `apply_I_I` and `apply_II_I`, that take closures belonging to data types `Clo_I_I` and `Clo_II_I`. We see that another closure constructor is also introduced, `Add2`, representing the closure of `add` binding two arguments. This can be the result of partially applying a closure `Add1` (i.e., `add` with one argument) to another argument, creating a new closure of `add` with two arguments. Partial application of `Add1` closures is done by function `apply_II_I_I`.

3 The Source and Target Languages

In this section we describe HL_M, a higher-order functional language with modules that will serve as the source language for modular defunctionalization. We also describe FL, its first-order subset that is the target language of our algorithm. Finally, we discuss how standard defunctionalization fails to separately transform HL_M modules.

3.1 The Source Language HL_M

The language HL_M is a Haskell-like higher-order functional language with modules [9]. A program in HL_M is organized in modules, each having a name, a list of data types and functions that are imported from other modules, a list of data type declarations, and a list of function definitions. HL_M is defined by the following abstract syntax, where μ ranges over module names, a ranges over data type names, b ranges over basic data types, x ranges over function parameters and pattern variables, op ranges over built-in constant operators, f ranges over top-level functions, and κ ranges over constructors:

p	$::=$	m^*	*program*
m	$::=$	$\texttt{module}\ \mu\ \texttt{where imports}\ I^*\ \delta^*\ d^*$	*module*
I	$::=$	$\mu\ (\mu.a)^*\ (v : \tau)^*$	*import*
δ	$::=$	$\texttt{data}\ \mu.a = (\mu.\kappa : \tau)^*$	*data type*
τ	$::=$	$b \mid \mu.a \mid \tau \to \tau$	*type*

d	$::=$	$\mu.f\ x^* \ =\ e$	*definition*
e	$::=$	$(x \mid v \mid op)\ e^* \mid \texttt{case}\ e\ \texttt{of}\ b^*$	*expression*
v	$::=$	$\mu.f \mid \mu.\kappa$	*top-level variable*
b	$::=$	$\mu.\kappa\ x^*\ \to\ e$	*case branch*

In HL_M we assume that type names (a), top-level function names (f) and constructor names (κ) are always qualified by the name of the module (μ) in which they are defined. Function parameters and pattern variables (x) are local names; they are not qualified. In this way, every module has its own *namespace*: every top-level function is distinct and two different modules can define functions, data types or constructors with the same name, without the danger of name clashes. In our presentation, we will follow Haskell's convention: all functions and variables start with a lowercase letter, while data types, constructors, and modules start with an uppercase letter.

An example program that is organized in two modules `Lib` and `Main` is Listing 1.

Listing 1 Example of a program organized in two modules.

```
module Lib where

Lib.high g x = g x
Lib.h y      = y + 1
Lib.test     = Lib.high Lib.h 1
Lib.add a b  = a + b


module Main where

import Lib ( Lib.h    :: Int→Int , Lib.high :: (Int→Int)→Int→Int
             Lib.test :: Int     , Lib.add  :: Int→Int→Int        )

Main.result = Main.f 10 + Lib.test ;
Main.f a    = a + Main.high (Lib.add 1) + Lib.high Main.dec 2
Main.high g = g 10
Main.dec x  = x - 1
```

3.2 The Target Language FL

The language FL is the first-order subset of HL_M, without modules. In other words, in programs written in FL:

1. All functions and data type constructors are first-order.
2. Module qualifiers are considered parts of the names of functions, data types and constructors.
3. All module boundaries have been eliminated; programs are lists of data type declarations and function definitions.

For the purpose of our presentation, FL is used as the target language of our defunctionalization transformation. In a real compiler, FL would be replaced by (or subsequently transformed to) native object code.

3.3 The Problem with Naïve Separate Defunctionalization

Let us go back to the two modules `Lib` and `Main` that were defined in §3.1. If we defunctionalize them separately, we obtain the two modules presented in Listing 2.

Listing 2 `Main` and `Lib` modules, defunctionalized independently.

```
module Lib where

data Lib.CloI_I  = Lib.H

Lib.high g x = Lib.apply_I_I g x
Lib.h y      = y + 1
Lib.test     = Lib.high Lib.H 1
Lib.add a b  = a + b

Lib.apply_I_I c z = case c of
                      Lib.H → h z


module Main where

import Lib ( Lib.h    :: Int→Int , Lib.high :: (Int→Int)→Int→Int
             Lib.test :: Int      , Lib.add  :: Int→Int→Int        )

data Main.CloI_I = Lib.Add Int | Main.Dec

Main.result = Main.f 10 + Lib.test ;
Main.f a    = a + Main.high (Lib.Add 1) + Lib.high Main.Dec 2
Main.high g = Main.apply_I_I g 10
Main.dec x  = x - 1

Main.apply_I_I c z = case c of
                       Lib.Add aC → Lib.add aC z
                       Main.dec   → Main.dec z
```

First of all, we see that different modules generate closure constructors that may populate the same closure type, here `Int` → `Int` → `Int`, but these constructors and their closure dispatching functions are spread over different modules. This problem is evident when the expression `Lib.high Main.Dec 2` is evaluated: `Lib.high` will call `Lib.apply_I_I`, which does not know the closure constructor `Main.Dec` and the program will terminate with an error.

We observe that closure types, closure constructors and closure dispatching functions must be treated specially, if functions from different modules are to exchange higher-order expressions. On the other hand, all other data types, constructors and functions can be safely compiled separately and coexist, since it is guaranteed that there are no name clashes.

4 Modular Defunctionalization

The solution to the problem described in the previous section is to have a proper way of managing the code that is generated by defunctionalization: closure types, constructors and their dispatchers must be collected together from all modules and code for them must only be generated at link-time. Our technique applies defunctionalization separately to each module, transforming to FL code, introducing closure constructors and invoking closure dispatchers whenever needed. It remembers the closure constructors that were required for each module and collects this information together with the target code generated for each module. Subsequently, in a final linking step, it generates code for the closure dispatchers based on the collected information.

Our modular defunctionalization is therefore a two-step transformation:

1. *Separate defunctionalization*: Each module is defunctionalized separately. This results to (i) a set of defunctionalized data type declarations; (ii) a set of defunctionalized top-level function definitions; and (iii) information about the closures that were used in this module. The third part serves as the *defunctionalization interface* of the module. At this point, the defunctionalized definitions from each module can be compiled separately to object code, assuming that closure constructors and dispatching functions are external symbols to be resolved later, at link-time.
2. *Linking*: The separately defunctionalized code is combined and the missing code is generated for closure constructors and dispatching functions, using the defunctionalization interfaces from the previous step. The missing code can then be compiled and linked with the rest of the already generated code, to produce the final program.

This section formally presents a module-aware variant of defunctionalization. The two steps mentioned above are described in the next two subsections.[1]

4.1 Separate Defunctionalization

This step defunctionalizes each module, generating a list of defunctionalized data type and function definitions, and a list of all closure constructors that are used in the transformed code. In the rest of this section, we describe how a single module m is defunctionalized.

The variant of defunctionalization presented here is type-driven (however, this is not essential for our technique, which can also be used for defunctionalizing untyped source languages). We therefore assume that type checking (and/or type inference) has already taken place and that all type information is readily available. To simplify presentation, we assume that expressions are annotated with their types (e.g., e^τ) but most of the times we will omit such annotations to facilitate the reader.

We also assume a mechanism for generating unique *names* during defunctionalization. All such names will be free of module qualifiers and suitable for use in FL. In particular:

- $\mathcal{N}(\mu.a)$, $\mathcal{N}(\mu.f)$, and $\mathcal{N}(\mu.\kappa)$ generate names for module-qualified types, top-level functions and constructors that appear in the source code of a module;
- $\mathcal{C}\ell(\tau)$ generates the name of a data type corresponding to closures of type τ;
- $\mathcal{C}(v, n)$ generates the name of a constructor corresponding to the closure of v, binding n arguments; and
- $\mathcal{A}(\tau, n)$ generates the name of the closure dispatching function for closures of type τ, supplying n arguments.

[1] A prototype implementation in Haskell of the technique described in this section is available at: `http://www.softlab.ntua.gr/~gfour/mdefunc/`.

A number of auxiliary functions for manipulating types will be useful:

- $\mathsf{arity}(\tau)$ returns the arity of a type (i.e., how many arguments must be supplied before a ground value is reached).

$$\begin{array}{lcl} \mathsf{arity}(b) & \doteq & 0 \\ \mathsf{arity}(\mu.a) & \doteq & 0 \\ \mathsf{arity}(\tau_1 \to \tau_2) & \doteq & 1 + \mathsf{arity}(\tau_2) \end{array}$$

- $\mathsf{ground}(\tau)$ converts higher-order types to ground types, by replacing function types with the corresponding closure types.

$$\begin{array}{lcl} \mathsf{ground}(b) & \doteq & b \\ \mathsf{ground}(\mu.a) & \doteq & \mathcal{N}(\mu.a) \\ \mathsf{ground}(\tau_1 \to \tau_2) & \doteq & \mathcal{C}\ell(\tau_1 \to \tau_2) \end{array}$$

- $\mathsf{lower}(\tau)$ converts higher-order types to first-order, by replacing the arguments of function types with the corresponding closure types, if necessary.

$$\begin{array}{lcl} \mathsf{lower}(b) & \doteq & b \\ \mathsf{lower}(\mu.a) & \doteq & \mathcal{N}(\mu.a) \\ \mathsf{lower}(\tau_1 \to \tau_2) & \doteq & \mathsf{ground}(\tau_1) \to \mathsf{lower}(\tau_2) \end{array}$$

The defunctionalization process is formalized using four transformations: $\mathcal{T}(\delta)$, $\mathcal{D}(d)$, $\mathcal{E}(e)$, $\mathcal{B}(b)$, for type declarations, top-level function definitions, expressions and case branches, respectively. They are defined as follows:

$$\begin{array}{lcl} \mathcal{T}(\texttt{data}\ \mu.a\ =\ \mu.\kappa_1 : \tau_1 \mid \ldots \mid \mu.\kappa_n : \tau_n) & \doteq & \texttt{data}\ \mathcal{N}(\mu.a)\ =\ \mathcal{N}(\mu.\kappa_1) : \mathsf{lower}(\tau_1) \\ & & \qquad\qquad\quad\ \mid\ \ldots \\ & & \qquad\qquad\quad\ \mid\ \mathcal{N}(\mu.\kappa_n) : \mathsf{lower}(\tau_n) \end{array}$$

$$\begin{array}{lcll} \mathcal{D}(\mu.f\ x_1 \ldots x_n\ =\ e) & \doteq & \mathcal{N}(f)\ x_1 \ldots x_n\ =\ \mathcal{E}(e) & \\ \mathcal{E}(x) & \doteq & x & \\ \mathcal{E}(x^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{A}(\tau, n)\ x\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n > 0 \\ \mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{N}(v)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n = \mathsf{arity}(\tau) \\ \mathcal{E}(v^\tau\ e_1\ \ldots\ e_n) & \doteq & \mathcal{C}(v, n)\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \text{if } n < \mathsf{arity}(\tau) \\ \mathcal{E}(op\ e_1\ \ldots\ e_n) & \doteq & op\ \mathcal{E}(e_1)\ \ldots\ \mathcal{E}(e_n) & \\ \mathcal{E}(\texttt{case}\ e\ \texttt{of}\ b_1\ ;\ \ldots\ ;\ b_n) & \doteq & \texttt{case}\ \mathcal{E}(e)\ \texttt{of}\ \mathcal{B}(b_1)\ ;\ \ldots\ ;\ \mathcal{B}(b_n) & \\ \mathcal{B}(\mu.\kappa\ x_1\ \ldots\ x_n\ \to\ e) & \doteq & \mathcal{N}(\mu.\kappa)\ x_1\ \ldots\ x_n\ \to\ \mathcal{E}(e) & \end{array}$$

In principle: (i) partial applications of top-level functions and constructors are replaced by closure constructors; (ii) functional parameters or pattern variables are applied by using the corresponding closure dispatching functions; (iii) data types are also defunctionalized: all higher-order types in the signatures of constructors are replaced by the corresponding closure data types.

During the first step of the transformation, useful information is collected for every closure corresponding to a top-level function or constructor. This is achieved with function $\mathcal{F}(v^\tau)$, defined as follows. We assume that v is a top-level function or constructor that is used in a closure and τ is its type.

$$\mathcal{F}(v^\tau)\ \doteq\ \mathsf{info}(v, \tau, [\,])$$

$$\begin{array}{lcll} \mathsf{info}(v, \tau, \tau^*) & \doteq & \{(\tau, \mathcal{N}(v), \tau^*)\} \cup \mathsf{info}(v, \tau_2, \tau^* +\!\!+ [\mathsf{ground}(\tau_1)]) & \text{if } \tau = \tau_1 \to \tau_2 \\ \mathsf{info}(v, \tau, \tau^*) & \doteq & \emptyset & \text{if } \tau \text{ is a ground type} \end{array}$$

Function $\mathcal{F}(v^\tau)$ returns a set of triples, one for each possible closure in which v can be used. Each triple contains: (i) the type of the closure; (ii) the name of v; (iii) the types of arguments contained in the closure. Notice that, for each triple, the number of arguments remaining to be supplied is equal to the arity of the closure's type. As an example, consider the function `add` from an earlier example:

```
add a b c = a + b + c
```

This function can be used in three closures, when 0, 1 and 2 arguments are supplied:

$$\begin{aligned}\mathcal{F}(\mathtt{add}^{\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}}) \;=\; \{\; &(\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}, \mathsf{add}, []),\\ &(\mathtt{Int}\to\mathtt{Int}\to\mathtt{Int}, \mathsf{add}, [\mathtt{Int}]),\\ &(\mathtt{Int}\to\mathtt{Int}, \mathsf{add}, [\mathtt{Int}, \mathtt{Int}]) \;\}\end{aligned}$$

It is possible that not all of the different closures generated by function $\mathcal{F}(v^\tau)$ will actually be used in the final program. The implementation is free to use a subset of these closures, e.g. taking just the ones that are generated in the code of the module. However, the final set of closures after linking is not just the union of those generated in the code of each linked module; more closures need to be automatically generated by the dispatching functions, in the case of partial application.

4.2 Linking

After separately defunctionalizing a number of modules, we are left with object code, i.e., defunctionalized definitions, and information about closures. To link the final executable program, we must merge all defunctionalized definitions and add the missing closure dispatching functions. Let I be the union of closure information from all modules to be linked.

As our presentation is at the source level, we start by generating data type definitions for closures; this would not be necessary if we were linking native code. For each closure type τ, we generate a definition for $\mathcal{C}\ell(\tau)$ as follows:

$$\mathtt{data}\; \mathcal{C}\ell(\tau) \;=\; \{\; \mathcal{C}(x,n) : \tau^* \to \mathcal{C}\ell(\tau) \mid (\tau, x, \tau^*) \in I \text{ and } n = \mathsf{arity}(\tau) \;\}$$

To generate the closure dispatching functions we use again the closure information I. As the program is closed at link-time, we only need to create closure dispatchers for all constructors in I. For every closure type τ, there may be two kinds of closure dispatchers. One is for the full application of such a closure, when all remaining arguments are supplied. However, if $n = \mathsf{arity}(\tau) > 1$, there are also $n - 1$ closure dispatchers corresponding to the partial application of such a closure, when only m arguments are supplied $(1 \leq m < n)$. The first kind of dispatchers returns ground values, whereas the second kind returns closures of smaller arity. Both kinds can be treated uniformly if we define $\mathcal{C}(x, 0) \doteq x$. The definition for $\mathcal{A}(\tau, m)$, where now $1 \leq m \leq n$, can be written as follows: a dispatcher for closures of type τ when m arguments are supplied.

$$\begin{aligned}\mathcal{A}(\tau,m)\; x_0\; x_1\; \ldots\; x_m \;=\; &\mathtt{case}\; x_0\; \mathtt{of}\\ &\{\; \mathcal{C}(x,n)\; y_1\; \ldots\; y_k \to \mathcal{C}(x, n-m)\; y_1\; \ldots\; y_k\; x_1\; \ldots\; x_m\\ &\;\mid (\tau, x, \tau^*) \in I \text{ and } n = \mathsf{arity}(\tau) \text{ and } k = |\tau^*| \;\}\end{aligned}$$

5 Modular Defunctionalization in a Haskell-to-C Compiler

Apart from a simple prototype implementation for a small subset of a Haskell-like language with modules, we have implemented this technique in GIC,[2] a compiler from a large subset of Haskell to low-level C that is based on the intensional transformation [11]. Defunctionalization is used in the front-end of the GIC compiler, transforming from Haskell to a first-order language with data types, which is subsequently processed by the intensional transformation [16, 17] to generate C code using lazy activation records [7].

As in our prototype implementation, defunctionalizing a Haskell module in GIC generates a set of function definitions. These can be transformed to C and then compiled to native code. The defunctionalized definitions contain references to external symbols corresponding to closure dispatching functions. Closure constructor information for each module is kept in a separate file, which describes the *defunctionalization interface* of the module.

This technique permits each module to be independently compiled to an object file. These files can be combined by the linker, which does the following:

- It builds the final closure constructor functions and closure dispatchers for all closures in the defunctionalization interfaces;
- It compiles the generated code of closure constructors and dispatching functions to a separate object file; and
- It calls the C linker to combine the compiled code of the modules and the compiled generated code of defunctionalization, in order to build the final executable.

Modular defunctionalization enables incremental software rebuilding for our Haskell subset. Moreover, it enables the building of shared libraries from defunctionalized Haskell code, provided that defunctionalization interfaces are distributed together with object files; such libraries can then be used by any third-party source code that has an appropriate linker.

6 Related Work

Pottier and Gauthier point out that defunctionalization can be modular for languages that are richer than our HL_M and support recursive multi-methods [14]. Our technique is simpler, as it only records closure constructor information for every module.

GRIN's front-end had some support for separate compilation, but the back-end was a whole-program compiler [5]. The Utrecht Haskell Compiler (UHC), which is also based on the GRIN approach, supports separate compilation for a special bytecode format that runs on an interpreter but not for native code [10]. In the context of the specialization transformation in UHC, Middelkoop pointed out that to fully support separate compilation in the presence of defunctionalization, some information should be kept that looks like the abstract syntax tree of a function [12]. We do the same by keeping only closure constructor type information, which is enough to generate the final abstract syntax tree of the required closure dispatchers.

A variant of defunctionalization that introduces no closure constructors nor dispatchers was proposed by Mitchell [13]. Consequently, it is not affected by modularity problems of generated code and is compatible with separate compilation. However, it cannot defunctionalize all higher-order programs, while our transformation is equally powerful with traditional defunctionalization.

[2] Available at `http://www.softlab.ntua.gr/~gfour/dftoic/`.

7 Conclusion

To the best of our knowledge, our approach is the first concrete implementation of the defunctionalization transformation that supports separate compilation to native code. We do so by defunctionalizing program modules separately while at the same time recording information about closure constructors. We then build and compile closure dispatchers for these constructors and for all program modules at link-time.

Our technique may lose opportunities of inter-module optimizations such as inlining, but loss of these optimizations is a general problem of separate compilation.

An open problem is how to combine our technique with polymorphism. There are more than one ways to implement polymorphism in a defunctionalizing compiler similar to ours, such as MLton's monomorphisation [6], UHC's type classes with dictionaries [10], or the techniques used in other defunctionalizing compilers [18, 19]. Each technique may interact differently with the modular defunctionalization presented here. Pottier and Gauthier's polymorphic defunctionalization [14] is another approach to implement polymorphism under defunctionalization; it requires guarded algebraic data types in the target language.

Acknowledgements Work partially supported by the research project "ΘΑΛΗΣ−ΕΜΠ: Handling uncertainty in data intensive applications on a distributed computing environment (cloud computing)" (MIS 380153), funded by the European Social Fund and the Greek national funds through the Operational Program "Education and Lifelong Learning".

References

1 Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Transactions on Software Engineering and Methodology*, 3(1):3–28, January 1994.

2 Mads Sig Ager, Dariusz Biernacki, Olivier. Danvy, and Jan. Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, March 2003.

3 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 8–19, New York, NY, USA, 2003. ACM.

4 Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 25–37, New York, NY, USA, 1997. ACM.

5 Urban Boquist and Thomas Johnsson. The GRIN project: A highly optimising back end for lazy functional languages. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages*, number 1268 in LNCS, pages 58–84, Berlin, Heidelberg, September 1996. Springer-Verlag.

6 Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. Flow-directed closure conversion for typed languages. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, pages 56–71, London, UK, 2000. Springer-Verlag.

7 Angelos Charalambidis, Athanasios Grivas, Nikolaos S. Papaspyrou, and Panos Rondogiannis. Efficient intensional implementation for lazy functional languages. *Mathematics in Computer Science*, 2(1):123–141, 2008.

8 Lasse R. Danvy, Olivier; Nielsen. Defunctionalization at work. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 162–

174, 2001. A more comprehensive version is available as Technical Report BRICS-RS-01-23, Department of Computer Science, University of Aarhus.

9 Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren. A formal specification of the Haskell 98 module system. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 17–28, New York, NY, USA, 2002. ACM.

10 Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the Utrecht Haskell compiler. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 93–104, New York, NY, USA, 2009. ACM.

11 Georgios Fourtounis, Nikolaos Papaspyrou, and Panos Rondogiannis. The generalized intensional transformation for implementing lazy functional languages. In *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages*, January 2013. In print.

12 Arie Middelkoop. Uniqueness Typing Refined. Master's thesis, Universiteit Utrecht, the Netherlands, 2006.

13 Neil Mitchell and Colin Runciman. Losing functions without gaining data: Another look at defunctionalisation. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 13–24, New York, NY, USA, 2009. ACM.

14 François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, 2006.

15 John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM Annual Conference*, volume 2, pages 717–740, New York, NY, USA, 1972. ACM. Reprinted in *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

16 P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *Journal of Functional Programming*, 7(1):73–101, January 1997.

17 P. Rondogiannis and W. W. Wadge. Higher-order functional languages and intensional logic. *Journal of Functional Programming*, 9(5):527–564, September 1999.

18 Andrew Tolmach. Combining closure conversion with closure analysis using algebraic types. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation*, 1997.

19 Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

Towards Automated Program Abstraction and Language Enrichment*

Sergej Chodarev, Emília Pietriková, and Ján Kollár

Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
`{Sergej.Chodarev,Emilia.Pietrikova,Jan.Kollar}@tuke.sk`

Abstract

This paper focuses on the presentation of a method for automated raise of programming language abstraction level. The base concept for the approach is a code pattern – recurring structure in program code. In contrast to design patterns it has a specific representation at a code level and thus can be parameterized and replaced by a new language element. In the article two algorithms for automated recognition of patterns in samples of programs are described and examined. The paper also presents an approach for language extension based on the found patterns. It is based on an interactive communication with the programming environment, where recognized patterns are suggested to a programmer and can be injected into the language in a form of new elements. Conducted experiments are evaluated in regard to the future perspective and contributions.

1998 ACM Subject Classification D.3.2 Language Classifications – Extensible languages, D.2.6 Programming Environments

Keywords and phrases Abstraction, code patterns, language extension, projectional editing

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.51

1 Introduction

One of the matters that make software development hard is the complexity. It is caused by the inherent complexity of the problems that developed systems need to solve and also by the need to comply to existing norms, interfaces and protocols [5]. With growth of software systems, expression complexity of their properties in a programming language mounts up as well. As the answer to complexity, higher levels of abstraction can be introduced. Abstraction allows expressing problems more simply by defining new, more abstract concepts that encapsulate complex expressions. This allows hiding the implementation details. Therefore, a promising solution for growth of program complexity can be an abstraction based on a language, allowing reduction of the complexity through defining new, more abstract concepts and language constructions.

Abstractions are usually organized into several levels, where each level is built on the abstractions provided by the level below it. This practice is called stratified [1] or layered design. Provided that lower levels are already in place, it is possible to concentrate on problem solution that can be expressed in high level terms relevant to the domain of solved problem.

* This work was supported by project VEGA 1/0341/13 *Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.*

Programming languages are also part of the abstraction level hierarchy. They provide a number of built-in abstractions that can be used to build programs. Moreover, they can also provide ways to define new abstractions. For example, it is possible to define new functions, data structures and classes. This allows distinguishing two ways of how new abstraction can be defined [21]:

1. *Linguistic* abstraction, with abstractions built into the language, what is typical for DSLs (Domain-Specific Languages),
2. *In-language* abstraction, with abstractions expressed by concepts available in the language, typical for GPLs (General-Purpose Languages).

Linguistic abstraction makes new abstract concepts part of the language itself. Definition of the concepts becomes a part of language translator or interpreter. Integration of abstractions into a language provides several advantages. First of all it provides a possibility to use the most appropriate notation. The use of appropriate linguistic abstraction instead of describing the same program using more general concepts can also provide additional semantic information about programmer intents for language processor, thus allowing optimizations to be performed without the need to reverse engineer the semantics. This is because no details irrelevant to the model need to be expressed, increasing conciseness and avoiding over-specification [21].

Compared to the linguistic, in-language abstraction constitutes achieving conciseness by providing facilities allowing users to define new (non-linguistic) abstractions in programs [21], including procedures or functions, and higher-order functions or monads. This way, abstraction can be provided but no declarativeness, in the sense of direct mapping of the program concepts to the model semantics. Thus, in-language abstraction is more flexible as users can build only those abstractions they actually need.

Linguistic abstraction is a basic element of Language-Oriented Programming [22, 6]. In this methodology, the first step of program design is a definition of high-level domain-specific language suitable for solving a specific problem[1]. Next, the program itself is implemented using the new language which is built upon the existing (less abstract) language. From this point of view, each level of abstraction is represented by a language, where each language is defined using a lower level language.

In this paper we propose new approach for the introduction of linguistic abstractions based on automated pattern recognition in program code. Section 2 provides a motivation for our work. In section 3 the proposal of programming environment supporting language enrichment is presented. Section 4 describes our experiments with two methods for pattern recognition. Finally section 5 concludes the paper and proposes ideas for future research.

2 Motivation

Let us consider two pieces of pseudo-code expressing transformation of the array values:

```
results = new Array();
for (int i = 0; i < data.size; i++){
  a = data[i];
  results[i] = f(a) * 5 + 3;
}
```

[1] The process of solving a problem by designing new language first is itself also called metalinguistic abstraction [2]

```
squares = new Array();
for (int i = 0; i < numbers.size; i++){
  b = numbers[i];
  squares[i] = pow(b, 2);
}
```

Both pieces of pseudo-code take all the values of arrays *data* and *numbers*, and transform them according to the appropriate calculations. The first pseudo-code uses function $f()$, multiplication and addition; the second pseudo-code uses power function. All the final values are then stored in arrays *results* and *squares* respectively.

As both pieces of pseudo-code are very similar, replacement of the repeated structures might be convenient. First, let us consider a new pseudo-code, applicable to both examples:

```
«output» = new Array();
for (int i = 0; i < «input».size; i++){
  x = «input»[i];
  «output»[i] = «op x»;
}
```

Where:
- `«input»` can be considered as a variable replaceable by arrays *data* and *numbers*;
- `«output»` represents a variable for arrays *results* and *squares*; and
- `«op x»` represents a variable for the calculations: $f(x) \times 5 + 3$ and x^2.

As it is possible to apply this new pseudo-code to both examples, it can be regarded as a *pattern*, which is repeated in the examples.

Abstraction has one simple goal in mind: To replace repeated code structures in order to increase expression abilities of the language. For the discussed examples, the identified pattern might be reduced and simplified with a new construct *map* (inspired by functional programming):

```
«output» = map («input», «op x»);
```

Where *map* can be considered as an abstraction to the identified pattern, representing the entire structure of the cycle with appropriate parameters. For the two examples, it is now possible to use new, more abstract pseudo-code (with notation backslash denoting an anonymous function):

```
results = map(data, (\x -> f(x) * 5 + 3));
squares = map(numbers, (\x -> pow(x, 2));
```

This approach enables the program code to be much shorter, thus less prone to errors.

Several implications arise according to the mentioned considerations:
- If it is possible to recognize language structures within a source code, then it is possible to identify recurring structures as well.
- If there is a large group of source code belonging to the same application domain, then it is possible to identify plenty of recurring structures within the domain.
- If frequently repeated structures are abstracted into the new ones, then it is feasible to form a new language dialect.
- If the new language structures are named by concepts of the appropriate application domain, then the resultant dialect is domain-specific.
- If a programmer is able to write short codes in concepts of the appropriate application domain instead of long codes in concepts of the general-purpose language, then his work might become much more effective.

- If the programming environment would provide help with definition of new abstractions, then there is a higher chance abstractions would be actually used.

Moreover, analysis of the current state within application of programming languages proved that along with system development in various application areas, there is a demand for the following language features [4, 14]:

- Increasing level of abstraction when expressing complex issues
- Increasing expression ability of a language, and thus effectiveness of its application
- Specialization of languages on specific domains of use
- Increasing flexibility when using a language in other domains

Considering importance of the abstraction concept in programming, there are a lot of open questions remaining, particularly regarding automatic analysis and introduction of abstraction. Therefore, in the following sections we will try to find answer (or more answers) to the following main question: *How can increase of abstraction be automated?*

Our approach to this task is based on resolving two basic problems concerning tool support for automated program abstraction and language enrichment:

1. Recognizing recurring patterns in program code.
2. Finding a way to inject identified patterns into a language as new constructs.

3 Proposal

We decided to base our approach to program abstraction on the concept of *patterns* – recurring structures in program codes. The conceptual scheme of the proposal can be seen in Fig. 1.

To propose a solution for automated introduction of new language abstractions based on patterns found in source code the problem of recurring pattern recognition should be resolved. Manual analysis of code may be a hard and tedious task. However, a tool for automatic pattern recognition can greatly help in this task.

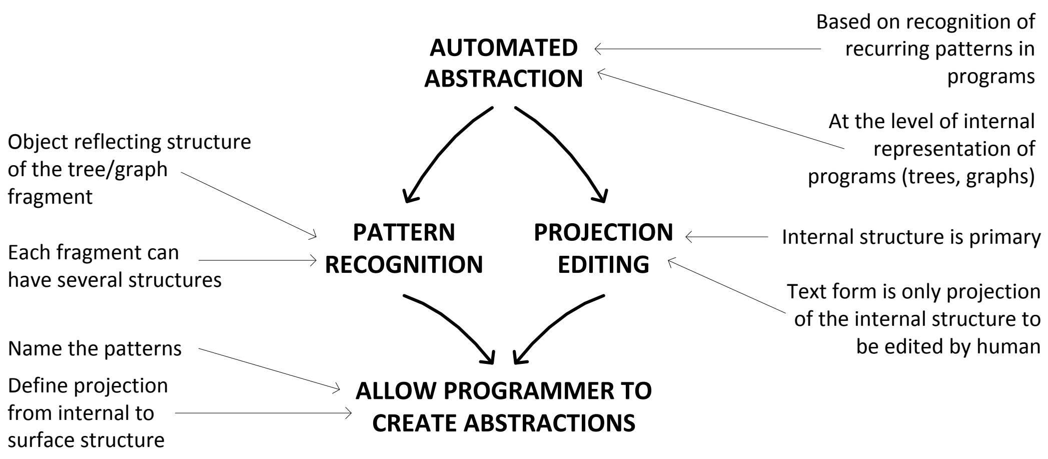

AUTOMATED ABSTRACTION
Based on recognition of recurring patterns in programs
At the level of internal representation of programs (trees, graphs)
Object reflecting structure of the tree/graph fragment
Each fragment can have several structures
PATTERN RECOGNITION
PROJECTION EDITING
Internal structure is primary
Text form is only projection of the internal structure to be edited by human
Name the patterns
Define projection from internal to surface structure
ALLOW PROGRAMMER TO CREATE ABSTRACTIONS

Figure 1 Proposal conceptual scheme.

3.1 Pattern Recognition

Pattern for our purposes is a recurring structure in program code. This structure can be expressed by fragment of a program with parts that may be different, replaced by *pattern variables*. This concept is different from *design patterns* [9] which describe recurring patterns and their usage on higher level. On the other hand, we are currently concentrating on patterns of a smaller scale.

Patterns would be recognized at the level of language abstract syntax. The abstract syntax tree or graph of a program contains all important information about structure of its code without syntactic details.

We have developed two approaches to this task:

1. Pattern recognition by comparing
2. Pattern recognition by collecting

Recognition by comparing is based on traversing trees representing programs from leaves up and comparing the subtrees to find groups of subtrees with the same structure.

The second approach that we propose is based on collection of abstracted structural schemas of program fragments. They can be obtained by replacing parts of the fragment by variables. Each fragment of code may be described by several structures of different level of detail. If such structural schemas are identified and stored with references to program fragments that contain them, it is possible to analyze frequency of their usage and by this way identify possible pattern.

3.2 Language Enrichment

Since recognized patterns represent recurring structures found in programs, they also present potential extensions of the language. To inject a pattern into the language there is a need to name it and define its syntactic or surface structure that would be used to represent it in program code.

Enrichment of language syntax and semantics directly by its users, though, is rarely allowed. In most cases it requires modification of the original language implementation, since a composition of the language with new elements is needed. Doing so with traditional textual language processed using a parser usually generated by some parser generator according to the grammar specification, it may result in several problems caused by possible ambiguity of the resulting grammar. Partially, this is caused by the fact, that grammar subclasses supported by common parser generators are not closed under composition [11]. Moreover, to allow composition and extension of a language without modification of the original implementation, it requires special tool support [7] and knowledge in the field of language development.

A possible way to solve the problem of language composition is a transition to *concept composition*. This means that instead of composing languages and their grammar rules, only concepts in a single base language are composed. This requires lowering the role of language grammar and is possible to be achieved at least in two ways:

1. Using single syntax for composed languages.
2. Using projectional editing.

In our case, the first way means not to use special syntax for injected patterns. Patterns would only be named and one of the shapes predefined in the language syntax would be assigned to them. This is similar to the definition of a function – it is assigned a name and standard syntax of function call or operator application. Another example is extension of Lisp and its dialects using macros that are based on the uniform syntax of S-expressions [10],

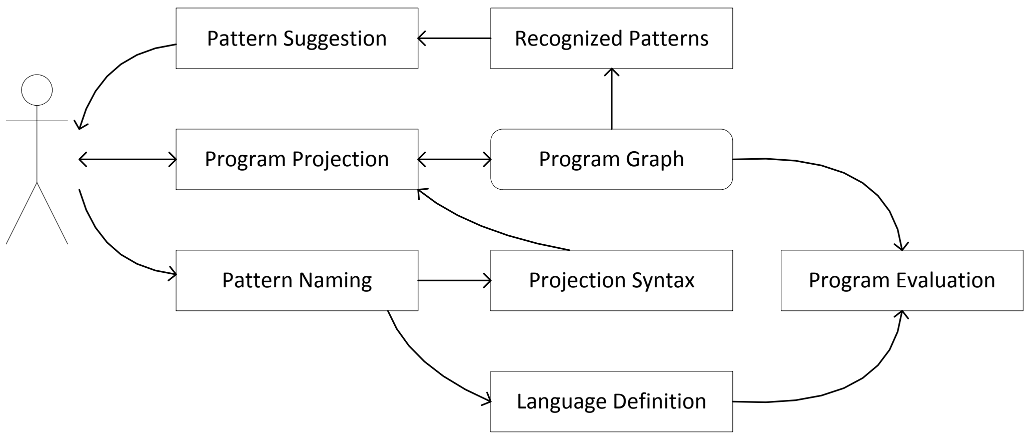


Figure 2 Architecture of language enrichment environment.

or definition of XML based language. Disadvantage of such an approach is, indeed, low flexibility of choosing an appropriate notation.

On the other hand, projectional editing keeps different notations for languages on the surface, while using unified representation internally. The syntax becomes only a matter of projection and actual information of a program (code) is stored in some different form, not visible for language user.

Projectional editing is used by some language workbenches, for example JetBrains MPS [6] or Intentional Workbench [19]. They use internal graph-based representation as main form of a program. Editable form is only a projection of internal form [8]. When a user is issuing editing commands at the projection, the internal structure is modified and the projection is updated accordingly. This allows different types of editable representation in addition to the textual, for example graphical or table-based.

In this way, elements of languages developed using a language workbench are actually only concepts of an internal representation language (which is usually not textual). This means that in this case the composition of languages corresponds to the composition of concepts inside a single language. Textual composition is only its projection which is not required to be unambiguous.

3.3 Concept of Language Enrichment Environment

Based on the considerations described above, it is possible to construct an environment for automated language enrichment based on patterns found in programs. The principles of its functionality are depicted in Fig. 2. The primary representation of the program is its abstract syntactic graph (ASG) editable through the projection. Syntax specification is used as a basis for projection and editing environment.

While program is created by a programmer, the structure of its elements is collected inside the environment to recognize the recurring patterns. Then, the environment should use the collection to suggest recognized patterns to programmer. Patterns can have various uses beside the automated abstraction. For instance, they can provide automatic completion of snippets for frequently used constructs.

If a programmer decides the pattern is semantically significant, he can name the pattern and therefore enrich the language he uses. He needs to provide specific concrete syntax for the named pattern that would be appropriate for conveying its meaning. A new syntax rule would be added to the language syntax definition and used by the projectional editor.

At the same time, named pattern becomes a part of the language structure and semantics definition. By default it expands the pattern during program evaluation or translation. Definition though can be amended by some optimization rules based on semantics of the pattern.

Introduction of new abstractions into the language can also have a negative effect. The more new constructs specific to a program are added into the language, the more is a programmer forced to learn new abstractions. Moreover, if more than one programmer is working with the same program code, and only few of them know the new abstractions, a problem may occur as they may not understand program codes of each other. This situation may occur already in the current abstraction range of various general-purpose and domain-specific languages.

Considering this, it would be appropriate if the environment would also provide reverse mode of work. The user should be allowed to switch the level of abstraction used in the code and display it in expanded form. That is, our experiments and algorithms for pattern recognition should result in two instruments available to user of the programming environment:

- Pattern contraction – pattern replacement by new syntactic element
- Pattern expansion – syntactic element replacement by its implementation through elements of lower level abstraction

For instance, if one programmer knows list comprehension construct, the other one, who does not, should by able to specify directly within the environment that he does not want to use list comprehension, or that he wants to learn their structure. Then, any list comprehension would be equivalently substituted, e.g. through map or filter function. This expansion can be even applied repeatedly as it is shown in Listing 1. On the contrary, a programmer may also use several complex structures, and then replace them through the known patterns by equivalent, shorter structures, and thus noticeably reduce the program code length.

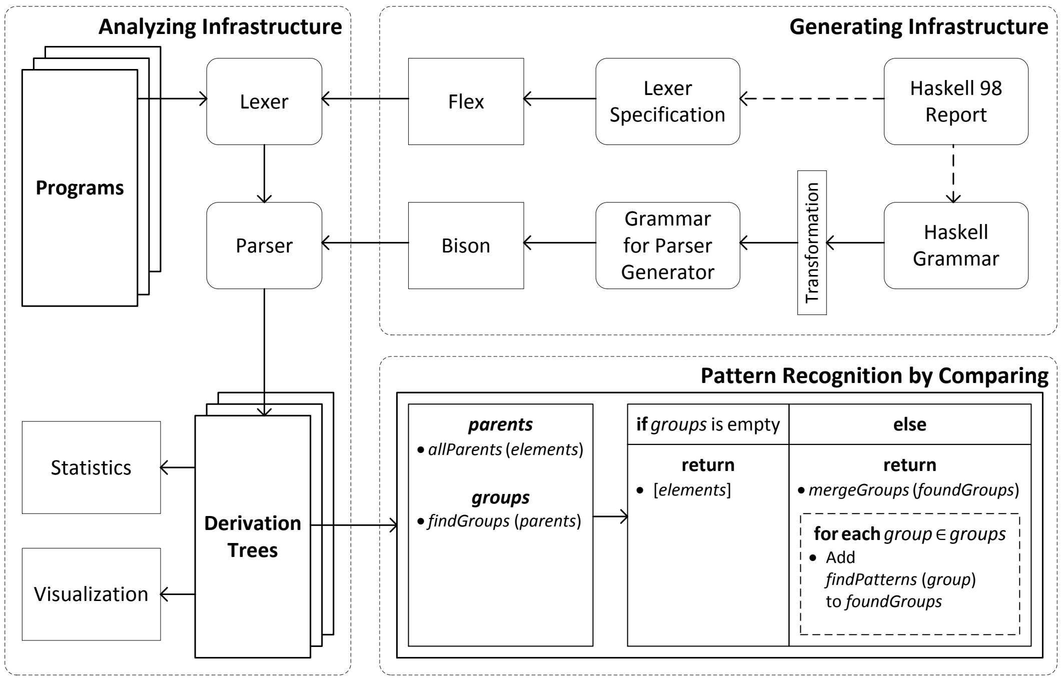
Listing 1 Example of language element expansion.

```
squares = [x ^ 2 | x <- xs]
   ↓
squares = map (\x -> x ^ 2) xs
   ↓
squares [] = []
squares (x:xs) = (x ^ 2) : squares xs
   ↓
squares xs | null xs = []
           | otherwise = (head x ^ 2) : squares (tail xs)
```

4 Experiments

For pattern recognition, we suggest two different methods: by comparing and by collecting. For experimental purposes, the first method has been performed and examined on a large group of Haskell programs while the second method has been performed on a simple language of functions and expressions.

Pattern recognition by comparing is also a successor to another research, as it uses results and implementations acquired by experiments associated with effects of abstraction [16].



🟨 **Figure 3** Architecture of Haskell syntax analyzing tools, including pattern recognition by comparing.

On the other hand, pattern recognition by collecting is based on the evaluation of the first method, reducing unnecessary implementations and extending possibilities of the pattern recognition.

4.1 Pattern Recognition by Comparing

Pattern recognition method based on comparing program fragments was developed on top of the set of tools gathering information from Haskell programs to get a proper knowledge about the used constructs in analyzed programs. As a result of the program analysis, derivation tree is produced, consisting of the used rules of Haskell grammar [15]. The architecture consists of three parts – *generating infrastructure*, *analyzing infrastructure*, and *pattern recognition* (see Fig. 3).

The goal of the generating infrastructure is to prepare lexer and parser, which are used within the analyzing infrastructure, and are developed using generative methods. Haskell grammar specification is analyzed and transformed into a form suitable as an input for lexer and parser generators. The analyzing infrastructure contains lexer and parser of Haskell programs, intended for analysis of Haskell into lexical units and then processing them into derivation trees. Derivation trees are produced in XML format and are further processed to retrieve statistical data on Haskell programs and to recognize common language patterns.

The third part of the architecture, which builds on the generating and analyzing infrastructures, is dedicated to pattern recognition by comparing. It is based on the principle of comparing different fragments of a program to find groups of similar ones. The use of derivation trees generated by the analyzing infrastructure is indeed not obligatory for the proposed algorithm. Nevertheless, generating and analyzing parts of the Haskell syntax analysis tools have already been part of other range of experiments devoted to effects of abstraction in programming languages, also published and more particularly described in [16].

To recognize syntactic patterns in a program or a set of programs, it is important to decide which parts of the analyzed programs may be considered similar. The simplest possibility is to consider only the equal trees. However, this approach is exceedingly limiting. Trees can be considered similar if their structure is the same except for the attributes of terminal symbols (approach that has been chosen in this experiment).

Another approach is to allow differences in whole subtrees rooted in the same type node. This should allow more complex syntactic variables and is, however, more difficult to implement using specified approach, as it requires comparisons of program fragments on different levels of the tree.

To find patterns in the program derivation tree, a simple algorithm is used, based on the function *findPatterns* defined in Listing 2 (see also Fig. 3).

Listing 2 Pseudo-code algorithm for pattern recognition by comparing.

```
parents ← allParents(elements)
groups ← findGroups(parents)

if groups is empty then
  return [elements]
else
  for all group ∈ groups do
    Add findPatterns(group) to foundGroups
  end for
  return mergeGroups(foundGroups)
end if
```

Function *findPatterns* takes a list of the tree elements and recursively examines their parents to find a set of groups of subtrees that have a similar structure. It uses helper functions where *allParents* returns a set of parents of all tree elements in a group and *mergeGroups* merges the list of group lists into a single list. An important function is *findGroups*. Given a set of tree elements, it returns list of groups of elements with similar subtrees.

To initiate the algorithm, the *findPatterns* function is called on terminal symbols of the tree. Then it tries to walk up to the root of the tree while it can find groups of subtrees with similar structure. List of subtree groups is a result of the algorithm, where each group corresponds to a found pattern and contains all occurrences of the pattern.

Let us look at a simple example program in Listing 3 defining functions *insert* and *delete* manipulating binary search trees. Derivation tree of this program is represented in Fig. 4.

Using the described method, it is possible to find several recurring patterns in this program (see Fig. 4). The most important are:

- `| x` α `y = Bin y t1 (` β `x t2 )`
- `Bin y (` α `x t1 ) t2`
- α `x ( Bin y t1 t2 )`
- α `x Nil`

Greek letters in the patterns regard the syntactic variables that can be replaced with concrete syntactic elements. Other identified patterns are too small to be mentioned.

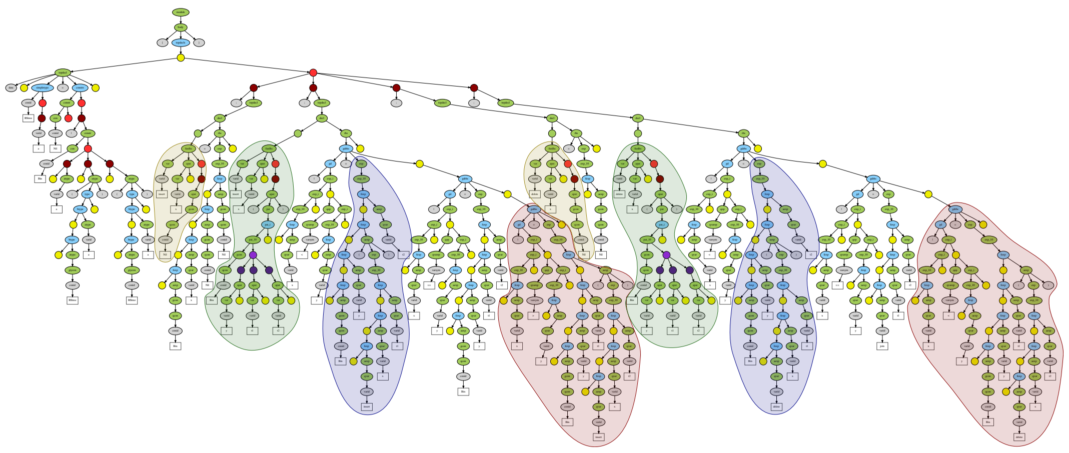


Figure 4 Example of program derivation tree with recognized patterns.

Listing 3 Example program code for pattern recognition by comparing.

```
data BStree a = Nil
    | Bin a (BStree a) (BStree a)

insert x Nil = Bin x Nil Nil
insert x (Bin y t1 t2)
    | x < y  = Bin y (insert x t1) t2
    | x == y = Bin y t1 t2
    | x > y  = Bin y t1 (insert x t2)

delete x Nil = Nil
delete x (Bin y t1 t2)
    | x < y  = Bin y (delete x t1) t2
    | x == y = join t1 t2
    | x < y  = Bin y t1 (delete x t2)
```

4.2 Pattern Recognition by Collecting

Another experiment has been performed to evaluate different algorithm for pattern recognition. It is based on collecting potential patterns found in a processed program. This allows evaluation of their frequency in a program and selection of patterns that may be interesting for a programmer. Basically, compared to the first method (by comparing), it is unique in its ability to recognize new patterns which differ from each other in their subtrees, not only leaves as it is in the first method. As opposed to the previous experiment, the input consists of abstract syntax trees and not derivation trees.

The main idea of the proposed algorithm is to avoid direct comparison of program fragments with each other. Instead, structure of a fragment should be described using *structural schema*. This is a data structure that reflects structure of program fragment with some details omitted. It is obvious that a fragment of program can correspond to several structural schemas which describe different parts of it and with different levels of detail.

In our case structural schema is implemented as modified abstract syntax tree with some nodes or leaves replaced by variables. Variables can match different subtrees allowing coverage of program fragments that differ whole subexpressions. Structural schemas are

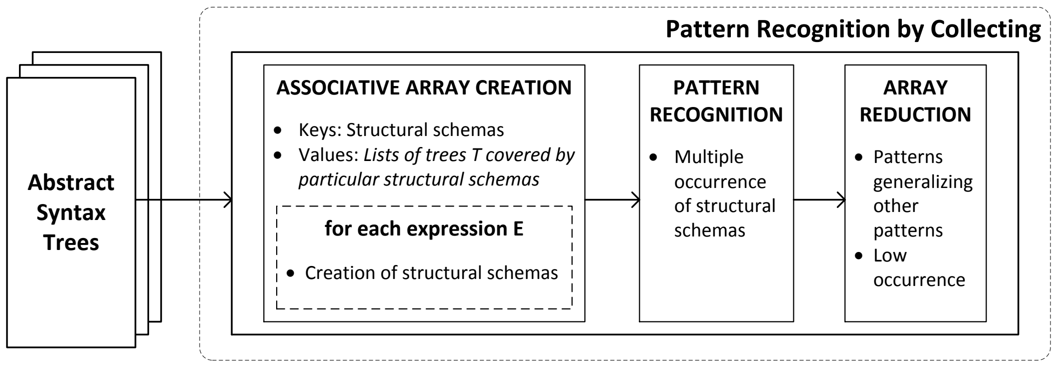

Pattern Recognition by Collecting
Abstract Syntax Trees
ASSOCIATIVE ARRAY CREATION
Keys: Structural schemas
Values: Lists of trees T covered by particular structural schemas
for each expression E
Creation of structural schemas
PATTERN RECOGNITION
Multiple occurrence of structural schemas
ARRAY REDUCTION
Patterns generalizing other patterns
Low occurrence

Figure 5 Architecture of pattern recognition by collecting tools.

actually potential patterns and therefore they have the same structure.

Fig. 5 depicts an algorithm for pattern recognition by collecting that was performed within the experiment. The input is a sequence of abstract syntax trees representing expressions of a program or collection of programs. In the first step structural schemas are derived from each expression tree. The process is outlined in Listing 4.

Listing 4 Pseudo-code algorithm for pattern recognition by collecting.

```
patterns ← empty associative array
for all expression ∈ expressions do
  subexpressions ← allSubtrees(expression)
  for all subexpression ∈ subexpressions do
    schemas ← structuralSchemas(subexpression)
    for all schema ∈ schemas do
      Add subexpression to patterns[schema]
    end for
  end for
end for
return patterns
```

The key part is the generation of structural schemas based on the fragment of program syntax tree (represented by the function *structuralSchemas* in the listing). Generated schemas actually represent possible modifications of a particular tree. By modification, we mean substitutions of the tree leaves or nodes by variables. In different schemas different combinations of nodes would be substituted.

As the result, an associative array is created of which the keys are structural schemas and values are lists of trees or subtrees covered by a particular structural schema. Within the associative array it is possible to determine multiple occurrences of the structural schemas. These schemas are important as they can be considered as patterns. Moreover, it is possible to reduce the associative array by those schemas that represent generalization of other schemas, without higher frequency.

Disadvantage of the algorithm is that it is not possible to generate and store all possible structural schemas for larger program fragments. This limitation can by overcame using different approaches:

1. Collecting only schemas for small program expressions.
2. Limit the depth of subtrees that are taken into account while generating structural schemas. All details below the specified threshold would be always replaced by variables.

While the first approach would limit possibly recognized patterns significantly, the second one may still be able to recognize a lot of useful patterns.

This algorithm has been evaluated within an experiment based on a simple language of functions and expressions. To simplify the development and make the relation between internal structure and concrete syntax more direct, S-expressions were used in the experiment.

For instance, if the code in Listing 5 is used as an input for the algorithm, it successfully recognizes a pattern in Listing 6.

Listing 5 Example program code for pattern recognition by collecting.

```
(def squares xs
    (if (= xs nil) nil
                   (cons (* (head xs) (head xs))
                         (squares (tail xs)))))
(def withTwo xs
    (if (= xs nil) nil
                   (cons (+ (head xs) 2)
                         (withTwo (tail xs)))))
(def op xs
    (if (= xs nil) nil
                   (cons (- (* (head xs) 2) 2)
                         (op (tail xs)))))
(def positives xs
    (if (= xs nil) nil
                   (cons (>= (head xs) 0)
                         (positives (tail xs)))))
```

Listing 6 Pattern found in example code (variables marked with greek letters).

```
(def α xs (if (= xs nil) nil (cons β γ)))
```

5 Conclusion and Future Work

In this article, we have proposed a solution for automated introduction of new language abstractions based on patterns that in this study are understood as recurring structures in program code.

As part of the solution, two different approaches were experimentally developed to recognize language patterns: pattern recognition by comparing and by collecting. The first approach is based on comparing program fragments based on derivation trees of the applied Haskell grammar rules, generated by a complex set of analyzing tools [16]. Its principle lies in traversing particular derivation trees and recognizing the highest possible subtrees of the same structure.

While implementation of pattern recognition by comparing is relatively simple, and it is able to recognize most of the common recurring program structures, it does not allow substitution of particular subtrees by variables. That is, it cannot recognize some specific patterns. For instance, if we considered example code from Listing 5, the algorithm would only recognize the following pattern:

```
(def α xs (if (= xs nil) nil (cons (β (head xs) γ) (α (tail xs)))))
```

However, this covers only two of the four structurally similar program fragments. More general pattern (mentioned in Listing 6) was supported and recognized only by the second technique, which is pattern recognition by collecting. It reduced some drawbacks of the previous method, thus allowing substitution of subtrees by variables. On the other hand, the number of potential patterns inspected for each subtree needs to by limited since it is not possible to collect all of them for every size of a subtree.

Unlike in the previous experiment, the input of this algorithm consists of abstract syntax trees (not derivation trees) and its main principle lies in sequential generation of an associative array, with the keys of structural schemas and values of collected subtree lists corresponding to particular structural schemas.

The approach to pattern recognition by collecting is also more suitable for interactive use as a part of programming environment, because it allows incremental addition of program expressions into the pattern recognition process.

These experiments are significant to further part of the proposal focused on the concept of pattern based language enrichment using projectional editing. They show that it is possible to automatically find structural patterns in program code. To make more significant conclusions, it is necessary to perform experiments on greater set of programs and to further develop the algorithms. Development of the language enrichment environment is also needed to fully evaluate the proposal.

Further research in this area can be focused on the possibility to recognize patterns on higher level than the structure of code. These patterns may be scattered in the program code but semantically interconnected. Therefore, the pattern recognition process needs to have a high degree of knowledge about program semantics.

The contribution of the presented proposal for language enrichment is the new approach to the extension of programming language based on the needs of programmers [20]. It tries to combine the advantages of both linguistic and in-language abstractions, allowing language users to define new abstractions that are integrated into the language. In addition, the process of language enrichment is aided by automated patterns recognition.

However, upon the presented results, the most significant is the contribution to automated software evolution. Clearly, this would mean to shift from a language analysis to language abstraction, associating concepts to formal language constructs [17], and formalizing them by means of these associations. In this way, we expect to integrate programming and modeling, associating general purpose and domain-specific languages [18], [13], as well as to perform a qualitative move from an automatic roundtrip engineering [3], [12] to the automated roundtrip software evolution.

References

1 Harold Abelson and Gerald J Sussman. Lisp: A language for stratified design. Technical report, Cambridge, MA, USA, 1987.

2 Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Electrical Engineering and Computer Science. The MIT Press, second edition, 1996.

3 Uwe Aßmann. Automatic roundtrip engineering. *Electronic Notes in Theoretical Computer Science*, 82(5):33–41, 2003.

4 D. Astapov. Using haskell with the support of business-critical information systems. *Practice of Functional Programming (in Russian)*, 2, 2009.

5 Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, april 1987.

6 Sergey Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2), November 2004. Available at `http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf`.

7 Sebastian Erdweg, Paolo G. Giarrusso, and Tillmann Rendel. Language composition untangled. In *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA)*, 2012. to appear.

8 Martin Fowler. Language workbenches: The killer-app for domain specific languages? 2005. Available at `http://martinfowler.com/articles/languageWorkbench.html`.

9 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

10 Paul Graham. *On Lisp.* Prentice Hall, 1994.

11 Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of Onward! 2010.* ACM, 2010.

12 Carsten Lohmann, Joel Greenyer, and Juanjuan Jiang. Applying triple graph grammars for pattern-based workflow model transformations. *Journal of Object Technology*, 6(9):253–273, 2007.

13 Ivan Luković, Pavle Mogin, Jelena Pavićević, and Sonja Ristić. An approach to developing complex database schemas using form types. *Software – Practice & Experience*, 37(15):1621–1656, December 2007.

14 Alex Ott. Using scheme in the development of "dozor-jet" family of products. *Practice of Functional Programming (in Russian)*, 2, 2009.

15 Simon Peyton Jones. Haskell 98 language and libraries – the revised report. Technical report, Cambridge England, 2003.

16 Emília Pietriková, Ľubomír Wassermann, Sergej Chodarev, and Ján Kollár. The effect of abstraction in programming languages. *Journal of Computer Science and Control Systems*, 4(1):137–142, 2011.

17 Jaroslav Porubän and Peter Václavík. Extensible language independent source code refactoring. In *AEI '2008: International Conference on Applied Electrical Engineering and Informatics*, pages 58–63, 2008.

18 Miroslav Sabo and Jaroslav Porubän. Preserving design patterns using source code annotations. *Journal of Computer Science and Control Systems*, 2(1):53–56, 2009.

19 Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 451–464, New York, NY, USA, 2006. ACM.

20 Guy L. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999.

21 Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages.* CreateSpace Independent Publishing Platform, 2013.

22 Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

Part III

XML and Applications

Publishing Linked Data with DaPress*

Teresa Costa[1] and José Paulo Leal[2]

1 **CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto Porto, Portugal**
`up200101764@alunos.dcc.fc.up.pt`

2 **CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto Porto, Portugal**
`zp@dcc.fc.up.pt`

Abstract

The central idea of the Web of Data is to interlink the information available in the Web, most of which is actually stored in databases rather than in static HTML pages. Tools to convert relational data into semantic web formats and publish then as linked data are essential to fulfill the vision of a web of data available for automatic processing, as web content is currently available to humans. This paper presents DaPress, a simple tool to publish linked data on the Web, that maps a relational database to an RDF triplestore and creates a SPARQL access point. The paper reports the use of DaPress to publish the database of Authenticus, a system that automatically assigns publication authors to known Portuguese researchers and institutions.

1998 ACM Subject Classification H. Information Systems; H.2 Database Management; H.2.5 Heterogeneous Databases;

Keywords and phrases RDF, RDF Schema, Relational data; Semantic web

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.67

1 Introduction

The World Wide Web has deeply changed the way information is produced, published and consumed. Nowadays it is trivial to produce a document in HTML format, publish it on a HTTP server and virtually anyone, anywhere on the planet, can access it using a web browser and benefit from its content. Anyone but not anything.

Information on the web is produced and formatted for humans. It is simple for a person to understand web content and navigate trough hyperlinks with a meaningful purpose. However, building a software agent that gathers information from the web for a fairly simple task, such as setting an appointment with a doctor or planning a business trip, is still a challenge after more than a decade of research.

The goal of the semantic web is to open the vast amount of data available on the web to software processing. The first attempt was to markup with semantic annotations the content already available on web pages. The use of XML languages and the separation of content from formatting was expected to contribute to that goal. However, the forces that shape the evolution of the web clearly favor graphical user interaction over semantic content. Hence, nowadays is harder to provide semantic annotations to web apps and web services than it was to last century hand-made web pages.

* This work is in part funded by the ERDF/COMPETE Programme and by FCT within the FCOMP-01-0124-FEDER-022701 project.

Fortunately, most web content is actually generated from databases. Thus, rather than to extract information from web pages it is more effective to collect it directly from raw data sources. The *linked data* initiative promotes best practises for publishing the data that supports web content. This data should be published in open formats so that it can be read and processed by any software. Moreover data from different sources should be interlinked to create a global web of content.

Navigation on the world wide web relies on content being linked using *Uniform Resource Locations* (URLs). Linked data follows a similar approach to enable software agents to navigate trough data available from different sources. If URLs are used as identifiers the content of a database may refer the content of another.

Interoperability has been a concern in databases for a long time. Any relational database management system imports and exports data in open formats, such as XML or comma-separated values (CSV), and relational databases themselves are based on open standards, such as the Structured Query Language (SQL). Unfortunately these open standards are not enough to build a web of linked data and must be complemented with semantic web technologies for a number of reasons.

Firstly, the structure of a relational database is rigid. The software that processes a relational data is designed and implemented for a particular database schema, and needs to be updated to reflect changes in that schema. A program to process a generic relational database, independently of its schema, would be too hard to implement. In contrast, the *Resource Description Framework* (RDF) data has a simple and uniform structure – a collection of triples – and the schemata from the various databases are recorded also as RDF data using a special vocabulary – RDF Schema.

Secondly, the semantics of the data stored in relational database is not explicit. An application that processes relational data relies on an implicit knowledge of the meaning of the data, and linking related data from different sources is a difficult task. To a human it may be obvious that the tables named "teacher" and "docent" from two different academic databases contains similar data, but that kind of reasoning is extremely difficult to automatise. The semantic web provides ontologies to describe a domain of data shared by different databases.

Lastly, a typical relational database contains both data that should be published mixed with sensitive or irrelevant data that should not be published. Also, publishable data may need to be preprocessed to normalize either its content or its structure. An approach to achieve it is to map relational databases into RDF data and web ontologies, while providing absolute control of this process to the data owner.

The motivation for the ongoing research presented in this paper is the development of a simple and flexible approach to publish the content of relational databases as linked data on the Web. The corner stone of the proposed approach is a system called DaPress that maps relational databases to RDF and RDF Schema based on a XML configuration file. Relational data from the source database is periodically loaded into the DaPress triplestores, which is accessible trough a SPARQL access point.

The remainder of this paper is organized as follows. Section 2 summarizes the concepts, languages and tools related to linked data. Section 3 presents the design and implementation details of DaPress, the proposed linked data publishing system. This approach was validated on the database of Authenticus, a system that automatically assigns publication authors to known researchers and institutions and the results are reported on Section 4. Section 5 concludes with a summary of the work presented in this paper and points to future developments of DaPress.

2 Linked Data

Linked Data is a methodology for publishing structured data based on two fundamental Web technologies: Uniform Resource Identifiers (URIs) and the HyperText Transfer Protocol (HTTP). The term *Linked Data* highlights the fact that this methodology establishes links among data from different sources, creating a *web of data*. Unsurprisingly, the concept of linked data is due to Tim Berners-Lee, the father of the World Wide Web, who introduced a set of basic rules for publishing data on the Web [4], namely:

- *Use URIs as names for things;*
- Use HTTP URIs so that people can look up those names;
- *When someone looks up a URI, provide useful information, using standards (RDF, SPARQL);*
- Include links of other URIs so that they can discover more things.

The resources, or "things" as Tim Berners-Lee calls them, are identified by URIs and these entities can be looked up simply by dereferencing the URI over the HTTP protocol. The HTTP protocol provides a simple and universal mechanism for retrieving resources or retrieving descriptions of entities.

By using HTTP URIs (or URLs) to identify entities, the HTTP protocol as retrieval mechanism and RDF data model to represent data, Linked Data builds on the general architecture of the Web.

The remainder of this section details the fundamental technologies used by Linked Data, such as RDF and RDF Schema, as well as related systems to publish relational data as RDF.

2.1 Resource Description Framework

The Resource Description Framework (RDF) [2, 10] is a framework for representing any kind of information available in the web. The RDF data model provides an abstract, conceptual framework for defining and using metadata, that has a graph-based data model, and is easy to process and manipulate by applications. It provides interoperability between applications that exchange machine-understandable information on the Web. Data in RDF format can be persistently stored in specialized repositories called triplestores, and retrieved using specialized RDF query languages, such as SPARQL. The interoperability of RDF data is supported by several serialization formats, both text and XML based.

2.1.1 Data Model

The basic element in RDF is a *statement*, a simple sentence with three parts – subject, predicate and object – expressing a relationship between things. The *subject* is a resource, the thing to describe, identified by an URI. The properties are a special kind of resource that describe relations between resources. A *property* specifies an aspect, characteristic, attribute or relation used to describe the resource. They are also identified by URIs. A specific resource together with a named property needs an *object*, in order to construct a statement. The object can be either a resource or an atomic value, named *literal*. Being composed of tree parts, RDF statements are also known as *triples*. A collection of triples forms a graph where the set nodes is given by subjects and objects of triples, and the arcs that connect them are given by predicates.

Table 1 Simple example of table-based tripes representation.

Subject	Predicate	Object
.../Person/QuentinTarantino	.../name	"Quentin Tarantino"
.../Movie/PulpFiction	.../name	"Pulp Fiction"
.../Person/QuentinTarantino	.../director	.../Movie/PulpFiction

Lets consider a simple example to show two different ways of represent a statement.

Quentin Tarantino is the director of Pulp Fiction.

Table 1 shows the triples extracted from previous phrase where ellipsis replace a common URL prefix for sake of terseness. Note that the concepts "Quentin Tarantino" and "Pulp Fiction" where replaced by URIs, as was the "is the director" property. By the cultural context, it is known that Quentin Tarantino is a person's name and Pulp Fiction a movie title. Using that information the other two statements assign a textual representation to both the subject and object of the previous sentence.

Figure 1 is a graph-based and equivalent representation of the same three statements. It is a directed graph, with labeled nodes and arcs. The arcs are directed from the resource (the subject) to the value (the object). This kind of graph is known as a semantic net.

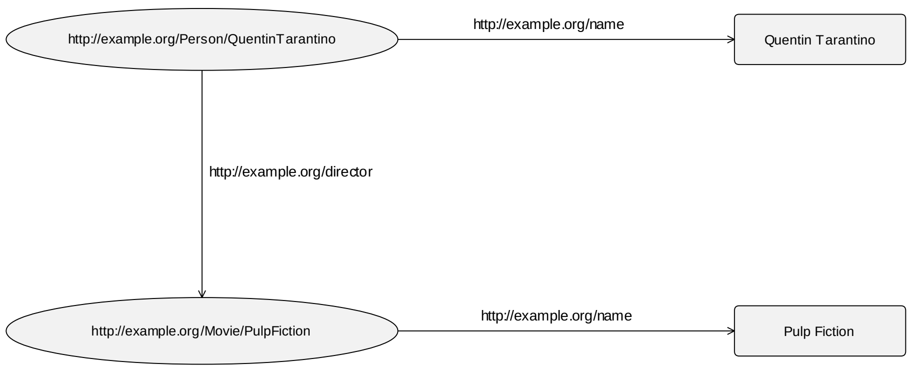

http://example.org/Person/QuentinTarantino
http://example.org/name
Quentin Tarantino
http://example.org/director
http://example.org/Movie/PulpFiction
http://example.org/name
Pulp Fiction

Figure 1 Simple example of graph-based triples representation.

The use of URIs in RDF is paramount. It is necessary to assign unique identifiers to each of the nodes so that they can be referred consistently across all the triples that describe the relationship. In a single dataset it is possible to use sequential numbers or strings to uniquely identify nodes. But for applications with multiple datasets, from heterogeneous sources, URI (especially URLs, where domain names are actually owned by the data publisher) ensure unique and consistent identifiers.

It is possible to provide information about a literal datatype. RDF supports the use of user defined and XML Schema types, which predefines a large range of basic types, including Boolean, integer, time and date. For typing complex concepts, such as resources and properties, one must use RDF Schema, as explain in subsection 2.2.

2.1.2 Persistence

Data in the RDF data model is persisted in a triplestore, a especial database for the storage and retrieval of triples. While relational databases are schema oriented, RDF triplestores are data oriented. That is, in relational databases data complies with a predefined schema, has explicit indexing and queries performs better with one-to-many relationships; on the other hand, data in a triplestore is semi-structured, triples are indexed and all relationships are many-to-many. Triplestores are built either as database engines from scratch or on top of existing relational database engines.

Just as SQL provides a query language across the relational database systems, SPARQL (Simple Protocol and RDF Query Language) provides a declarative interface for interacting with RDF graphs. It is an official W3C recommendation. SPARQL is both a standard query language and a data access protocol.

The SPARQL language consists of triple patterns, conjunctions (logical "and") and disjunctions (logical "or"). As in most the declarative languages, a query specifies a pattern in the data graph and the result set contains fragments that matched it.

2.1.3 Serialization

The RDF data model is very simple. Still, there are several methods available for serialization of RDF. A popular format is RDF/XML. In addition W3C introduced Notation 3 (N3), a text based format, that is related to Turtle and N-Triples. The non-XML serialization is easy to write by hand and, in some cases, easier to follow.

N-Triple notation is a very simple serialization, but still verbose. This simplicity makes this kind of serialization useful when hand-crafting datasets. Each line of output in N-Triple format represents a single statement containing a subject, predicate and object, followed by a dot. Every element is expressed as absolute URIs enclosed in angle brackets. The N-Triple simplicity causes redundant information that takes additional time to transmit and parse. While working with a small dataset it is not a problem, but the additional and redundant information becomes a liability when working with large amounts of data.

N3 condenses much of the information repetition in the N-Triple format. Every connection between nodes represents a triple. Since each node can have a large number of relationships, the number of characters can be reduced using prefixes. Similar to XML namespaces, N3 allows the definition of a URI prefix and identify resource URIs relative to a set of prefixes previously declared. N3 also reduces the repetition by allowing the combination of multiple statements about the same subject, by using a semicolon.

Turtle is a more verbose subset of N3 and an extension of N-Triples. It is a simple format for learning and making simple RDF Documents. The Turtle document is a collection of RDF-triples with `<subject> <relationship> <object>.` format. Each statement ends with a period and each element is an URI (except the `<object>` which can be a literal). If a subject has more than one statement, with different relationships, Turtle combines the multiple statements, using a semicolon. With Turtle it is also possible to define namespace prefixes, simplifying the document.

The RDF/XML is another way to serialize the RDF data model. Sometimes it is criticized for being difficult to read. Still it is one of the most frequently used formats. The RDF/XML is built up from a series of smaller descriptions each of which traces a path through an RDF graph. The path is described in terms of subject (nodes) and links (predicates) connecting the nodes. If there are several paths described in the document, all the descriptions must be children of a single RDF element. As with other XML documents, the top-level element is used frequently to define other XML namespaces used through the document. Paths are always described starting with a graph node, using the URI reference for the node. Predicate links are specified as child elements of the node. Literal objects can be specified as the text of an element. And if the object is a node, a new element is created.

2.2 Resource Description Framework Schema

RDF provides a way to express simple statements about resources, using properties and values. However, users also need the ability to define vocabularies that they intend to use in those statements. In other words, users need to indicate that they are describing specific kinds or classes of resources and will use specific properties in that description.

Since RDF itself provides no means for defining classes and properties, it is used a RDF extension called RDF Schema (RDFS) [1, 6] to provide a type system for RDF. As in the type systems of some object-oriented programming languages, resources are instances of one or more classes, organized in a hierarchy.

A class in RDFS corresponds to the generic concept of a type or category. A class can be used to represent almost any category of things. A resource that belongs to a class is called its instance.

In RDF a class of resource is assigned with the `rdf:type` property whose value is the resource `rdfs:Class`. The relationship between two classes is described using the predefined `rdfs:subClassOf` property to relate these two classes. The meaning of this relationship is that any instance of the subclass is also an instance of the class. A class may be a subclass of more than one class. RDF Schema also defines all classes as subclasses of class `rdfs:Resource` since the instances belonging to all classes are resources.

Besides describing specific classes, users also need to be able to describe properties that characterize those classes. In RDF Schema all properties are described using the class `rdf:Property` and the properties `rdfs:domain, rdfs:range` and `rdfs:subPropertyOf` of RDFS.

The RDF Schema also provides a vocabulary for describing how properties and resources are related. The most important information is supplied by using the properties `rdfs:domain` and `rdfs:range`.

The `rdfs:domain` property indicates that a particular property applies to a designated class. In RDF, property descriptions are, by default, independent and have global scope. Then, a RDF Schema could describe a property without a domain being specified, being possible to extend the use of a property definition to a different situation.

The `rdfs:range` property indicates that the values of a particular property are instances of a designated class. It is not possible in RDFS to define a specific property as having locally-different ranges, depending on the class of the resource it is applied to. Any range defined for a property applies to all uses of that property.

RDF Schema provides a way to specialize properties as well as classes. The specialization relationship between two properties is described using the predefined `rdfs:subPropertyOf` property. A property may be subproperty of zero, one or more properties. The range and domain properties that apply to an RDF property also apply to each of its subproperties.

The Figure 2 represents the connection between a RDF and a RDF Schema. The blocks are properties, ellipses above the dashed line are classes and ellipses bellow the dashed line are instances. The RDF resources are related with the RDFS classes, by types. The RDFS relates classes hierarchically. The properties domain and range are constrained by classes.

In summary, RDF Schema provides schema information as additional descriptions of resources but does not determine how the descriptions should be used by an application. The

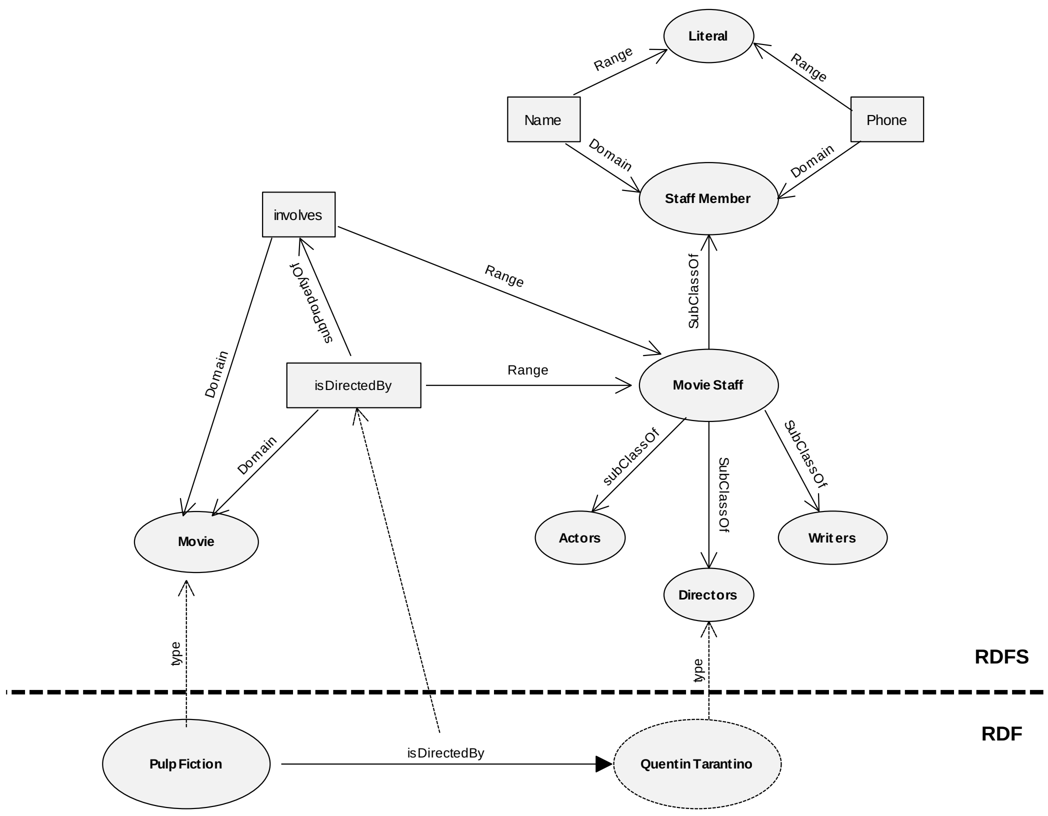


Figure 2 RDF and RDFS layers example.

statements in RDF Schema are always descriptions. They may also introduce constraints but only if the application interpreting those statements wants to treat them that way. All RDF Schemata provide a way of state additional information. If this information conflicts with the RDF data is up to the application to resolve it.

2.3 Software Tools

There is a wide range of systems described in the literature that can be used to publish existing relational databases as linked data. Some are complete database systems, as OpenLink Virtuoso, other are frameworks for developing semantic web applications, such as Jena, the framework selected for the implementation of DaPress.

OpenLink Virtuoso [8] is an open source edition of Virtuoso, an hybrid database management engine that supports multiple formats, including RDF, on top of a relational database.

The D2R Server [5] is another tool for publishing relational databases content as Linked Data. It supports RDF and HTML browsers to navigate the content of the database. It also has a SPARQL access endpoint.

Triplify [3] is a PHP plugin for web applications for small database contents (up to 100Mb). It exposes the semantic structure encoded in databases by making their content available as RDF, JSON or Linked Data. It is still a beta version and has not been updated since 2011.

Apache Jena [9] is an open source Semantic Web framework for Java. It provides an API to extract data from files, databases, URLs or a combination of these and produces RDF graphs that are serializable in RDF/XML, Turtle or N-Triple formats. This framework offers in-memory and persistent storage and supports SPARQL queries, among other query languages. Jena also provides a support for Web Ontology Language (OWL).

3 DaPress

DaPress is a tool that works as an intermediary between a semantic web client and a relational database, developed in Java with the Apache Jena Framework. This application extracts selected data from a relational database and transforms it into RDF. The generated triples are stored in a persistent triplestore using also a relational database[1].

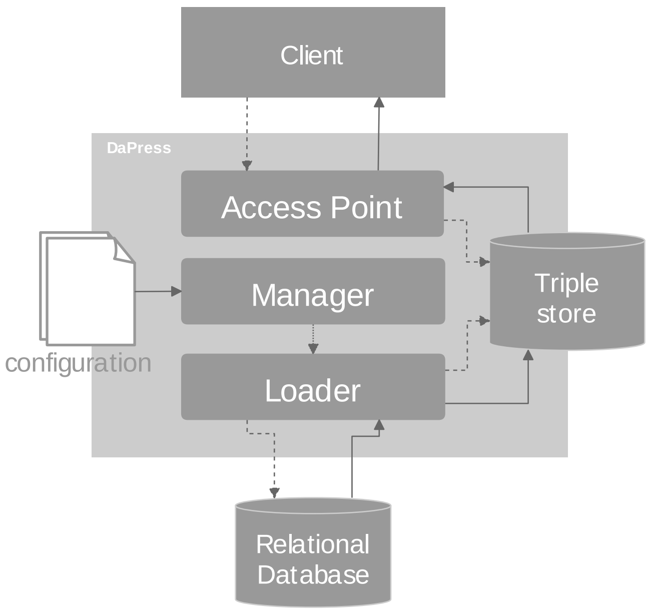


Figure 3 DaPress architecture.

As depicted in Figure 3 the architecture of DaPress aggregates three components:

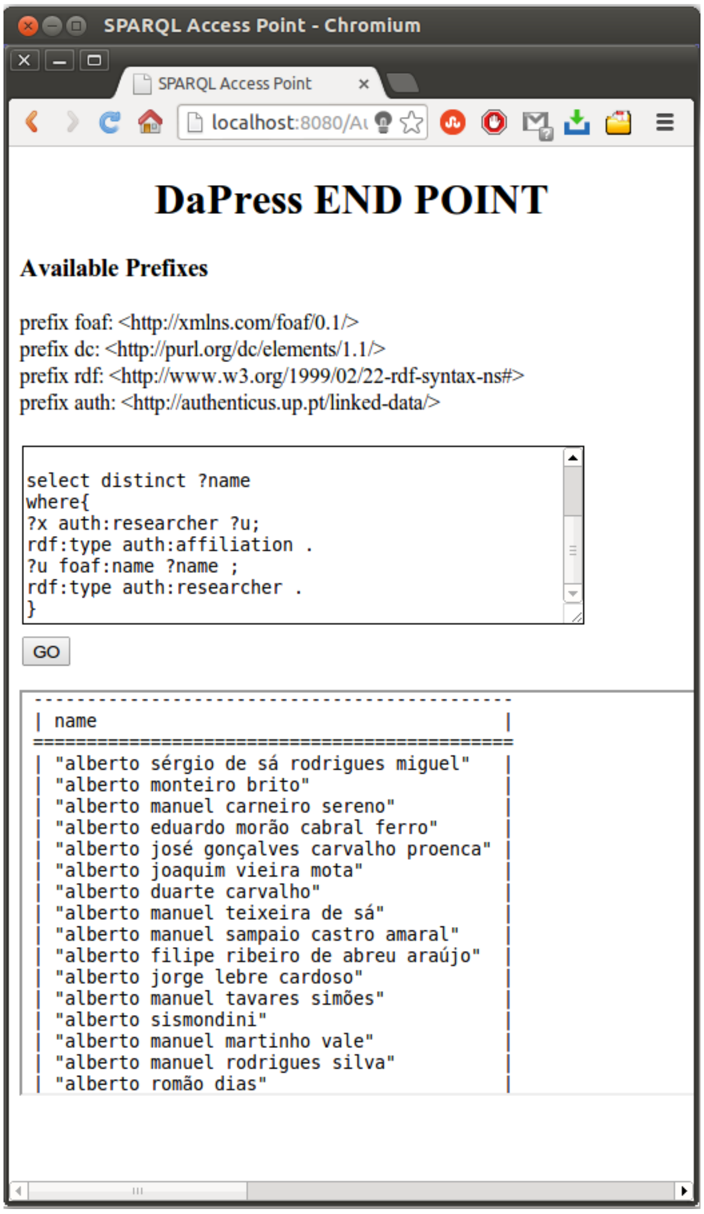
Manager The module responsible for loading configurations, opening database connections, and controlling the other modules.

Loader Is in charge of converting relational data into the RDF and the RDFS, and store it in the triplestore using the mapping algorithms from Section 3.1.

Access Point Provides a SPARQL interrogation point. It is a specialized *servlet* providing a web service. For testing purposes, there is a simple web page to interact with the stored model, writing queries and viewing responses on a web browser, as shown on Figure 4.

[1] The source database and the triplestore may share the same database management system

Figure 3 illustrates how control (dashed arrows) and data (full arrows) flow through the system. Initially, the loading process is started by the Manager using the data in configuration files, in an operation that is periodically repeated. The Loader receives that information to execute queries to the external relational database. With the data the Loader creates the RDF and RDFS models and stores them in the triplestore. Later on, when a client makes a query to DaPress, the request is handled by the Access Point that interrogates the model stored in the triplestore.



DaPress END POINT

Available Prefixes

prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix dc: <http://purl.org/dc/elements/1.1/>
prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
prefix auth: <http://authenticus.up.pt/linked-data/>

```
select distinct ?name
where{
?x auth:researcher ?u;
rdf:type auth:affiliation .
?u foaf:name ?name ;
rdf:type auth:researcher .
}
```

```
---------------------------------------------
| name                                      |
=============================================
| "alberto sérgio de sá rodrigues miguel"   |
| "alberto monteiro brito"                  |
| "alberto manuel carneiro sereno"          |
| "alberto eduardo morão cabral ferro"      |
| "alberto josé gonçalves carvalho proenca" |
| "alberto joaquim vieira mota"             |
| "alberto duarte carvalho"                 |
| "alberto manuel teixeira de sá"           |
| "alberto manuel sampaio castro amaral"    |
| "alberto filipe ribeiro de abreu araújo"  |
| "alberto jorge lebre cardoso"             |
| "alberto manuel tavares simões"           |
| "alberto sismondini"                      |
| "alberto manuel martinho vale"            |
| "alberto manuel rodrigues silva"          |
| "alberto romão dias"                      |
```

Figure 4 Screenshot of the DaPress access point web form.

The corner stone of DaPress is the mapping algorithm that converts relational data into RDF and RDF Schema driven by an XML configuration. The following two subsections detail both the algorithms and the DaPress configuration file.

3.1 Mapping Algorithm

This subsection presents the mapping algorithms of DaPress. For sake of clarity the algorithm for creating plain RDF triples from relational data is separated from the algorithm for creation RDF Schemata. Both algorithms use data provided by the XML data configuration file and data retrieved from the relational databases using SQL queries. Both algorithms produce a model, i.e. a collection of RDF triples. In DaPress these two models are merged in a single one and stored in the same triplestore.

In the following subsections, the RDF Mapping Algorithm and the RDFS Mapping Algorithm are described in detail. The last section presents an example of the application of the two algorithm.

3.1.1 RDF Mapping Algorithm

The RDF mapping algorithm receives as input configuration data and relational data and produces as output a model – a set of RDF triples created with Jena.

In Algorithm 1 the input is given by a collection of maps. Those maps resulting from configuration data have as prefix `selected`, such as `selectedTableNames`, returning a list of table names. In contrast, the input with prefix `get` corresponds to data coming from the relational database, such as: `getIds`, mapping table names to lists of ids; and `getValue`, mapping field and id pairs to values. Mappings with prefix `make` correspond to methods provided by the Jena API to create RDF elements, such as: `makeResource`, to make a resource from a type and an id.

The algorithm 1 iterates over the selected tables and for each one retrieves their identifiers from the database. For each identified record it creates a resource with `makeResource`. This resource is assigned with the *type* given by the `selectedResourceTypeName` map. This type is also assigned to the resource with the `type` property from the RDF vocabulary. For each field of the current record, a property is created using `makeProperty` function. The type associated with this property is given by the `selectedPropertyTypeName` map. According to the range selected for the field is created either a literal (typically a string) or a resource that is assigned to the object. In this last case the field value can be taken as a type, giving origin to a class hierarchy, as explain in the next sub-subsection. Finally a new statement is created and added to the model. The statement is created with the `makeStatement` using the previously created `subject`, `property` and `object`.

3.1.2 RDF Schema Mapping Algorithm

The algorithm for creating RDF Schema presented in Algorithm 2 is similar to the presented in the previous sub-subsection. Instead of creating RDF triples it creates classes and properties using the API available for that purpose in Jena. Although these methods create also RDF triples, they can be configured to use different vocabularies, such as RDF Schema or OWL, with the same implementation. To highlight the fact the model produced by this algorithm contains an ontology it is labelled as `ontModel`.

The RDF Schema algorithm has also the same inputs of the RDF algorithm. Although most of the data to create classes and properties comes from the configuration, the values from the database still have to be explored in cases where subclasses are encoded as auxiliary tables.

In Algorithm 2, for each selected table is created a new ontology class with the type name assigned to that table. This new class is added to the model and is used as domain of the properties related to this type.

Algorithm 1: RDF Mapping algorithm.

```
Input  : selectedTableNames(), selectedFieldNames()
Input  : selectedResourceTypeName(), selectedPropertyTypeName()
Input  : selectedValueAsType(), selectedRangeTypeName()
Input  : getIds(), getValue()
Output : model
model ← ∅
for tableName ∈ selectedTableNames() do
    for id ∈ getIds(tableName) do
        type ← selectedResourceTypeName(tableName)
        for fieldName ∈ selectedFieldNames(tableName) do
            predicate ← makeProperty(selectedPropertyTypeName(fieldName))
            value ← getValue(fieldName, id)
            range ← selectedRangeTypeName(fieldName)
            if range = $NULL$ then
                object ← makeLiteral(value)
            else
                if selectedValueAsType(fieldName) then
                    type ← value
                object ← makeResource(range, value)
            subject ← makeResource(type, id)
            model ∋ makeStatement(subject, predicate, object)
```

The selected fields of the current table are iterated and a property is created for each one. The previously created domain is immediately assigned to this property. The property range depends on the selected range for this field. If none was selected then it is a literal. This property is then added to the ontological model.

There is a special case when a field was selected as holding subclass names. In this case the records of this table must be iterated and a new ontological class created as subclass of the current domain. This new subclass is also added to the ontological model.

3.1.3 RDF and RDFS Mapping Algorithms Example

The following example illustrates how the two algorithms manipulate the information available in the relational database and the resulting RDF graph. Both tables, `Person` and `Town`, have an one-to-many relationship.

For each row in each table is generated a node. That node is identified by an unique URI and it is the subject of the triples. A node can be connected to another node. In the example, the property `isFrom` connects a person to a town. The node can also be connected to literal values such as the property `countryOf` that connects a town to its country.

Algorithm 2: RDF Schema Mapping algorithm.

```
Input   : selectedTableNames(), selectedFieldNames()
Input   : selectedResourceTypeName(), selectedPropertyTypeName()
Input   : selectedValueAsType(), selectedRangeTypeName()
Input   : getIds(), getValue()
Output  : ontModel
ontModel ← ∅
for tableName ∈ selectedTableNames() do
    domain ← makeOntClass(selectedResourceTypeName(tableName))
    ontModel ∋ domain
    for fieldName ∈ selectedFieldNames(tableName) do
        property ← makeOntProperty(selectedPropertyTypeName(fieldName))
        makeDomain(property, domain)
        range ← selectedRangeTypeName(fieldName)
        if range = $NULL$ then
            range ← literal
        makeRange(property, range)
        ontModel ∋ property
    if selectedValueAsType(fieldName) then
        for id ∈ getIds(tableName) do
            value ← getValue(fieldName, id)
            subClass ← makeOntClass(value)
            makeSubClass(subClass, domain)
            ontModel ∋ subClass
```

In the table `Person`, the gender column is used by the RDFS Mapping algorithm to create two different classes of `Person`, male and female. The prefix `a` is used to replace the full namespace URI `http://www.example.com`.

3.2 Configuration Files

The DaPress configuration is provided by an XML document that contains all the information required by the application. The document has four kinds of information: parameters to establish relational database connections; general configuration such as the SDB description file path and the delay of the updates; the selected resources and properties and related data. The structure of this document is formalized by an XML schema whose structure is depicted in diagram of Figure 6.

The most relevant part of the XML file is about the resources. The element `Resources` contains a sequence of elements for each resource. Each has a group of attributes that defines the name of the resource, the namespace, the type and the table in the relational database. The table is used in the SQL queries.

Each resource contains a set of properties. These properties also have a group of attributes defining the name of the property, the namespace, the column in the database, the range if applied, and a mandatory attribute. The mandatory attribute is a Boolean that allows the application to know if that attribute needs to exist; if True a `where` clause is added to the query stating that the current property (column in the query) must be `Not Null`.

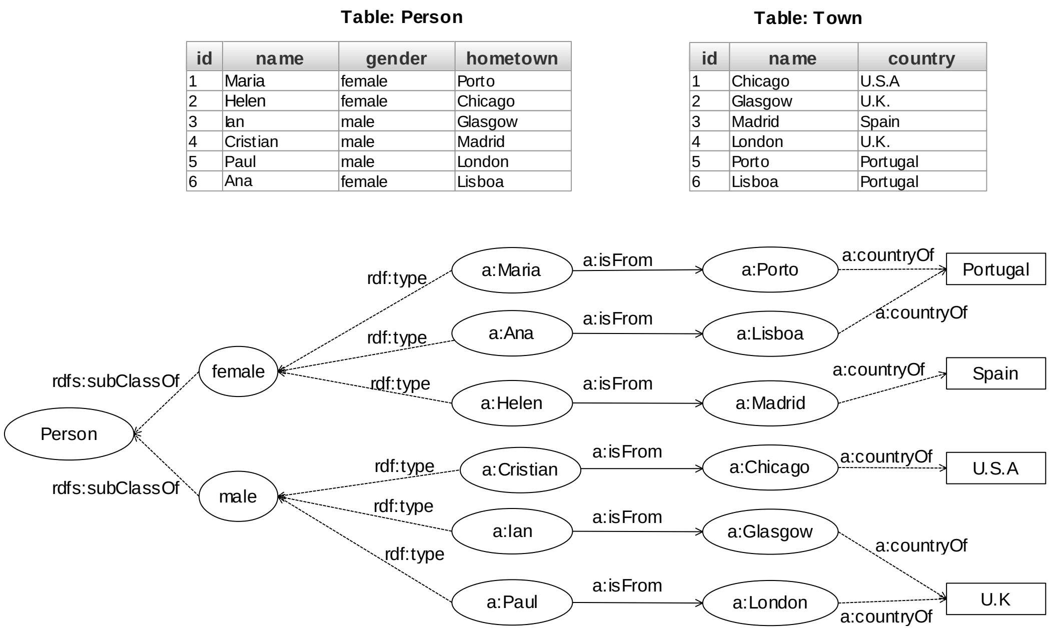


Table: Person

id	name	gender	hometown
1	Maria	female	Porto
2	Helen	female	Chicago
3	Ian	male	Glasgow
4	Cristian	male	Madrid
5	Paul	male	London
6	Ana	female	Lisboa

Table: Town

id	name	country
1	Chicago	U.S.A
2	Glasgow	U.K.
3	Madrid	Spain
4	London	U.K.
5	Porto	Portugal
6	Lisboa	Portugal

Figure 5 Example of the algorithm application.

A secondary configuration file is the SDB Description File. In this file is configured the connection to the triplestore an its path is defined in the DaPress main configuration file.

4 Validation

The validation of DaPress is based on the experience gained while publishing an existing relational database. The relational database selected for this purpose is part of *Authenticus* [7], a system to automatically assign publications and their authors to known Portuguese researchers and institutions. This system has several algorithms to perform the author name disambiguation and identification. One of the main outcomes of this project is a normalized and validated database of Portuguese publications, that is an apt example of the kind of data that should become available as linked data.

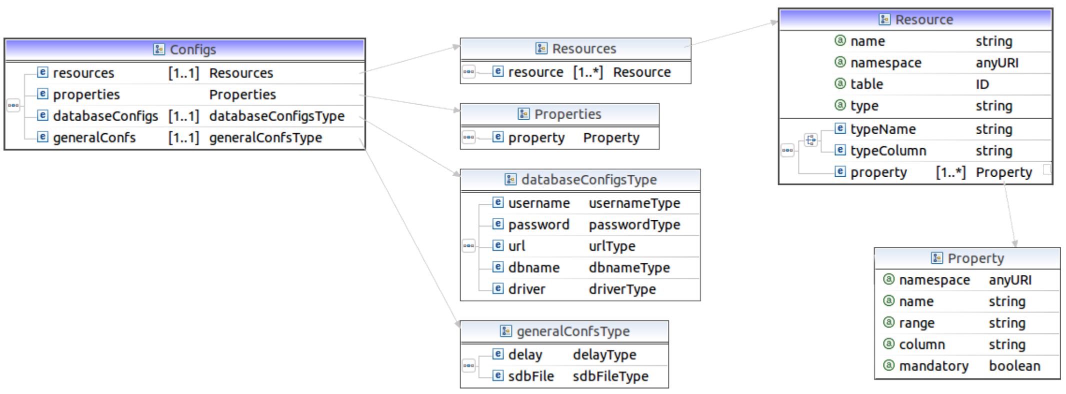


Figure 6 XML Configuration Schema.

The Authenticus database has currently 67 tables and a 210Mb of data. Of these 13 tables where selected with a size of 36,2Mb. The data sizes were computed from the SQL dumps of the referred sets of tables. The tables currently being used contain data on researchers, institutions, publications and journals. Some of the tables contain many-to-many relationships among these base entities. Tables from the original database that just support the web application where excluded from this mapping, such as those related to user management or containing precomputed values to speedup frequently requested listings.

The triplestore resulting from the mapping has a size of 153Mb when exported as an SQL dump. The significant increase in size is easily explained by the "explosion" in the number of records stored in the triplestore. The triplestore of DaPress and the database of Authenticus are currently on a two different relational database management system, although both running MySQL.

The machine where the mapping was processed is a Pentium 4 running at 2,4Ghz with 8Gb of RAM. It is operated by Linux Mandriva 2009 with a 2.6.19 kernel. The DaPress executed the mapping process in 194 minutes generating 1.456.353 triples, generating on average 1 triple in 0,006 seconds and producing 12Kb of data in one second. It should be noted that the machine available for these tests is rather old, with a single processor, thus the mapping should be even faster on a multi-core machine. Nevertheless, the order of magnitude of this time requires an incremental algorithm that is already planned for the next version of DaPress.

5 Conclusions and Future Work

This paper presents ongoing research to create a tool for publishing the content of relational database as linked data. The major contribution of this research is a pair of mapping algorithms, driven by configuration data stored in a single XML document, that convert relational data to RDF, and relational schemata to RDF Schema. These ideas are incorporated in the DaPress system and the design and implementation of this tool are also relevant contributions of this research. To validate the proposed approach DaPress was used with the content of the Authenticus database. Authenticus is a system that automatically assigns publication authors to known researchers and institutions. The experience gained using DaPress with Authenticus led to the identification of a number of issues in the current version that will be tackled in a near future.

The use of XML documents proved to be a simple and expedite way to define and store mapping information. However, it requires some knowledge of XML and the use of another tool to browse the relational database schema. An administrative web interface could show the tables and fields available on the relational database, enabling their selection and renaming for the mapping process.

The conversion from relational data to RDF does not increase the size of the data. However, the time necessary to convert the data, about a minute per megabyte, is too high for a regular update of the triplestore. The next version of DaPress must have an incremental algorithm to avoid reconverting unchanged data. This will be a challenge since DaPress does not assume any particular configuration of relational tables, such as the existence of time stamp fields with the creation/modification date. The current version of DaPress produces an ontology using RDF Schema, which includes the class hierarchy and the definition of properties based on those classes. This ontological data could be extended with OWL definitions, stating properties that cannot be represented in RDF Schema or that cannot be inferred from the relational schema. The use of OWL in DaPress must be investigated and may be included in future versions.

Linked data published by DaPress is ready to be interconnected with similar or related sources, by sharing URIs of classes, properties and resources, or by relating them at the ontological level. This is the case of the RDF data of Authenticus that can be interlinked with the Digital Bibliography & Library Project (DBLP), a related system that stores publication data that also has a SPARQL access point.

After interlinking data in DaPress with related sources it will be possible to revert the publishing process from those sources into the local triplestore. That is, related RDF data from remote sources will be available for download into the triplestore of DaPress and translated to a relational database. For instance, RDF data from DBLP could be downloaded to the DaPress triplestore containing Authenticus data. The mapping configuration could then be used in the other direction, to convert data from the triplestore to the original relational database, avoiding the use of ad-hoc data converters.

References

1 *A Semantic Web Primer.* MIT Press, 2004.
2 *Programming the Semantic Web.* O'Reilly Media, 2009.
3 Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann, and David Aumueller. Triplify: light-weight linked data publication from relational databases. In *Proceedings of the 18th international conference on World wide web.* ACM, 2009.
4 Tim Berners-Lee. Design issues: Linked data, 2006.
5 Christian Bizer and Richard Cyganiak. D2r server – publishing relational databases on the semantic web. Poster at the 5th International Semantic Web Conference, 2010.
6 Dan Brickey and R. V. Guha. Rdf vocabulary description language 1.0: Rdf schema. Technical report, World Wide Web Consortium, 2004.
7 Sylwia Teresa Bugla. Name identification in scientific publications. Master's thesis, Universidade do Porto, 2009.
8 Orri Erling and Ivan Mikhailov. Virtuoso: Rdf support in a native rdbms. In *Semantic Web Information Management*, pages 501–519. Springer, 2010.
9 Apache Software Foundation. Apache jena.
10 Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. Technical report, World Wide Web Consortium, 2004.

Seqins – A Sequencing Tool for Educational Resources

Ricardo Queirós[1], José Paulo Leal[2], and José Campos[3]

1 **CRACS & INESC-Porto LA & DI-ESEIG/IPP, Porto, Portugal**
`ricardo.queiros@eu.ipp.pt`
2 **CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal**
`zp@dcc.fc.up.pt`
3 **Lusíada University**
`jjscampos@eu.ipp.pt`

Abstract

The teaching-learning process is increasingly focused on the combination of the paradigms "learning by viewing" and "learning by doing." In this context, educational resources, either expository or evaluative, play a pivotal role. Both types of resources are interdependent and their sequencing would create a richer educational experience to the end user. However, there is a lack of tools that support sequencing essentially due to the fact that existing specifications are complex. The Seqins is a sequencing tool of digital resources that has a fairly simple sequencing model. The tool communicates through the IMS LTI specification with a plethora of e-learning systems such as learning management systems, repositories, authoring and evaluation systems. In order to validate Seqins we integrate it in an e-learning Ensemble framework instance for the computer programming learning.

1998 ACM Subject Classification K.3 Computers and Education; K.3.1 Computer Uses in Education; Computer-managed instruction (CMI)

Keywords and phrases e-Learning, Learning management systems, Automatic evaluation

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.83

1 Introduction

In the last few years, higher education institutions have been challenged by the emergence of new information and communication technologies (ICT). In this context, e-learning has been adopted as the preferred channel for the dissemination of knowledge.

In the mid-1980s, educational researchers found that learning was faster and better when some practice/evaluative resources were replaced by worked-out examples that demonstrated the lesson skills. Sweller and Cooper [10] found that errors were reduced in half when 12 math practice resources were replaced by 6 worked-out examples, each followed by practice.

Many experiments showed a combination of examples and practice exercises engages novice students rather than just practice. For instance, in a lesson on how to use programming variables, an example showing the steps to declare and initialise a variable would be followed by a practice in which the learner must repeat these steps in a slightly different situation. When viewing an example, the learner's working memory is free to build a new mental model of the skill. Then the learner can try out the new mental model in a practice problem [3].

Based on these thoughts, Sweller and Cooper [10] used a learner-centered approach to define a constructivist learning model, depicted in Figure 1, based on two learning paradigms: "learning by viewing" and "learning by doing." In this model educational resources, either expository or evaluative, play a pivotal role.

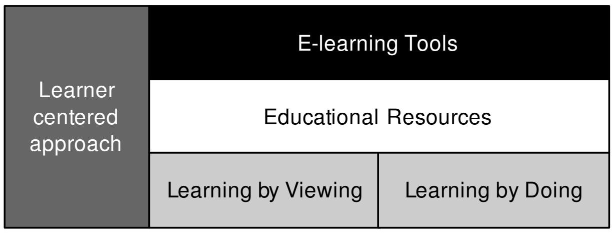


Figure 1 Sweller and Cooper constructivist learning model.

These resources are crucial for the teaching-learning process. It is important that an e-learning system provides a collection of resources covering a course syllabus and with different levels of difficulty. It has been shown that this can improve the performance of students and their satisfaction levels [12]. Students with lower computer skills can begin by viewing examples and solving easier problems in order to progressively learn and to stay motivated to read and solve harder topics later [6]. At the same time this gives them experience which is one of the factors that has a greater influence on student success in learning [13]. In recent years, a large number of educational resources have been developed and published and mostly of them stored in proprietary systems for their own use. Although some standard organisations (e.g. ISO/IEC, IMS, ADL) have created specifications (e.g. IEEE LOM, IMS CP, SCORM, IMS CC) for the description and aggregation of educational resources, each one has its own format, making it difficult to share among instructors and students. This poses several issues on the resources interoperability among e-learning systems.

At the same time, both types of resources are interdependent and their sequencing would create a richer educational experience to the end user. However, there is a lack of tools that support sequencing essentially due to the fact that existing specifications are complex.

In this paper we present Seqins as a sequencing tool for educational resources. Seqins is based on a simple sequencing model where students after seeing a set of examples on a particular topic, consolidate their knowledge solving a set of exercises on the same topic. The student must solve an (a set of) exercise(s) before proceeding to next topic. This approach has several advantages. The most important is the ability to foster heterogeneous classes, enabling students with different learning rhythms to progress autonomously.

The remainder of this paper is organised as follows. Section 2 enumerates the standards and specifications for sequencing resources and integrating tools in e-learning environments. In the following section we present Seqins with emphasis on its sequence, data and communication model. Then, to evaluate the sequencing tool, we present its integration on a network for the computer programming learning. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2 E-learning Specifications

2.1 E-learning Sequencing Specifications

Sharable Content Object Reference Model (SCORM) is a collection of standards and specifications for web-based e-learning. It allows the communication between client side content and a host system called the run-time environment, which is commonly supported by a learning management system. SCORM also defines how content may be packaged into a transferable ZIP file called "Package Interchange Format." Despite its enormous popularity

Listing 1 IMS SS excerpt within a IMS CP manifest.

```
<manifest ...>
 <metadata>...</metadata>
 <organizations default="TOC1">
  <organization identifier="TOC1">
   <title>Photoshop Tutorial</title>
    <item identifier="ITEM45" identifierref="QUESTION6">
     ...
     <imsss:sequencing>
      <imsss:controlMode choice="false" forwardOnly="true"/>
      <imsss:rollupRules>
       <imsss:rollupRule childActivitySet="all">
        <imsss:rollupConditions>
         <imsss:rollupCondition condition="attempted"/>
        </imsss:rollupConditions>
        <imsss:rollupAction action="completed"/>
       </imsss:rollupRule>
      </imsss:rollupRules>
     </imsss:sequencing>
    </item>
    ...
   </organization>
  </organizations>
  <resources>
   <resource>...</resource>
   ...
  </resources>
</manifest>
```

some researchers have questioned the instructional value of the SCORM model. The focus of the critics was whether atomic de-contextualized learning objects (LO) can support an effective pedagogic goal and whether it could convey an unified learning experience. In order to address these concerns SCORM included the IMS Simple Sequencing specification (IMS SS).

The IMS SS is a specification used to describe paths through a collection of learning activities. The specification declares the order in which learning activities are to be presented to the learner and the conditions under which a resource is delivered during an e-learning instruction. The IMS SS organizes and sequences the package items in a XML manifest section called Organizations. This section aims to design pedagogical activities and to articulate the sequencing of instructions. By default, it uses a tree-based organization of learning items pointing to the resources (assets) included in the package.

Unfortunately, IMS Simple Sequencing does not make the instructional designer's task of bringing instructional order and meaning through sequencing easy. The standard is highly technical and procedural, and lacks support for associating instructional meaning with the sequencing code (e.g. domain related segments). In fact, the IMS SS specification has been the target of much criticism such as:

- Modularity - included in the Learning Objects (IMS CP);
- Scope - widening the scope exaggerated ("spray and pray");
- Utility - More focused on quantity than quality ("shovelware");
- Complexity - difficult to implement.

The IMS CP manifest excerpt in listing 1 shows some of the IMS SS tags to define the conditions under which a specific item is delivered during an e-learning instruction.

IMS attempts to address this pedagogical issue by supplementing its simple sequencing standard with the IMS Learning Design standard. The IMS LD specification is a meta-language for describing pedagogical models and educational goals that can be constructed (e.g. Reload, LAMS) and rendered (e.g. CopperCore, .LRN). However (once again) this standard provides templates for learning in a multi-person/entity setting and is not really applicable to the web-based, single learner setting which is the focus of SCORM 2004.

2.2 E-learning Integration Specifications

Data integration is the simplest and most popular form of integration in content management. This type of integration uses the import/export features of both systems and relies on the support of common formats.

The major LMS vendors include also APIs to allow developers to extend their predefined features through the creation of plug-ins. Blackboard uses the Building Blocks technology to cover the integration issues with other systems allowing third parties to develop modules using the Building Blocks API. The new Moodle versions (from v.2.0 released in November 2010) includes several API to enable the development of plug-ins by third parties such as the Repository API for browsing and retrieving files from external repositories; and the Portfolio API for exporting Moodle content to external repositories. These two API are based on the File API - a set of core interfaces to allow Moodle to manage access control, store and retrieve files. The new File API aims to enhance file handling by avoiding redundant storage. A common interoperability standard that is increasingly supported by major LMS vendors is the IMS LTI specification. The IMS LTI provides a uniform standards-based extension point, allowing remote tools and content to be integrated into LMSs. The main goal of LTI is to standardize the process of building links between learning tools and the LMS. There are several benefits from using this approach: educational institutions, LMS vendors and tool providers by adhering to a clearly defined interface between the LMS and the tool, will decrease costs, increase options for students and instructors when selecting learning applications and also potentiate the use of software as a service (SaaS). The LTI has 3 key concepts as shown in Figure 2 [4]: the Tool Provider, the Tool Consumer and the Tool Profile.

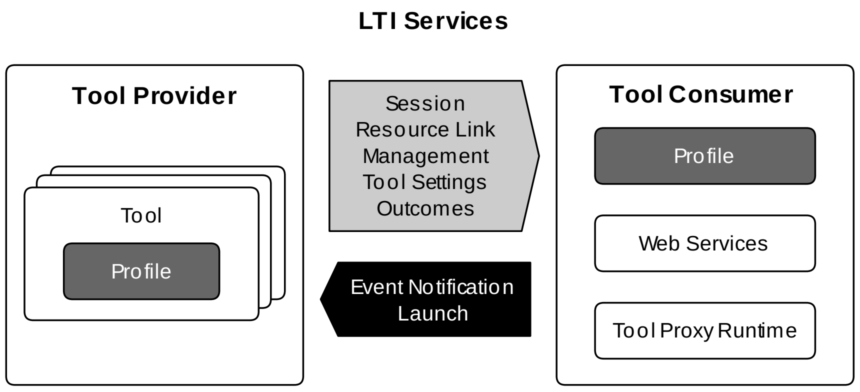


Figure 2 IMS Full LTI.

The tool provider is a learning application that runs in a container separate from the LMS. It publishes one or more tools through tool profiles. A tool profile is an XML document describing how a tool integrates with a tool consumer. It contains tool metadata, vendor information, resource and event handlers and menu links. The tool consumer publishes a Tool Consumer Profile (XML descriptor of the Tool Consumer's supported LTI functionality that is read by the Tool Provider during deployment), provides a Tool Proxy Runtime and exposes the LTI services.

A subset of the full LTI v1.0 specification called IMS Basic LTI exposes a single (but limited) connection between the LMS and the tool provider. In particular, there is no provision for accessing run-time services in the LMS and only one security policy (OAuth protocol[1]) is supported. For instance, to export content from Moodle to Mahara using the Basic LTI the teacher (or LMS administrator) must first configure the tool (Mahara) as a Basic LTI tool in the course structure. When a student selects this tool, Moodle launches a Mahara session for the student. The web interface for this session can either be embedded in Moodle's web interface as an iframe or launched in a new browser window.

Recently, IMS launched the Learning Tools Interoperability v1.1.1 (released in July 2012) that combines Basic LTI and LTI into just Learning Tools Interoperability. This version includes updates and clarifications as well as support for an outcomes service and bidirectional communication support. In this version, LTI has also support for the TP to call IMS Learning Information Services (LIS) when those services can be made available to the TP. LTI does not require LIS services, but the TC can send LIS key information to the TP using values in the basic launch request.

Comparing these three approaches [1] one could say that data integration is the best option when the development effort must be kept to a minimum or no one with technical skills (specially programming skills) is available, since the other two strategies require them. This strategy has also the advantage of not coupling the two systems and enabling a bi-directional communication.

API integration is best suited when batch integration is required since the other two strategies involve the use of the GUI of both systems. For instance, if the work of the students of a given set of courses must be copied on a regular basis from the LMS to their portfolios then the API strategies are recommended. The major drawbacks of this approach are the amount of development required and the tight coupling between the LMS and the other e-learning system, since special plug-ins must be implemented and API are vendor specific.

Tool integration is arguably the best choice in general since it provides a good balance between implementation effort and coupling and security. This is especially true if only unidirectional communication is required and Basic LTI is used. This tool integration flavour is simple to implement and is already supported by most LMS vendors. If bidirectional communication is required then full LTI is needed but in this case the implementation is harder and few LMS vendors support this flavor of the specification. In both cases, tool integration has the added value of providing some basic security features based on the OAuth protocol aiming to secure the message interactions between the Tool Consumer and the Tool Provider.

[1] OAuth security protocol: `http://oauth.net/`

3 Seqins

This section presents Seqins as a sequencing tool for educational resources categorized as:

- Expositives (e.g.: videos, PDF, HTML + images);
- Evaluatives (e.g.: assignments, tests, exercises).

Seqins follows a simple sequencing model. After seeing examples on a particular topic, students consolidate their knowledge by solving a set of exercises related to topic. The success in solving exercises leads the student to the next topic.

The integration of Seqins with other e-learning systems is straightforward since Seqins supports the IMS LTI 1.1. specification who has been increasingly supported by e-learning systems, especially by LMSs.

The following subsections present Seqins architecture and main components, and describe its sequence, data and communication models.

3.1 Architecture

Seqins is a web application for sequencing educational resources that mediates between an LMS and an Evaluator. The role of the LMS is to manage instruction and it will initiate a session in Seqins to sequence a course for a particular student. During the interaction with the student Seqins presents content (HTML, PDF, videos) embed in its web interface. The evaluation of exercises requires the use of a specialised e-learning system designated here as Evaluator. The communication of Seqins with these other systems relies on the LTI protocol described in the precious section.

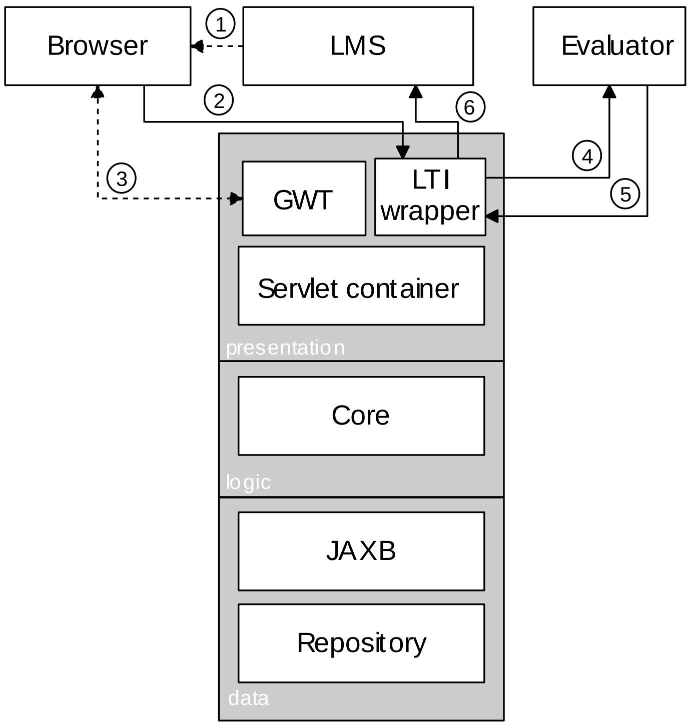


Figure 3 Seqins architecture.

Figure 3 presents the architecture of Seqins following a 3-tier model, divided in presentation, logic and data, highlighting the interaction with the surrounding systems, namely the LMS, the web browser and the automatic evaluator. In this diagram the LTI interactions are represented by solid arrow while plain HTTP interaction are represented by dotted arrows. For better understanding these arrows are marked with a number within a circle representing the sequence of invocations.

A typical use of Seqins starts with a HTTP message replied by the LMS to the student's browser (1) that starts an LTI request processed by the LTI wrapper (2). This request start a Seqins course for the student and creates a GWT interface (more on GWT later on) on the browser where the students interacts with the system. During this interaction, starting an exercise triggers an LTI request to the evaluator (4) handled by LTI Wrapper. Eventually the Evaluator answers with an LTI communication carrying a grade (5) and this information is processed by the core and persistently stored. The steps (4) and (5) are repeated for each attempted exercise. Finally, the results obtained by the student in the course are reported to the LMS using LTI.

One of the key components in this architecture is the LTI Wrapper that implements both sides of the LTI communication. This component receives LTI requests from both the LMS and the Evaluator, and also invokes their services. This Java package was developed for Seqins but can be used by any Java application requiring LTI communication. Both the LTI Wrapper and the GWT user interface are supported by a java servlet container.

The GUI component was developed using an Ajax framework called Google Web Toolkit (GWT). GWT is an open source Java software development framework that allows a rapid development of AJAX applications in Java. When the application is deployed, the GWT cross-compiler translates Java classes of the GUI to JavaScript files and guarantees cross-browser portability. The controls required by GUI are provided by SmartGWT, a GWT API's for SmartClient, a Rich Internet Application (RIA) system. The GWT code is organised in two main packages: the back-end (server) and the front-end (client). The back-end includes all the service implementations triggered by the user interface. These implementations are centralised in the Core component that relies on the LTI wrapper which implements the LTI 1.1 specification.

The data layer deals with the data persistence through JAXB (Java Architecture for XML Binding). JAXB allows Java developers to map Java classes to XML representations providing two main features: the ability to marshal Java objects into XML and the inverse, i.e. to unmarshal XML back into Java objects. In the following subsections we detail these XML representations.

3.2 Data Model

Seqins organizes its data based on two entities: student and course. Since Seqins follows a learner-centered approach it is important to store the learner data. This data is gather by the Tool Producer (Seqins) upon a launch request from the Tool Consumer (typically an LMS). The LTI variables sent in the request are organized in the following data levels: Course, Resource, User, Context and OAuth. In this case two levels are used: the data level is used to store/update information (e.g. identification, name, email) about the user; and the resource level to assign to the respective learner the last resource viewed/solved. Most of this information is stored in the filesystem in XML documents one for each user. Listing 2 shows one of these files.

Listing 2 XML representation of a learner.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<student id="2">
   <href>http://ensemble.dcc.fc.up.pt/crimsonHex/lo/unit01/2</href>
   <lastRequest>2012-12-10T17:18:27.464Z</lastRequest>
   <lastUpdate>2012-11-27T17:22:15.335Z</lastUpdate>
</student>
```

The root element student has an id attribute that identifies the student. The element contains three sub-elements:

- *href* - is the URL of the last resource viewed/solved;
- *lastrequest* - is a timestamp of the last request to see if there has been any change in student status;
- *lastupdate* - corresponds to the last time that a change occurred. In order to not generate too many requests we check for any update after the last request.

The course entity formalizes how a course is structured. The formalization of the course controls the generation of the tree that appears in Seqins's GUI to enable navigation through the course resources. The formalization of a course is made by the definition of a XML Schema depicted in Figure 4, and Listing 3 shows a valid XML instance of a course.

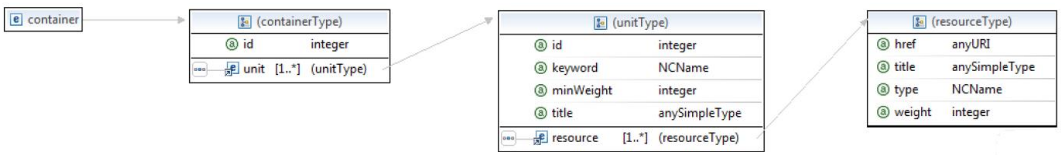


Figure 4 XML representation of a course.

A *course* element contains one or more *unit* elements. Each unit contains a set of *resource* elements. Each resource has four attributes: *title* – name of the resource; *type* – type of the resource (*exp* – expositive and *eval* – evaluative); *weight* – weight for evaluation purposes; and *href* - URL pointing to the resource on the Web.

This XML representation contains two important attributes: weight (w) and minWeight (mw). These attributes operate at resource and unit level respectively and are used by Seqins to determine the progress of a learner within a course. The sequencing model is quite simple:

1. Weights are assigned to the evaluative resources (R) of an unit;
2. Within a unit the visualization/solving is sequential;
3. In order to obtain success in an unit U' the sum of all weights of the successfully solved resources must be greater than or equal to the minimum weight of the associated unit.

$$\sum_{i=1}^{U'length} wR_i \in U' \geq mwU' \tag{1}$$

This model fosters competitiveness since the success in a unit unlocks the access to the following, as if it were a computer game. This feature is assumed by the authors and it has its own risks. Competitive learning is a learning paradigm that relies on the competitiveness of students to increase their programming skills [2, 9], although the concept of “winners and losers” may hinder the motivation of some students [11].

Listing 3 XML representation of a course.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<course title="C# Programming" author="Jose Campos" date="15-03-2013">
 <unit id="1" title="Intro C#" minWeight="70">
  <resource
    title="First program - Hello World!"
    type="exp" weight="0"
    href="http://ensemble.dcc.fc.up.pt/crimsonHex/lo/unit01/1" />
  <resource
    title="Hello ESEIG!!"
    type="eval" weight="40"
    href="http://ensemble.dcc.fc.up.pt/petcha/unit01/1" />
  <resource
    title="Constants and arithmetic operators"
    type="exp" weight="0"
    href="http://ensemble.dcc.fc.up.pt/crimsonHex/lo/unit01/2" />
  <resource
    title="Area of a circle"
    type="eval" weight="30"
    href="http://ensemble.dcc.fc.up.pt/petcha/unit01/2" />
  </unit>
   ...
</course>
</student>
```

3.3 Communication Model

The integration of Seqins with other e-learning systems relies on the LTI specification. The LTI specification recommends REST as the web service flavour for exchanging data among e-learning tools. The LTI functions are summarised in Table 1.

Table 1 LTI functions.

Function	REST	LTI	
		Basic	Full
`Launch`	`POST TA_URL` $<$ `LTI_PARAMETERS`	yes	yes
`ReplaceResult`	`POST LIS_OUTCOMES_URL` $<$ `LIS_SOURCE_ID + GRADE`	no	yes
`ReadResult`	`POST LIS_OUTCOMES_URL` $<$ `LIS_SOURCE_ID` $>$ `GRADE`	no	yes
`DeleteResult`	`POST LIS_OUTCOMES_URL` $<$ `LIS_SOURCE_ID`	no	yes

The `Launch` function allows the execution of a particular external tool within the LMS. Two steps are required before launching Seqins from the LMS: 1) the teacher (or LMS administrator) must configure Seqins as an external tool in the LMS control panel by setting the name and the URL of the external tool; 2) the teacher must add an activity to the course structure referring the external tool. Later on, when a student selects the external tool, the LMS uses the URL to launch Seqins through an HTTP POST. This request includes a set of launch parameters (`LTI_PARAMETERS`) as hidden form fields. Table 2 organizes the most important parameters in four groups.

Table 2 LTI launch parameters.

Groups	Variables	Description
Resource	`resource_link_id`	Unique identifier of a resource
	`resource_link_title`	A title for the resource
	`resource_link_description`	A description for the resource
User	`user_id`	Unique identifier of a user
	`user_image`	URI for an image of the user
Context	`context_id`	Context id of the link being launched
	`context_title`	A title of the context
	`context_label`	A label for the context
LIS	`lis_person_name_full`	Full name of the user
	`lis_person_contact_email_primary`	E-mail of the user
	`lis_outcome_service_url`	Unique identifier of the launch
	`lis_result_sourceid`	Outcomes service URL of the TC

This list can be extended by adding custom parameters. The syntax is

`custom_keyname = value`.

Three new parameters were added to the launch request: `custom_collection_id` — defines a link for a collection of exercises in an IMS DRI repository; `custom_sequencing` — defines if the exercises should be solved sequentially; `custom_time_limit` - defines a date/time limit for the solving of all exercises. Listing 4 shows a subset of the launch parameters that the LMS (Tool Consumer) sends to Seqins (Tool Provider).

In this example, Seqins presents the exercises of the `vectors` collection and students should solve them sequentially till November 7, 2011.

Table 1 also refers to three functions included in the IMS LIS Outcomes Service. These functions use the `lis_result_sourceid` parameter included in the launch request that is unique for every combination of `resource_link_id` / `user_id` parameters and identifies a unique row and column within the TC gradebook. After computing a grade, the TA calls the LTI Basic Outcomes Service using the URL stated in the `lis_outcome_service_url` launch parameter. The service supports setting, retrieving and deleting of LIS results associated with a particular user/resource combination (`lis_result_sourceid` parameter). The `replaceResultRequest` function sets a numeric grade (0.0 - 1.0) for a particular result. The `readResultRequest` function returns the current grade for a particular result. The `deleteResultRequest` function deletes the grade for a particular result.

Listing 4 LTI launch with default parameters.

```
resource_link_title = Sum two vectors
lis_person_name_full= Pimenta Ana
roles = Student
context_title = Algorithms and Programming
lis_result_sourceid={"data":{"instanceid":"1","userid":"2","launchid":1914382991},"hash":"..."}
lis_outcome_service_url=http://crimsonhex.dcc.fc.up.pt:8080/moodle/mod/lti/service.php
custom_collection_id = http://crimsonhex.dcc.fc.up.pt:8080/crimsonHex/lo/myCollection/vectors
custom_sequencing = true
custom_time_limit = 2011-11-07 12:00:00
```

4 Integration on Ensemble Framework

For validation purposes, Seqins was integrated with an instance for computer programming of an e-learning framework called Ensemble [7].

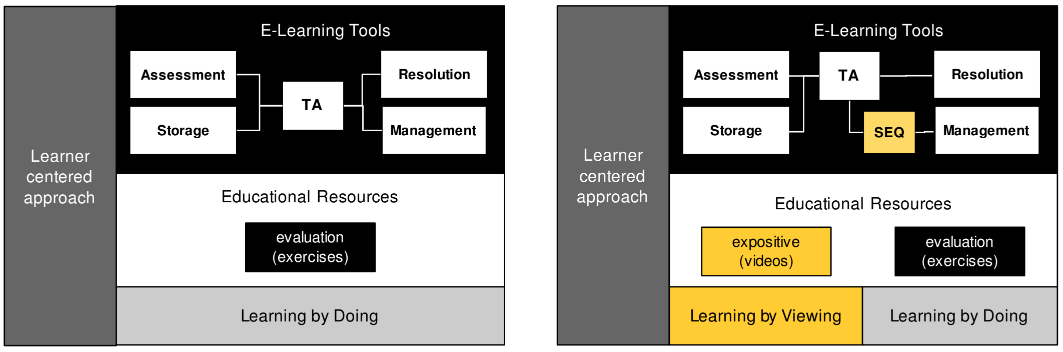


Figure 5 Evolution of Ensemble framework instance to the Sweller and Cooper model.

Ensemble is a conceptual tool to organize networks of e-learning systems and services based on international content and communication standards. Ensemble is focused exclusively on the teaching-learning process. In this context, the focus is on coordination of educational services that are typical in the daily lives of teachers and students in schools, such as the creation, resolution, delivery and evaluation of assignments.

The Ensemble instance for the computer programming domain relies on the practice of programming exercises (evaluative resources) to improve programming skills. This instance includes a set of components for the creation, storage, visualisation and evaluation of programming exercises orchestrated by a central component (teaching assistant) that mediates the communication among all components. The integration of Seqins in this framework instance complies with the model advocated by Sweller and Cooper (Figure 5).

This integration is made at two levels: **communication** and **data**. At communication level Seqins implements the LTI 1.1 specification for the communication with other e-learning systems. The components diagram of Seqins is depicted in Figure 6.

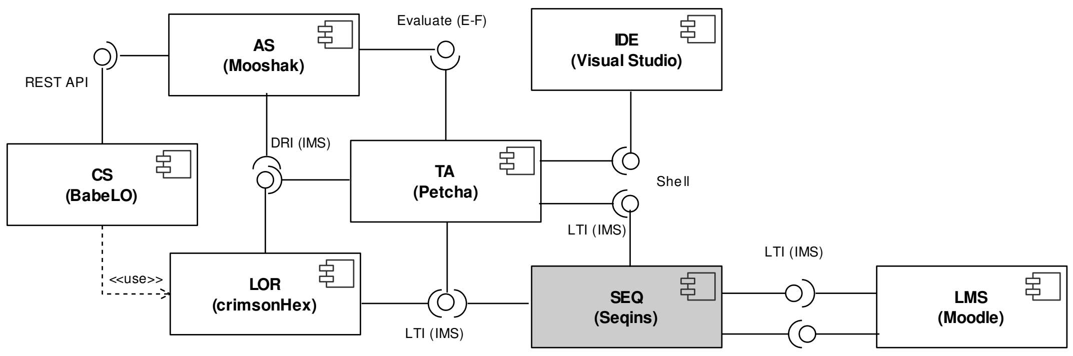
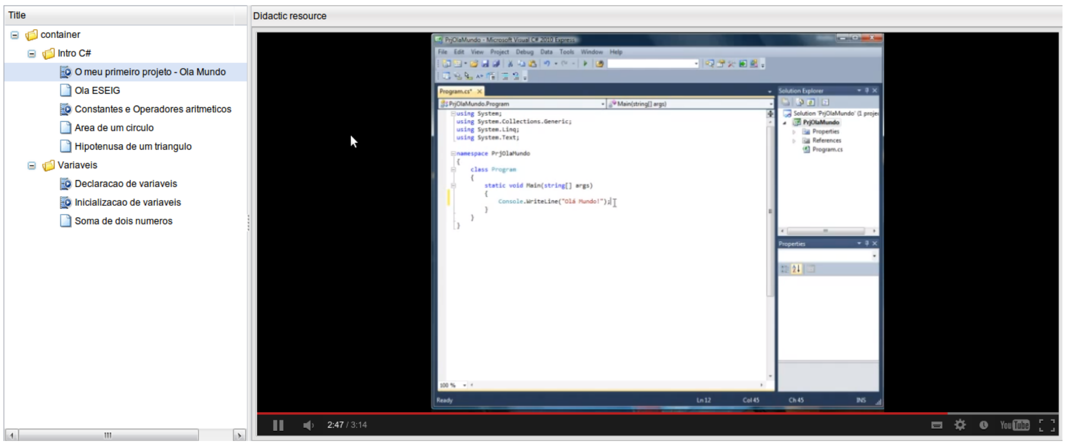


Figure 6 Seqins components diagram.

The integration of Seqins with the LMS and the Teaching Assistant (TA) relies on the LTI specification. After the launch of Seqins through the LMS the LTI parameters are sent by HTTP POST to Seqins. Then, if the learner select an evaluative resource, the same parameters are sent by HTTP POST to the TA. The TA has two main tasks in this setup: to assist teachers in the authoring exercises and to help students in solving them. When a student submits an exercise for evaluation, the grade is sent back to Seqins. This grade will influence the progress of the student. At same time Seqins clones the HTTP response of the TA and sends it back to LMS in order to feed the LMS's grade book.

At the data level, Seqins uses two types of resources: expositives and evaluatives. These resources can be seen in Figure 7 in the left panel.


Title

- container
  - Intro C#
    - O meu primeiro projeto - Ola Mundo
    - Ola ESEIG
    - Constantes e Operadores aritmeticos
    - Area de um circulo
    - Hipotenusa de um triangulo
  - Variaveis
    - Declaracao de variaveis
    - Inicializacao de variaveis
    - Soma de dois numeros

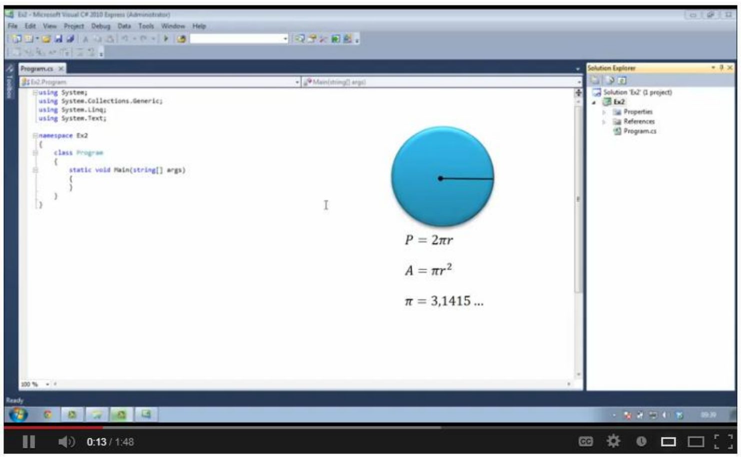
Didactic resource

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PrjOlaMundo
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Olá Mundo!");
        }
    }
}
```

Figure 7 Seqins screenshot.

The former are typically videos (Figure 8) with working examples of exercises solving. The videos were created with Camtasia, a software that records screen activity and voice.



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Ex2
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Figure 8 Video screenshot.

The videos were deployed on Youtube. The design requirements for the videos were:

- Coverage - covering all the curricula;
- Diversity - several difficulty levels;
- Fragmented - “video clip time” ($\leq$ 5 minutes);
- Complete - composed by pictures, sound, subtitles, layers.

The evaluative exercises comply with the IMS CC package specification and were extended through an interoperability language called PExIL [8]. PExIL describes the programming exercise life cycle from its creation to its evaluation. The resources were created in the Teaching Assistant and stored in a IMS DRI compliant repository (e.g. CrimsonHex [5]).

5 Conclusions and Future Work

Seqins is a sequencing tool for educational resources. It has a simple sequencing model that depends on the score obtained by the learner on solving exercises. Seqins can communicate with any tool that supports the IMS LTI specification. In order to evaluate this feature, Seqins was integrated with an Ensemble instance for computer programming learning. In this context, learners can launch Seqins through the LMS and browse resources of two types: expositive and evaluative. The former are typically short videos with workout examples on solving programming exercises. The latter are programming exercises that learners should solve on their favourite IDE guided by the pivot component of Ensemble, the Teaching Assistant.

The main contributions are the design and implementation of a sequencing tool based on a simple sequencing model. The LTI integration is also detailed and can prove to be useful for others educators with a similar sequencing goal in heterogeneous environments.

As future work we intend to:

- Create more expositive resources (e.g. C# programming course and other courses);
- Improve sequencing model (e.g. optative - “solve N of X” and conditional - “If Then Else”);
- Improve graphical user interface (e.g. sophisticated graphical controls through Smart-GWT);
- Include a competitive facet (e.g. statistics on number of people that solved a particular exercise).

References

1 *Integration of ePortfolios in Learning Management Systems.* Springer Verlag, 2011.

2 Juan C. Burguillo. Using game theory and competition-based learning to stimulate student motivation and performance. *Comput. Educ.*, 55(2):566–575, September 2010.

3 Ruth Colvin and Clark Richard. Learning by viewing versus learning by doing : Evidence-based guidelines for principled learning environments. *Performance Improvement*, 47(9):5–13, 2008.

4 T. Gilbert. Leveraging sakai and ims lti to standardize integrations. In *10th Sakai Conference*, 2010.

5 José Paulo Leal and Ricardo Queirós. Crimsonhex: a service oriented repository of specialised learning objects. In *ICEIS 09 - 11th International Conference on Enterprise Information Systems, Milan, Italy*, volume 24 of *Lecture Notes in Business Information Processing*, pages 102–113. Springer-Verlag, LNBIP, Springer-Verlag, LNBIP, May 2009.

6 Fong lok Lee and Rex Heyworth. Problem complexity: A measure of problem difficulty in algebra by using computer, 2000.

7 Ricardo Queirós and José Paulo Leal. Petcha - a programming exercises teaching assistant. In *ACM SIGCSE 17th Anual Conference on Innovation and Technology in Computer Science Education*, Haifa, Israel, July 2012 2012. ACM.

8 Ricardo Queirós and José Paulo Leal. Pexil: Programming exercises interoperability language. Conferência - XML: Aplicações e Tecnologias Associadas (XATA), 2011.

9 Atiq Siddiqui, Mehmood Khan, and Sohail Akhtar. Supply chain simulator: A scenario-based educational tool to enhance student learning. *Comput. Educ.*, 51(1):252–261, August 2008.

10 John Sweller and Graham Cooper. The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra. *Cognition and Instruction*, 2:59–89, 1985.

11 Maarten Vansteenkiste and Edward L. Deci. Competitively contingent rewards and intrinsic motivation: Can losers remain motivated? *Motivation and Emotion*, 27:273–299, 2003. 10.1023/A:1026259005264.

12 Fu Lee Wang and Tak-Lam Wong. Designing programming exercises with computer assisted instruction. In *Proceedings of the 1st international conference on Hybrid Learning and Education*, ICHL ’08, pages 283–293, Berlin, Heidelberg, 2008. Springer-Verlag.

13 Susan Wiedenbeck, Deborah Labelle, and Vennila N. R. Kain. Factors affecting course outcomes in introductory programming. In *In 16th Annual Workshop of the Psychology of Programming Interest Group*, pages 97–109, 2004.

XML to Annotations Mapping Patterns

Milan Nosáľ and Jaroslav Porubän

Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics,
Technical University of Košice
Letná 9, 042 00, Košice, Slovakia
`milan.nosal@tuke.sk,jaroslav.poruban@tuke.sk`

Abstract

Configuration languages based on XML and source code annotations are very popular in the industry. There are situations in which there are reasons to move configuration languages from one format to the other, or to support multiple configuration languages. In such cases mappings between languages based on these formats have to be defined. Mapping can be used to support multiple configuration languages or to seamlessly move configurations from annotations to XML or vice versa. In this paper, we present XML to annotations mapping patterns that can be used to map languages from one format to the other.

1998 ACM Subject Classification D.3.4 Processors – Parsing

Keywords and phrases Mapping Patterns, Language Design, Annotations, Attribute-oriented Programming, XML

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.97

1 Introduction

Our paper concerns software system configuration metadata formats. We will use the term *software system metadata* for the total sum (everything) of what one can say about any program element, in a machine or human understandable representation. This definition is in compliance with the understanding of software system metadata in Guerra et al. [6] or Schult et al. [12].

By the association model (Duval et al. [3]) there are two types of metadata.

- **Embedded** (or internal) **metadata** are metadata that share the source file with the target data. Embedded metadata use in-place binding, they are associated with the target data by their position.
- **External metadata** are metadata that are stored in different source files than the target data. They use navigational binding where metadata include references to the target data.

Attribute-oriented programming (@OP) is a program level marking technique. This definition shared by many works in the field [8, 11] is a basis for classifying @OP as a form of embedded metadata. An annotation is a concrete mark annotating (marking) a program element.

XML on the other hand is a classic form of external metadata [4, 15, 9]. XML allows structuring metadata and storing them externally to the source code. From the point of view of the language theory, XML is a generic language that can be used to host concrete domain-specific languages [1].

These two are both widely used metadata formats used as notations for configuration languages in professional frameworks. Java EE uses both formats in many technologies such as Java Persistence API (JPA) or Enterprise Java Beans (EJB). The .NET framework extensively uses both XML formats (e.g., in the Enterprise Library) and .NET attributes (.NET attributes can be used for annotating, for example as in MyBatis.NET or Windows Communication Foundation (WCF)).

Most of the frameworks started with supporting XML as a notation for a configuration language, but after the introduction of annotations extended the configuration apparatus to support the annotations as well. Designing a good new notation for a configuration language with the same expression power in terms of supported configuration can be difficult. Mapping patterns, which can be used as a basis for designing (and thus for implementing as well) a new notation, may significantly reduce this effort.

2 Annotations and XML

In this paper, we want to present mapping patterns between this two formats. But why would anyone want to map a language in one format to the other? There are characteristics of these formats that make them complement each other. In some situations, annotations are better; in other the XML documents are advantageous.

Fernandes et al. [4] present a case study that compares three forms of configuration metadata – annotations, databases and XML. They compare these formats according to three criteria: the ability to change metadata during runtime of a system; the ability to use multiple configurations for the same program elements; and support for the definition of custom metadata. Tilevich et al. [15] compares annotations and XML in few aspects such as programmability, reusability, maintainability and understandability.

Annotations' association model and their native support in a language is the reason why Tansey et al. [14] talk about annotations as a tool for more robust and less verbose software system metadata. The XML navigational binding is more fragile during refactoring and evolution of the program than in-place binding [13, 15]. Annotations' compactness and simplicity is a consequence of native support in a language, that lowers the redundancy of structural information [10]. Since the annotations are a part of a host language, changes in annotations need recompilation. If runtime changes of configurations are a requirement, external metadata are a solution [6, 9].

The fact that annotations are scattered in the code puts a programmer into a situation when he/she needs to search whole source code to understand configuration [15, 9]. On the other hand, when examining only one program component a programmer can see the component code and configuration in one place [15].

All these arguments show that usages of both annotations and XML have their sense and meaning. Therefore, we think that there are situations when one language format is no longer the better and there is a need to port the configuration language to the other. If it is expected that there will be situations in which annotations are more adequate and also situations in which XML is better, then it is useful to support two notations for a configuration language, one annotation-based and one XML-based. This we consider the main motivation for using our mapping patterns. An interesting related work that deals with finding mapping between XML and other data format is a comprehensive discussion of XML to objects mapping by Lämmel and Meijer [7].

2.1 XML to @OP Mapping Patterns

We were dealing with the abstraction of multiple configuration sources in our previous work [9]. We have designed and implemented a tool called *Bridge To Equalia* (BTE) that facilitates unified access to @OP-based and XML-based notations for one configuration language by combining the sources into a complete configuration. One of the problems we had to deal with was finding a default mapping that would be common in existing frameworks. In this paper, we present XML to annotations mapping patterns that we identified while working on experiments with BTE.

XML to @OP mapping patterns are patterns that specify ways how an XML document or parts of it can be mapped to its counterpart in annotations (and vice versa). The following catalogue contains patterns that we have identified in experiments we have performed so far. The patterns should be found useful in the following scenarios:

- *Rewriting an existing system from one configuration format to another* – when a system author decides to change configuration notation from XML to annotation or vice versa, the patterns can help him/her to do better decisions in designing a new configuration language.
- *Adding a new configuration notation to a system* – in this situation a system supports XML or annotations but not both. A framework author wants to support a new configuration notation in order to gain benefits of supporting multiple configuration formats.
- *Building a new framework supporting multiple configuration notations* – mapping patterns can be used even when the two configuration notations are designed simultaneously from scratch.
- *Designing a mapping apparatus for a configuration abstraction tool* – in an abstraction tool there has to be way to define a mapping between supported notations; there has to be a mapping language. In this context, mapping patterns are perfect candidates for the concepts of such a language.

These situations can be abstracted and summarised by Figure 1. In short, there is a need to define mappings between annotations and XML to define configurations both with XML and annotations. If the case is that configuration is moving from one format to another, then mappings are needed to reuse domain knowledge from the old configuration language.

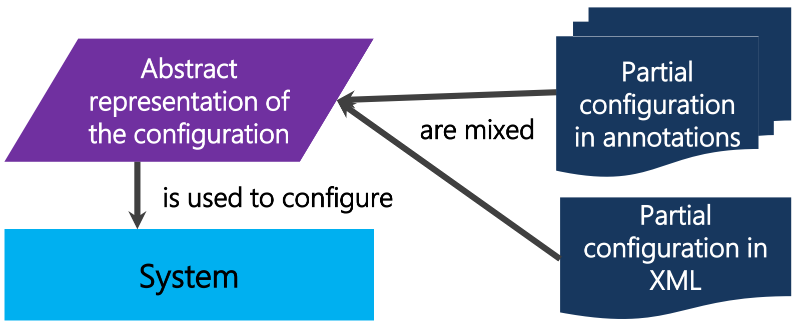

Abstract representation of the configuration
are mixed
Partial configuration in annotations
is used to configure
System
Partial configuration in XML

Figure 1 Supporting XML and annotation-based configuration.

3 Pattern Catalogue

In the next sections we introduce a catalogue of XML to annotations mapping patterns. We have divided these patterns into two groups according to their roots.

The following describes the pattern's description elements (inspired by [5]):

- The *Motivation* presents a problem context in which the pattern's solution is suitable.
- The *Problem* states the question to which a pattern provides an answer.
- *Forces* lists conflicting forces that the pattern should help balance.
- The *Solution* describes the mapping pattern.
- *Consequences* lists the possible positive and negative consequences in a pattern.
- *Known Uses* lists known uses of the described pattern.
- The *Example* illustrates the pattern usage.
- *Related Patterns* describes how the pattern interacts with the other ones from this catalog.

3.1 Structural Mapping Patterns

The first group of the patterns are simple structural patterns. They aim to show the fundamental mappings between XML and annotations. They do not deal with the relationship of a configuration to the program structure, merely with the structural mappings between the languages.

An example when these patterns can be sufficient may be global configurations that are not configurations of some program element but of a system as a whole. In case of configurations through annotations this means that the binding of configuration annotations to program elements does not have to be significant. Probably the best case would be if they could just annotate a system as a whole. In .NET framework there is an option to annotate assemblies. However, currently the Java annotations do not support annotating a whole system. Package annotations are usually used for this purpose. Another solution may be annotating an arbitrary class or a class that is somehow significant for configuration. Interesting example of such usage are configuration classes in the Spring framework.

3.1.1 Direct Mapping Pattern

Motivation. The first and basic problem when mapping a language from XML to annotation (or vice versa) is the question of how to represent language constructs from one language in another. E.g., if there is an annotation-based configuration language and the new language is supposed to be built on XML, there has to be a simple and direct way to match constructs from the first language to the second to have a starting point.

Problem. What is the simplest way to map annotations' constructs to XML and vice versa?

Forces.
- An annotation-based and an XML-based language need to be able to represent the same configuration information.
- Both languages do not have to conform to any special rules.

Solution. The most common and straightforward way is to use the Direct Mapping pattern. By default, the Direct Mapping pattern proposes to map annotation types to XML elements with the same name. Annotation type parameters are mapped to elements with the same name, too. This simple mapping is usually sufficient. From the point of view of XML an XML element is by default mapped to an annotation that has its simple name identical to the corresponding XML element's name. XML attributes are mapped to annotation type parameters of the same name.

This pattern can be parameterized with the name mapping and XML attribute/element mapping choice. Naming parameterization allows different names (in this context we can speak of keywords) for mapped constructs in both languages. It allows keeping naming conventions in both formats, identifiers starting with uppercase for Java annotation types and identifiers starting with lowercase in XML. For some language constructs in annotation-based languages it might be interesting to use XML attributes instead of elements. This concerns only marker annotations (see [16]) and annotation members that have as return type a primitive, string or a class (if its canonical name is sufficient in XML). For example, arrays are excluded since XML attributes are of simple type and there cannot be more than one attribute with the same name.

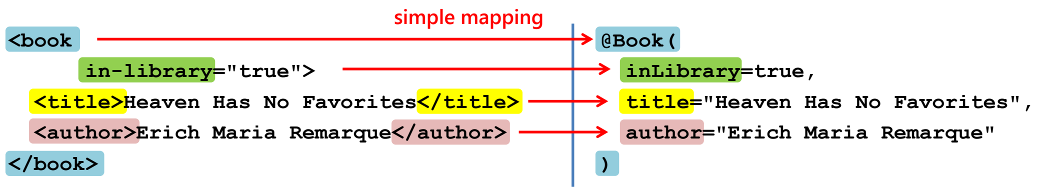
Consequences.

[+] Simple XML structures can be mapped to annotations and vice versa.
[+] Naming parameterization allows different names in annotations and in XML.
[+] In some cases element/attribute choice parameterization allows more convenient notation in XML, because attributes are less verbose than elements.
[–] More complex mappings cannot be realised merely using this pattern.

Known Uses.

- JAX-WS's mapping of the `serviceName` parameter of the `@WebService` annotation to `wsdl:service` element of WSDL language.
- JSF's mapping of the `@ManagedBean` annotation mapped to `managed-bean` element.
- JPA maps the `@Table` annotation to the `table` XML element and its `name` parameter to the XML attribute `name`.

Example. Figure 2 shows an example of the Direct Mapping pattern. A simple sentence states that there is a book *Heaven Has No Favorites* in a library. The `book` element is mapped to the `@Book` annotation, its attributes (`in-library`) and child elements (`title` and `author`) are mapped to corresponding parameters of the `@Book` annotation. The naming parameterization can be seen in mapping the `@Book` annotation to the `book` element (blue colour), where the name "book" has been capitalized in the annotation name. The attribute parameterization is present in mapping between `in-library` attribute and `inLibrary` annotation parameter (green colour).



```
<book                                               @Book(
      in-library="true">                              inLibrary=true,
  <title>Heaven Has No Favorites</title>              title="Heaven Has No Favorites",
  <author>Erich Maria Remarque</author>               author="Erich Maria Remarque"
</book>                                             )
```

Figure 2 An example of the Direct Mapping pattern.

Related Patterns. The Direct Mapping pattern proposes only mappings of keywords one to one. Usually, it is combined at least with the Nested Annotations pattern to define the simplest mappings between annotations and XML.

3.1.2 Nested Annotations Pattern

Motivation. Another structural problem of mapping is handling the XML tree structure in annotations. A tree structure of the XML is usually used to model some language property and therefore it is significant.

Problem. How to preserve XML tree structure in annotation-based languages?

Forces.

- XML allows to structure configuration information to trees of element nodes and their attributes.
- Meaning of the tree structure is significant and therefore it has to be preserved in some form in annotations.

Solution. The Nested Annotations pattern proposes to nest annotations in order to model a tree structure in annotations. The root of the tree in XML is modelled by an annotation type and its direct descendants (in XPath the children axis of the XML element) are modelled by the parameters of the annotation type. If one of the descendants has children itself, its type will be an annotation type too. This annotation type defines the children by its parameters.

Consequences.

`[+]` XML tree structures can be mapped to annotations.

`[-]` Currently annotations' implementations do not support cyclic nesting, so if the XML language has an element that can have itself as a descendant, this pattern cannot be used. XML allows modelling arbitrary trees, e.g., there can be a `node` XML element with child elements of the same type. The same is not allowed in annotations (at least not in Java or .NET attributes).

`[-]` The sequence of annotation members is not preserved during the compilation; therefore if the order is significant, this approach will fall short. The only way of preserving the order of the elements in annotations is usage of an array as an annotation parameter type.

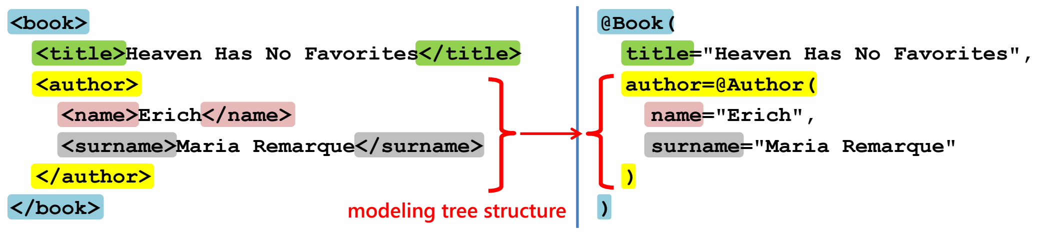
`[-]` In some programming languages, that do not support annotation nesting at all (e.g. .NET), the pattern is not applicable.

`[-]` All nested annotations inherit the target program element from the root annotation. Therefore, the modelled XML tree has to apply to the same target program element.

Known Uses.

- In JPA there is a `@Table` annotation with the parameter `uniqueConstraints` that is of the type array of `@Unique` annotations. The `@Table` is mapped to `table` element and `uniqueConstraints` are mapped to the `unique-constraint` element with its own children.
- EJB and its `@MessageDriven` annotation with the parameter `activationConfig` that is of type array of `@ActivationConfigProperty` annotations. This models a branch from XML, where the `@MessageDriven` annotation is mapped to the `message-driven` element and the parameter `activationConfig` to the `activation-config-property` element.
- Java Servlets technology uses nested annotations pattern in the `@WebFilter` annotation. Its parameter `initParams` is an array of `@WebInitParam` annotations. The `@WebFilter` is mapped to the `filter` XML element and the configuration branch represented by the `initParams` annotation parameter is mapped to an XML subtree `init-param` (or in fact multiple subtrees, since the `initParams` parameter is an array).

Example. Figure 3 shows an example of the Nested Annotations pattern. The library language from the example from the Direct Mapping pattern has been slightly changed. For simplification we have omitted the flag that specifies whether the book is in the library and decided to split the name of the author into first name and surname. In XML this is easy to do using child elements (or possibly attributes). This splitting created a new subtree (branch). In annotations it is mapped using the Nested Annotations pattern to `@Book`'s parameter `author` (yellow colour), that holds another annotation called `@Author`. `@Author` annotation deals with structuring the author's name using its own parameters.



```
<book>
  <title>Heaven Has No Favorites</title>
  <author>
    <name>Erich</name>
    <surname>Maria Remarque</surname>
  </author>
</book>
```

```
@Book(
  title="Heaven Has No Favorites",
  author=@Author(
    name="Erich",
    surname="Maria Remarque"
  )
)
```

Figure 3 An example of the Nested Annotations pattern.

Related Patterns. A closely related pattern is the Parent pattern that can be used as an alternative for mapping XML tree structures to annotations. The Parent pattern provides more expressiveness than the Nested Annotations pattern, as we argue in its description.

3.1.3 Enumeration Pattern

Motivation. Sometimes there is a piece of configuration information represented by a set of mutually exclusive marker annotations. Designing an XML language with "marker" XML elements might seem a little too verbose and it increases the complexity of XML instance documents.

Problem. How to elegantly map a set of mutually exclusive marker annotations to XML?

Forces.

- There is a configuration property that is in annotation represented by a set of mutually exclusive marker annotations.
- The Direct Mapping pattern makes the XML language too complex.

Solution. The Enumeration pattern proposes to map a set of mutually exclusive marker annotations to XML element values. Each of the marker annotations is mapped to an enumeration value.

This pattern can be extended for those that are not mutually exclusive merely by allowing more occurrences of the enumeration element. Modifications may include allowing one annotation with parameter (such as in case of JSF in the Known Uses paragraph) that is mapped to XML element's values that are not a part of the enumeration.

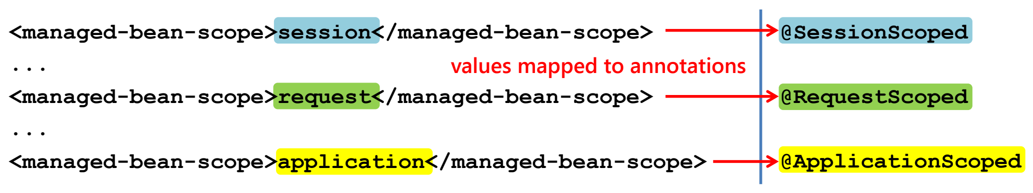
Consequences.

[+] XML language can be more compact and comprehensible.
[+] At the same time marker annotations are more compact regarding readability.
[−] Such an indirect mapping may make mixing configurations from XML and annotations more complex.

Known Uses.

- JSF technology uses this pattern to specify the scope of their managed beans. The annotations `@ApplicationScoped`, `@RequestScoped`, `@SessionScoped` and other scoping annotations from the `javax.faces.bean` package are mapped to single XML element – the `managed-bean-scope` element. It is modified pattern, because the `@CustomScoped` annotation has a parameter that is mapped to the `managed-bean-scope` value that differs from the enumeration values. While the standard cases are handled by marker annotations, specific cases are handled by using custom values.
- EJB use `@Stateless`, `@Stateful` and `@Singleton` annotations to configure session beans. These three annotations are mapped to the value of the `session-type` XML element in the deployment descriptor.

Example. Figure 4 presents a simple example of the Enumeration pattern. The example is based on the JSF managed beans technology. Annotations are mapped to the value of the `managed-bean-scope` XML element and not to the element itself.



<managed-bean-scope>session</managed-bean-scope>
...
<managed-bean-scope>request</managed-bean-scope>
...
<managed-bean-scope>application</managed-bean-scope>
values mapped to annotations
@SessionScoped
@RequestScoped
@ApplicationScoped

Figure 4 An example of the Enumeration pattern.

Related Patterns. Enumeration pattern is an alternative to the Direct Mapping pattern, but the Enumeration pattern is applicable only in special cases. The Enumeration pattern can improve code and configuration readability in a situation when in an annotation there is a set of mutually exclusive marker annotations, or vice versa, in XML there is an XML element that has values of an enumeration type.

3.1.4 Wrapper Pattern

Motivation. In some situations there may be an implicit grouping property in annotations that has to be mapped to XML. An example may be a grouping of some pieces of configuration information according to their bound target elements. In annotations, this structuring is implicit according to their usage, for example they annotate the members of the same class.

Problem. How to represent grouping in XML that is based on some implicit property of annotations?

Forces.

- Annotations use some implicit grouping property that does not have appropriate language construct that could be mapped to XML.
- Grouping is closed on branches and depth – grouping is defined strictly on one branch in a tree and it groups merely constructs on the same level in the tree.

Solution. The Wrapper pattern proposes to map implicit groupings from annotations to so called wrapper XML elements. Wrapper XML element is an element, that groups together elements according to some property. A wrapper XML element does not have its counterpart in annotations, at least not an explicit language construct like an annotation or an annotation parameter. The Wrapper pattern is usually just a notation enhancement, for example, binding to the members of the same class can be recovered from target program element specifications. The Wrapper pattern simply structurally enhances the notation in XML, for example, for design reasons.

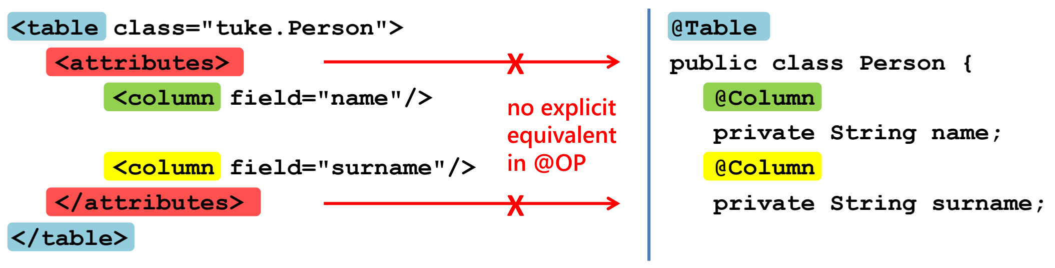
Consequences.

[+] XML can model some implicit properties of the annotations or the target program.
[+] The Wrapper pattern may increase readability of the configuration in the XML language.
[+] @OP Wrapper pattern is used to overcome the problem of annotating the program element with more annotations of the same type (so called Vectorial Annotation idiom [5]).
[-] Since the Wrapper pattern proposes a use of a language construct in one language that does not have a corresponding counterpart in the other, this may increase complexity of mixing the configuration.

Known Uses.

- JPA uses the `attributes` XML element to group column definitions in the `entity` element. The `attributes` element does not have an @OP equivalent in the code.
- JPA also uses vectorial annotation `@NamedQueries` that is a Wrapper for an array of `@NamedQuery` annotations. In XML, the `@NamedQueries` does not have an equivalent, the `named-query` XML elements (that are equivalents to `@NamedQuery`) are directly situated in the root `entity-mappings` element.
- The `@MessageDriven` annotation from EJB has a parameter that is of type array of `@ActivationConfigProperty`. While using the nested annotations pattern this would be mapped to a set of `activation-config-property` XML elements that would be direct descendants of the `message-driven` XML element, EJB uses the Wrapper pattern to wrap the `activation-config-property` elements into the `activation-config` element.

Example. Figure 5 presents an instance of the Wrapper pattern. The binding to program elements is done with the Target pattern combined with the Parent pattern (introduced in Section 3.2.2), mapping of the elements is highlighted with colours (to keep it simple the binding to program elements is not considered). The important thing here is the `attributes` XML element that has no direct equivalent in annotation-based language and is only used to wrap `column` elements.



```
<table class="tuke.Person">
    <attributes>
        <column field="name"/>

        <column field="surname"/>
    </attributes>
</table>
```

```
@Table
public class Person {
    @Column
    private String name;
    @Column
    private String surname;
```

Figure 5 An example of the Wrapper pattern.

Related Patterns. The Parent and the Nested Annotations patterns are related to this pattern. The Parent and the Nested Annotations are used to group and structure configuration information too, but they have direct and explicit representations in both languages. They carry significant configuration information. The Wrapper pattern is more of a design decision (e.g., to make the language more readable).

3.1.5 Distribution Pattern

Motivation. Sometimes the same configuration information is supposed to be distributed in one configuration language differently than in the other. There may be some configuration information that in XML is due to design reasons separated from its logical tree structure, but it still has to be somehow associated together as a logic unit. An example may be dealing with so called Fat Annotation annotations' bad smell (Correia et al. in [2]). While an XML element with many children elements might be good, overparameterized annotation might increase code complexity and reduce code readability.

Problem. How to handle different distribution of configuration information in XML and annotations?

Forces.

- Due to design decisions one or more constructs in the first language are mapped to one or more constructs in the second language, while the mapping is not straightforward.
- Distributed constructs need to be tied together to form a logical unit.
- Logical units in both languages need to be unambiguously mapped to their counterparts.

Solution. The Distribution pattern proposes to map distributed constructs by sharing a unique identifier. This identifier is unique in the configuration. The complete model for the corresponding configuration information is built up from all the constructs with the same identifier. This unique identifier may even be a target program element, as in case of mapping one XML element to more simple annotations. The unique identifier is shared between both languages to serve for mapping between logical units.

Consequences.

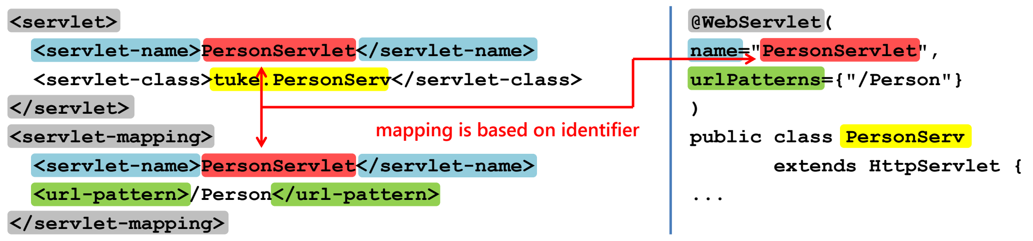
`[+]` More complicated tree structures of configuration languages are possible that do not have to strictly follow each other, and therefore there is more space to adapt the languages to good design.

`[-]` Such differences in the languages may increase the complexity and readability of the mapping itself and therefore of the configuration as well.

Known Uses.

- Java Servlets technology is a nice example (since it is quite simple, we used it in the example section). The `@WebServlet` annotation annotates a servlet implementation and through parameters specifies its name and its URL mappings. XML configuration distributes these pieces of information to the `servlet` XML element, that specifies the servlet and binds it to its implementation, and to possibly multiple `servlet-mapping` XML elements (one for each URL mapping in annotations). The `servlet-mapping`, the `servlet` elements are tied together by the servlet's name, which is servlet's identifier.
- Java Servlets use the same approach with the `@WebFilter` annotations.

Example. Figure 6 shows an example of the Distribution pattern. In this example the `PersonServ` servlet configuration information is represented by one construct in the annotations – the `@WebServlet` annotation. In the XML language the same annotation is distributed to two XML elements – the `servlet` and the `servlet-mapping` elements. These elements are bound through the servlet name in `servlet-name` elements – "PersonServlet" (highlighted in red). In the example there is again used the Target pattern (yellow colour), that is explained in Section 3.2.1.



```
<servlet>
  <servlet-name>PersonServlet</servlet-name>
  <servlet-class>tuke.PersonServ</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>PersonServlet</servlet-name>
  <url-pattern>/Person</url-pattern>
</servlet-mapping>
```

```
@WebServlet(
name="PersonServlet",
urlPatterns={"/Person"}
)
public class PersonServ
       extends HttpServlet {
...
```

Figure 6 An example of the Distribution pattern.

Related Patterns. The Target pattern binds the XML constructs to target program elements. The Distribution pattern can be used to indirectly bind scattered XML elements to program elements. The Target pattern can be used to bind one of the distributed constructs to target element, then the Distribution pattern that ties distributed constructs together propagates this binding to the rest of them.

3.2 Program Elements Binding Patterns

The structural patterns are fundamental for mapping between annotations and XML. However, in practice they are combined with program elements binding patterns. That is because most of the configuration usually directly concerns program elements. Annotations deal with this easily, because annotations by definition annotate program elements. They are a form of embedded metadata and they bind themselves to program elements using in-place binding. Annotations are written before (annotate) program elements. Metadata of an annotated program element are enriched by the metadata represented by its annotation. This of course raises a question of how to take into account this binding to program structure in an XML-based language. This group of patterns deals with this issue and shows how XML structures can be bound to their target program elements.

3.2.1 Target Pattern

Motivation. @OP is a form of embedded metadata. Annotations are always bound to program elements. The same way the configuration author wants the XML elements to be bound to program elements.

Problem. How to bind XML elements and/or attributes to program elements?

Forces.

- XML elements/attributes have to be bound to the same program elements as their corresponding annotation constructs.
- XML elements/attributes are a form of external metadata so they do not provide in-place binding of embedded metadata.

Solution. The Target pattern proposes to use a special dedicated XML element or attribute to set a target program element for an XML element. An attribute is preferred to make the notation more compact. The name of the dedicated element/attribute has to be unique in a given context for the node to be distinguishable from other nodes used to carry configuration information. By default the generic name is used, like "class" for binding elements to class definitions. By means of parameterization a special name can be used, that may suit better for the language. Another reason may be preventing name clashes if the language already defines a node with the same name that is used to represent configuration information. To make the notation shorter, the target program element is inherited in XML branch, the same way as XML namespaces are. Thus, by default a whole branch is bound to the same target program element.

A more structured approach can be used as well. The reference can be structured to more XML elements/attributes. For example, a canonical name of the class member can be split to the class name and to the simple name of the member. This can be useful when there is a need to use both of these pieces of configuration information. Then the canonical name does not have to be parsed every time configuration is read.

Consequences.

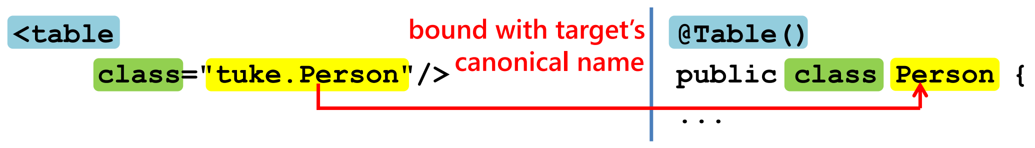
[+] XML elements and attributes can be bound to target program elements.

[-] Navigational binding using canonical name of the target program element is error-prone due to the absence of code refactoring that would take into account the composition of languages. If the programmer changes the name of a method, thanks to in-place binding, an annotation will be still valid while XML will need manual refactoring (or implementing a tool that would be able to automate it).

Known Uses.

- JSF technology uses the Target pattern for example in the `managed-bean` element to bind it to the program element, to which its counterpart – the `@ManagedBean` annotation, is bound. The `managed-bean` XML element has a child of name `managed-bean-class` that states the target program element canonical name.
- Servlets have the XML `servlet` element with a child `servlet-class` element that binds the servlet to its implementation.
- The JPA `entity` XML element is bound to its class using the `class` attribute.
- EJB uses the `resource-ref` XML element as an equivalent to the `@Resource` annotation. The `resource-ref` element is bound to its target program element by `injection-target` element, that has two descendants, `injection-target-class` specifying the target class, and `injection-target-name` identifies the actual field to be injected.

Example. Figure 7 shows an example of the Target pattern. The `table` XML element is navigationally bound to class `tuke.Person` exactly as its corresponding annotation `@Table`. The binding uses the canonical name of the class (highlighted in yellow), the pattern uses the generic name for binding the XML attribute (highlighted in green).



```
<table
      class="tuke.Person"/>
```

```
@Table()
public class Person {
...
```

Figure 7 An example of the Target pattern.

Related Patterns. The Target pattern is related to the Distribution pattern. The Target pattern proposes a way to define bindings between XML and program elements by using a program element's identifier. The Distribution pattern proposes a way to bind distributed language structures together using a custom identifier. The Distribution pattern can be used to indirectly bind distributed XML constructs to program elements by binding constructs without explicit target program elements to a construct that is already bound to a target program element.

3.2.2 Parent Pattern

Motivation. Sometimes the Nested Annotations pattern is not suitable for modelling the XML tree. In the ORM the columns of the table belong to the table. In XML this is modelled by putting the column element into the table element. Using the Nested Annotations pattern this would be modelled by a single annotation `@Table` with a member which would have the type of array of `@Column` annotations. But these annotations have a different target program element type than the @Table that annotates the class. They enhance metadata of the class members. According to the Target element pattern, in the Nested Annotations pattern, the parameters of annotations are bound to the same target program element as the annotations themselves.

Problem. How to model XML tree structure in annotations when the descendants of some element belong to a different target program element then their parent does?

Forces.
- XML allows to group elements by their meaning and to create trees of element nodes.
- Meaning of the structure is significant and therefore it has to be preserved in some form in annotations.
- Some of the XML elements or attributes in the tree belong to different target program element than the root.

Solution. The Parent pattern proposes to define parent-child relationships between annotation types. This relationship defines two roles, parent and its children. Parent-child relationships can be used to define logical tree structures consisting of annotations. Metadata carried by annotations in the children role are considered to be on the same level as the parent's members.

The matching of child annotations with their parents has to be unambiguous. Usually, a matching based on program structure axes is sufficient. For example, in the Java programming language a program structure is built from packages, classes and class members. Commonly used axis is the descendant axis, where the children annotations annotate descendants of the program element annotated by the parent annotation. A concrete example of using this axis can be found in the example section of this pattern. Another example can be the self axis. In this case all children annotations annotate the same program element as their parent. In the XML tree the new target program element of some of the descendants can be specified both using absolute name – full canonical name of the program element; or using relative name, specifying merely the identifier relative to the current target element context[1].

This pattern allows overriding the inherited target program element. In XML that is easy, just by explicitly specifying a new target element. However, annotations have to use this logical relationship to preserve the desired structure and to still properly annotate different target program elements.

[1] E.g., if the context is `tuke.Person` and a subtree has as target program element the `surname` field of the `tuke.Person` class (the `tuke.Person.surname` field), its relative name in this context is merely `surname`.

Consequences.

[+] Tree structures of XML can be mapped to annotations.

[+] Different target program elements of the configuration information in the tree are preserved in the annotation's concrete syntax.

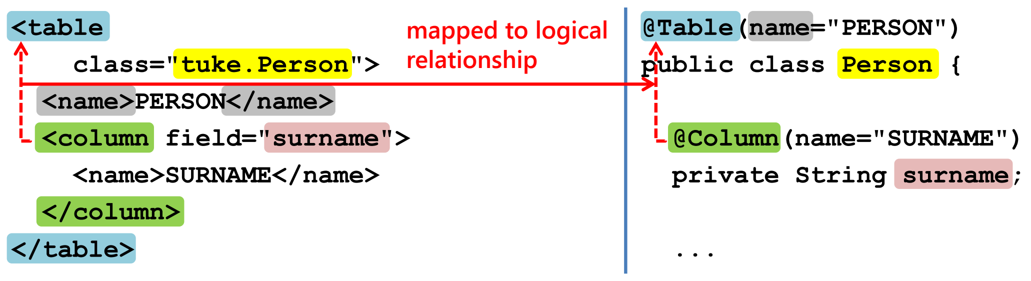
[+] In programming languages that do not support annotation nesting this pattern can substitute the nested annotations pattern (using the self axis).

[-] Using unnatural or complicated matching algorithms can make annotation languages difficult to understand and use. We believe using the direct descendant axis is easiest to understand.

Known Uses.

- JPA provides a classical example of the Parent pattern on the direct descendant program axis. The `@Entity` annotation annotates a class. The annotations `@Id`, `@Basic` and `@Column` annotate members members of the same class. In XML, their equivalents are logically structured as descendants of the `entity` element, the equivalent of the `@Entity` annotation.
- JSF uses the Parent pattern on the self axis with annotations `@ManagedBean` and scope annotations, such as `@RequestScoped` or `@SessionScoped`. Scope annotations are logical descendants of the `@ManagedBean` annotation.

Example. Figure 8 shows an example of the Parent pattern. The `@Column` annotation is on the descendant program axis of the `@Table`'s target program element. Class members are descendants of the class itself in the program structure in Java. The `@Table` is in this example a parent of the `@Column` annotation. This relationship models the structure in XML, where the `table` element is the parent of the `column` element (while the `column` have different target program elements than the `table`). The `table`'s simple descendants are mapped to `@Table`'s members using the Direct Mapping pattern.



```
<table
    class="tuke.Person">
  <name>PERSON</name>
  <column field="surname">
    <name>SURNAME</name>
  </column>
</table>
```

```
@Table(name="PERSON")
public class Person {

  @Column(name="SURNAME")
  private String surname;

  ...
```

Figure 8 An example of the Parent pattern (using relative naming).

Related Patterns. Related pattern is the Nested Annotations parent. Both of these patterns are used to model the tree structure of XML languages, but the Parent pattern is more flexible.

3.2.3 Mixing Point Pattern

Motivation. If the system supports both XML and annotations, there has to be the mechanism to recognize whether some configuration information is in both annotations and XML or merely in one of the formats. The common situation may be using an annotation-based configuration language to define a default configuration and an XML-based language to override the default configuration. In such a situation processing only one configuration format based on users choice is not sufficient, finer granularity in mixing is required than merely on the root construct level. The languages should not only be substitutable but rather be able to complement each other.

Problem. How to provide finer granularity for duplicity checking in XML and annotations configuration languages?

Forces.

- The annotation-based and XML-based configuration languages can both be used to define the configuration.
- Some pieces of the configuration information are duplicated in either XML and annotations.
- Both XML and annotations configuration are not complete, a complete configuration is a combination of both.

Solution. The Mixing Point pattern proposes to define so called mixing points in a tree structure of the languages. A node that is a mixing point has to be unambiguously identified, e.g., by its name and target program element (or another policy may be used, like merely by its name). These two properties are easily represented in both formats using the Direct Mapping pattern and the Target pattern. When a node in a tree is a mixing point, the checks for duplication on its level must be performed in both annotation and XML-based trees and the configuration information is mixed from both sources. If a node is not a mixing point, no mixing is performed and simply the configuration information is taken from the language with higher priority. The Mixing Point pattern is parameterized by the priority parameter that specifies which format has higher priority and thus overrides the duplicated configuration information. A root node is always by default a mixing point.

Consequences.

`[+]` A finer granularity of combining two partial configurations in XML and annotations.

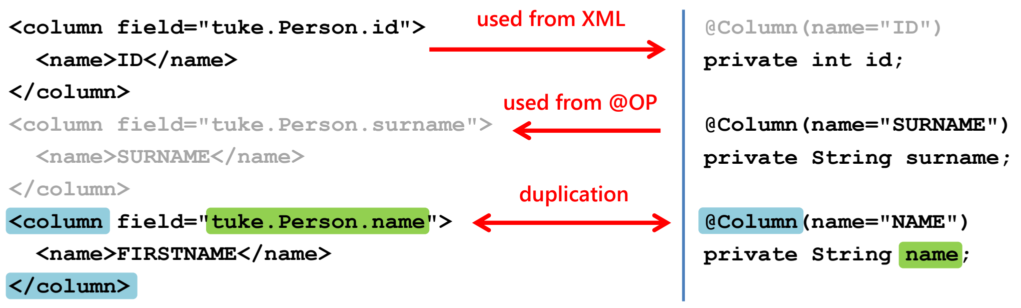
`[-]` Sometimes even this granularity might be too coarse-grained. The same name and the target element can be found in array annotation members. So if there is a need to combine two arrays without duplicates, some more adequate duplication detection mechanisms must be used (for example based on values).

Known Uses.

- The Spring framework currently supports mixing of both annotations and XML configuration, an example may be using both approaches to configure dependency injection, where the XML configuration can be complemented by annotations (`@Autowired` can be considered one of the mixing points in this example). XML has the higher priority in the Spring framework.
- Hibernate (ORM framework) supports mixing too; it allows specifying mappings in both annotations and XML. However, as mixing points there are classes and not fields, it is possible to configure one class through annotations and one through XML, but it is not possible to configure one class by mixing both formats. The annotations have higher priority in Hibernate.

Example. Figure 9 shows an example of the Mixing Point pattern. The configuration information specifing column for the field `tuke.Person.name` is duplicated. The duplication is detected using the naming parameter of the Direct Mapping pattern (`@Column` to `column` XML element – blue colour) and using the target element identifier (green colour). If there is a configuration information that is not duplicated, this information is directly used in whole configuration (configuration information for `tuke.Person.id` and `tuke.Person.surname`).

Otherwise the information from the language with higher priority is used, in our example it may be XML overriding the column value "NAME" to "FIRSTNAME".



```
<column field="tuke.Person.id">
  <name>ID</name>
</column>
<column field="tuke.Person.surname">
  <name>SURNAME</name>
</column>
<column field="tuke.Person.name">
  <name>FIRSTNAME</name>
</column>
```

```
@Column(name="ID")
private int id;

@Column(name="SURNAME")
private String surname;

@Column(name="NAME")
private String name;
```

Figure 9 An example of the Mixing Point pattern.

Related Patterns. The Distribution pattern is related to the Mixing Point pattern. While the Mixing Point pattern uses the unique identifier to find conflicts in configuration to prevent duplications, the Distribution pattern uses the identifier of the configuration information to logically bind pieces of information together.

4 Conclusion

This pattern catalogue presents common mapping patterns between XML and @OP. These patterns are the basis for designing new annotation-based or XML-based configuration languages that are based on existing configuration languages. Its contribution lies in recognizing and formalizing mapping patterns from practice. Future work can address mappings between other configuration formats – YAML, .properties, INI files or even proprietary domain-specific languages, if they are built upon generic languages.

Acknowledgements This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the Artifacts Written in Domain-specific Languages Driven by Language Evolution.

References

1 Sergej Chodarev and Ján Kollár. Language development based on the extensible host language. In *Proceedings of CSE 2012 International Scientific Conference on Computer Science and Engineering*, pages 55–62. EQUILIBRIA, s.r.o., 2012.

2 Diego A. A. Correia, Eduardo M. Guerra, Fábio F. Silveira, and Clovis T. Fernandes. Quality improvement in annotated code. *CLEI Electron. J.*, 13(2), 2010.

3 Erik Duval, Wayne Hodgins, Stuart A. Sutton, and Stuart Weibel. Metadata principles and practicalities. *D-Lib Magazine*, 8(4), 2002.

4 Clovis Fernandes, Douglas Ribeiro, Eduardo Guerra, and Emil Nakao. Xml, annotations and database: a comparative study of metadata definition strategies for frameworks. In *Proceedings of XATA 2010: XML, Associated Technologies and Applications*, XATA 2010, pages 115–126, 2010.

5 Eduardo Guerra, Menanes Cardoso, Jefferson Silva, and Clovis Fernandes. Idioms for code annotations in the java language. In *Proceedings of the 17th Latin-American Conference on Pattern Languages of Programs*, SugarLoafPLoP, pages 1–14, 2010.

6 Eduardo Guerra, Clovis Fernandes, and Fábio Fagundes Silveira. Architectural patterns for metadata-based frameworks usage. In *Proceedings of the 17th Conference on Pattern Languages of Programs*, PLoP2010, pages 1–14, 2010.

7 Ralf Lämmel and Erik Meijer. Revealing the x/o impedance mismatch: changing lead into gold. In *Proceedings of the 2006 international conference on Datatype-generic programming*, SSDGP'06, pages 285–367, Berlin, Heidelberg, 2007. Springer-Verlag.

8 Carlos Noguera and Renaud Pawlak. Aval: an extensible attribute-oriented programming validator for java: Research articles. *J. Softw. Maint. Evol.*, 19(4):253–275, July 2007.

9 Milan Nosáľ and Jaroslav Porubän. Supporting multiple configuration sources using abstraction. *Central European Journal of Computer Science*, 2(3):283–299, October 2012.

10 Renaud Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11):1–, November 2006.

11 Romain Rouvoy and Philippe Merle. Leveraging component-oriented programming with attribute-oriented programming. In *Proceedings of the 11th International ECOOP Workshop on Component-Oriented Programming*, WCOP'06. Karlsruhe University, July 2006.

12 Wolfgang Schult and Andreas Polze. Aspect-oriented programming with c# and .net. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC '02, pages 241–. IEEE Computer Society, 2002.

13 Myoungkyu Song and Eli Tilevich. Metadata invariants: checking and inferring metadata coding conventions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 694–704, Piscataway, NJ, USA, 2012. IEEE Press.

14 Wesley Tansey and Eli Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. *SIGPLAN Not.*, 43(10):295–312, October 2008.

15 Eli Tilevich and Myoungkyu Song. Reusable enterprise metadata with pattern-based structural expressions. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 25–36, New York, NY, USA, 2010. ACM.

16 Hiroshi Wada and Shingo Takada. Leveraging metamodeling and attribute-oriented programming to build a model-driven framework for domain specific languages. In *Proc. of the 8th JSSST Conference on Systems Programming and its Applications*, 2005.

Retreading Dictionaries for the 21st Century

Xavier Gómez Guinovart[1] and Alberto Simões[2]

1 **Galician Language Technology and Applications (TALG Group) Universidade de Vigo, Galiza, Spain**
`xgg@uvigo.es`

2 **Centro de Estudos Humanísticos, Universidade do Minho Campus de Gualtar, Braga, Portugal**
`ambs@ilch.uminho.pt`

Abstract

Even in the 21st century, paper dictionaries are still compiled and developed using standard word processors. Many publishing companies are, nowadays, working on converting their dictionaries into computer readable documents, so that they can be used to prepare new features, such as making them available online. Luckily, most of these publishers can pay review teams to fix and even enhance these dictionaries. Unfortunately, research institutions cannot hire that amount of workers.

In this article we present the process of retreading a Galician dictionary that was first developed and compiled using Microsoft Word. This dictionary was converted, through automatic rewriting, into a Text Encoding Initiative schema subset. This process will be detailed, and the problems found will be discussed. Given a recent normative that changed the Galician orthography, the dictionary has undergone a semi-automatic modernization process. Finally, two applications for the obtained dictionaries will be shown.

1998 ACM Subject Classification I.7.2 Document Preparation

Keywords and phrases dictionary, markup language, language processing, lexical information retrieval, Galician language

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.115

1 Introduction

Until recently lexicographers' typical training would not include information technology, and most of them would use a computer merely as an end-user, using tools such as Microsoft Word and, probably, a little of Microsoft Excel. At the same time, a lot of publishing companies did not have the tools or the mechanisms to maintain and compile a dictionary correctly, and they would propose lexicographers to use text processing tools for that task. These two factors led to the existence of dictionaries that are currently in the press, and that are only available as Microsoft Word, QuarkXPress or even PDF files. Although these formats are suitable to produce a printed document, or even to be made available for download, they are not suited to be processed automatically by computers.

An example of this situation is the "*Diccionario de sinónimos da lingua galega*," a Galician thesaurus published by *Editorial Galaxia* (Vigo) in 1997 [7]. Sixteen years after its publication, the authors wanted to provide a second revised edition of their work, in a format useful both for on-line querying and for Natural Language Processing (NLP) tasks.

Unfortunately, all the authors had was a set of Microsoft Word files that needed extra treatment to be usable by a computer program in NLP tasks, such as the extraction of synonyms, antonyms and hypernyms (as presented in [14] or [8]), or to be available as an on-line dictionary.

The solution was to develop a tool to rewrite the Microsoft Word files into a semantic-rich format. Given the existence of specific Extended Markup Language (XML) dialects for dictionaries, we decided to convert the Word files into one of these formats: the Text Encoding Initiative [15] (TEI) definition to encode dictionaries. In fact, given the detailed annotation supported by TEI, we decided to use just a subset[1] of the standard format.

After the automatic conversion of the Microsoft Word files into TEI, the dictionary was revised in a semi-automatic approach, marking the dubious parts of the document, leading the way to the human reviewing process.

Finally, the dictionary obtained is being used for two NLP projects: first, to extract synonyms for a WordNet-like structure for the Galician language (GALNET); and second, to enrich a Portuguese online dictionary (Dicionário-Aberto).

This paper is structured as follows: the next section will discuss the process of rewriting non semantic-structured documents (Microsoft Word files) into a semantic-rich format (TEI). Section 3 explains the spelling normalization of the document, updating it to the latest orthographic normative of the Galician language. Section 4 explains how the dictionary obtained will be used in two NLP projects: enriching a Portuguese on-line dictionary, and extracting concept semantic relations. Finally, we draw some conclusions about this process and the results obtained.

2 Word to TEI Conversion

This section describes the process applied to the dictionary Word files, and how they were transformed into well formatted TEI documents:

1. in the first place, Microsoft Word files were converted into very simple text files, with basic XML markup;
2. these text files were then rewritten into well formed XML files, following a TEI subset to encode dictionaries;
3. finally, a set of methods for error detection and correction is developed and applied.

2.1 Dealing with Microsoft Word Formats

The dictionary was presented as a set of Microsoft Word files, probably from Office'95, one for each letter. Fortunately, these files had very little formatting other than bold and italic markups, and the revision history. Figure 1 shows an entry as presented by Microsoft Word.

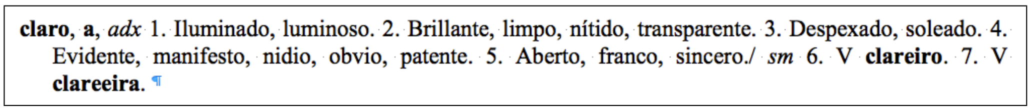

claro, **a**, *adx* 1. Iluminado, luminoso. 2. Brillante, limpo, nítido, transparente. 3. Despexado, soleado. 4. Evidente, manifesto, nidio, obvio, patente. 5. Aberto, franco, sincero./ *sm* 6. V **clareiro**. 7. V **clareeira**. ¶

Figure 1 Entry for the word *claro* as presented by Microsoft Word.

To explore and convert a Microsoft Word file is not easy. There are a number of tools for that task, but many of those just convert the Word file into other formats, such as plain text

[1] Nevertheless, the element names and their nesting rules follow the official format, making it easy for anybody familiar with the TEI format to quickly understand and process our subset.

(losing all markup). Given that we could not avoid performing the conversion, the best tool we could find to convert a Microsoft Word file was Microsoft Word itself.

The recent Microsoft Word versions can save their documents as *Microsoft Word Open XML Document* (known as `docx`). As the name shows, this format is based in XML, and therefore it should be easy to process using standard XML tools.

Although with the `docx` extension, these documents are compressed files, where a set of files (including a group of XML files) are stored in different folders. After expanding one of these documents we could find the main XML file easily, by looking at their file size. It is clearly named `document.xml`, and we went on exploring this file format.

Listing 1 shows the same entry presented above as codified by the `docx` format. There is documentation on this file format, but given the amount of details that can be present in these files, and the simplicity of our documents, we engaged into a quick exploration of the file format. For that purpose, we used a Perl module named `XML::DT` [1], which is able to create a skeleton program to transform specific XML files (that shows the specific entity tags and tag attributes that are really used in the supplied documents).

Listing 1 Entry for the word *claro* as stored in `docx` format.

```
<w:p w14:paraId="3001E63B" w14:textId="77777777" w:rsidR="006254C9"
w:rsidRDefault="00632CB3"><w:pPr><w:pStyle w:val="SINORMAL"/><w:jc
w:val="both"/><w:rPr><w:b/><w:color w:val="000000"/></w:rPr></w:pPr>
<w:r><w:rPr><w:b/><w:color w:val="000000"/></w:rPr><w:t>claro</w:t>
</w:r><w:r><w:rPr><w:color w:val="000000"/></w:rPr><w:t
xml:space="preserve">, </w:t></w:r><w:r><w:rPr><w:b/><w:color
w:val="000000"/></w:rPr><w:t>a</w:t></w:r><w:r><w:rPr><w:color
w:val="000000"/></w:rPr><w:t xml:space="preserve">, </w:t></w:r><w:r>
<w:rPr><w:i/><w:color w:val="000000"/></w:rPr><w:t
xml:space="preserve">adx </w:t></w:r><w:r><w:rPr><w:color
w:val="000000"/></w:rPr><w:t xml:space="preserve">1. Iluminado,
luminoso. 2. Brillante, limpo, íntido, transparente. 3. Despexado,
soleado. 4. Evidente, manifesto, nidio, obvio, patente. 5. Aberto,
franco, sincero./ </w:t></w:r><w:r><w:rPr><w:i/><w:color
w:val="000000"/></w:rPr><w:t>sm</w:t></w:r><w:r><w:rPr><w:color
w:val="000000"/></w:rPr><w:t xml:space="preserve"> 6. V </w:t></w:r>
<w:r><w:rPr><w:b/><w:color w:val="000000"/></w:rPr><w:t>clareiro</w:t>
</w:r><w:r><w:rPr><w:color w:val="000000"/></w:rPr><w:t
xml:space="preserve">. 7. V </w:t></w:r><w:r><w:rPr><w:b/><w:color
w:val="000000"/></w:rPr><w:t>clareeira</w:t></w:r><w:r><w:rPr><w:color
w:val="000000"/></w:rPr><w:t>.</w:t></w:r></w:p>
```

The next step was to divide the element tags present in these files into three major categories:

- **Elements to ignore:** many elements from the XML documents should be completely ignored, and the content should be discarded. Examples of these kind of elements are the revision markup, which delimits text that was deleted, hidden text, and certain meta-information;
- **Pass-through elements:** a couple of elements delimit strings, which are then annotated with different kind of information, such as whether the string is formatted in any specific way, or whether it is part of a replacement string, etc. For these elements we just return their content (which will then be formatted by the structured elements);
- **Structure elements:** these are the most interesting elements, which mark strings as

bold, italics or paragraphs. Unfortunately, and unlike other markup languages, there are different ways to set strings in bold or italics, and they all needed to be accounted for.
This categorization process was completely iterative: looking at the result obtained, comparing it with the original Microsoft Word file, and understanding the real meaning of each element. This result process is a set of paragraphs (annotated with the `entry` tag) and a number of strings annotated in bold or italics, as XML entities (as shown in Listing 2).

Listing 2 Entry for the word *claro* after the XML processing phase.

```
<entry>
<b>claro</b>, <b>a</b>, <i>adx </i>1. Iluminado, luminoso. 2.
 Brillante, limpo, íntido, transparente. 3. Despexado, soleado.
 4. Evidente, manifesto, nidio, obvio, patente. 5. Aberto, franco,
 sincero./ <i>sm</i> 6. V <b>clareiro</b>. 7. V <b>clareeira</b>.
</entry>
```

In this phase we found tagging failures due to formatting errors in the original text of the dictionary, as human editing was not always consistent. These errors had to be corrected at this stage, as their presence would have complicated the following processing steps. Examples include words that should be completely in bold but had one character that was not formatted accordingly (most at the end of the word, but we found a few cases in the middle of the word as well). Most of these errors were manually corrected in this early stage of the process.

2.2 Towards TEI: Enriching a Basic XML Format

The previously mentioned format was rewritten into a TEI subset for dictionaries. We will not discuss this subset here, but a formal definition of the structure (a Document Type Definition (DTD) file) was created, so that one could validate the rewriting process results. This rewriting approach was also implemented in Perl, using a `Text::RewriteRules` module, and applying a similar approach as the one used for Dicionário-Aberto [11]. Nevertheless, this task was harder for the Galician dictionary, as the document structure was scarcer.

The first task was to collect the abbreviations used in the dictionary in order to classify entries regarding their use or according morphological information. These abbreviations are not similar to any other words, and therefore their detection makes it easier to gain some more knowledge on the document structure. Unfortunately, not all abbreviations were correctly listed in the dictionary roll of classification terms. This lead to the manual addition of abbreviations to the list whenever we found a missing term.

Given the amount of rewriting rules, and the fact that most of them can be quite unreadable (regular expressions can be intricate), we will only explain our approach:

1. a first group of substitutions adds specific markup where there is no ambiguity about their annotation (for example, bold words at the beginning of entries that represent entries head words; or specific abbreviations that can be marked right away);
2. after that, by using the added markup, a set of rules tries to find more places where markup can be added with minimal doubt (as bold closing tags, right after the opening of the head word term);
3. this process is repeated, adding more markup, probably in more doubtful places;
4. after validating a number of the resulting TEI documents against the DTD, specific rules were added to treat specific issues and special cases.

A rule of thumb was applied: rewriting rules do not need to be 100% precise. Imagine that a specific rule has 30% of false positives. Nevertheless, the true positive cases can help other rules to be applied, and later other rules can use the extra markup that was meanwhile added to help fixing the wrong 30% cases.

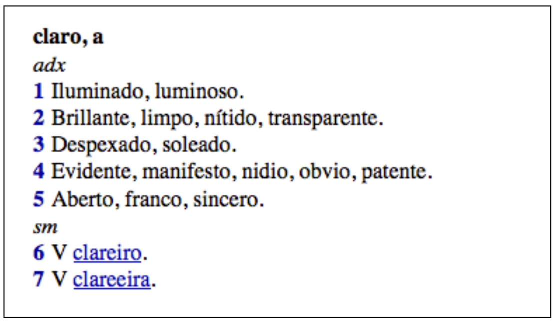
Listing 3 Entry for the word *claro* in the defined TEI subset.

```
<entry id="claro">
 <form>
  <orth>claro , a</orth>
 </form>
 <sense>
  <gramGrp>adx</gramGrp>
  <def n="1">Iluminado , luminoso . </def>
  <def n="2">Brillante , limpo , íntido , transparente . </def>
  <def n="3">Despexado , soleado . </def>
  <def n="4">Evidente , manifesto , nidio , obvio , patente . </def>
  <def n="5">Aberto , franco , sincero .</def>
 </sense>
 <sense>
  <gramGrp>sm</gramGrp>
  <def n="6">V <ref target="#clareiro">clareiro</ref>. </def>
  <def n="7">V <ref target="#clareeira">clareeira</ref>. </def>
 </sense>
</entry>
```

In the end, the XML files obtained were compliant with the previously defined DTD. Listing 3 presents a TEI-formatted entry[2].

2.3 Semi-automatic Correction of Conversion Errors

Having files that are compliant with the defined DTD is fine, but does not mean that they are semantically correct. In fact, that is far away from the truth. Taking profit of the existence of a valid DTD, it was straightforward to define a Cascading Style Sheet (CSS) to pretty print the dictionary, making it easier to browse it, and visually detect errors. Figure 2 shows an example of the CSS rendering.



claro, a
adx
1 Iluminado, luminoso.
2 Brillante, limpo, nítido, transparente.
3 Despexado, soleado.
4 Evidente, manifesto, nidio, obvio, patente.
5 Aberto, franco, sincero.
sm
6 V clareiro.
7 V clareeira.

Figure 2 Entry for the word *claro* as rendered using CSS[3].

[2] In order to enhance readability, we edited spaces and new lines in this snipped.
[3] The image suggests that the CSS rendering includes hyperlinks. Unfortunately that is not possible to obtain using CSS. For this task of visual validation the CSS fakes the link, formatting the text as if it was a valid link.

Errors from conversion are then corrected first by searching a set of error patterns, and secondly by browsing the pretty print of the dictionary. First of all, we identify the most common error patterns by an accurate human review of a number of the TEI files resultant from the automatic conversion, and elaborated a set of regular expressions for the error detection. After that, the patterns are searched in the TEI files, modifying them by applying in each case the necessary corrections in the text or in the tagging. Finally, we browse the files using the CSS rendering, searching and correcting unexpected errors remaining in the dictionary.

To illustrate this point, a common error coming from conversion is the addition of spurious blank spaces into the lemmas, that is, between their letters. The treatment of this error cannot be fully automatized, because the dictionary includes lemmas with genuine blank spaces between words. So the work on error correction after conversion has had to be designed as a regular-expression guided human task.

3 Linguistic and Spelling Normalization of Historical Variants

The official Galician orthography was introduced in 1982 and made law in 1983 by the Galician government. In July 2003 the Galician Royal Academy modified the language normative, introducing some important changes in spelling, morphology and lexicon. For the sake of the normalization of the dictionary, written in 1997 according to the normative of 1982, we designed a Perl program (in fact, a large set of regular expressions) to correct the text into the current official Galician normative established in 2003 [10, 5].

This Perl program replaces all "historical variants" of Galician words with their normative equivalent, leaving a mark which points to the type of normalization applied. Each mark implies a different human post-editing. So the "automatic normalization" performed by the Perl program is followed by a detailed post-edition human process.

The different types of automatic normalization and the different post-editing actions performed in each case are as follows:

- `[MX1]` Non-dubious morphological and lexical normalization according to [10]. This category includes more than 50 terms whose endings must be changed, or which should be completely changed, following the new normative: *catalana* > *catalá*, *diferencia* > *diferenza*, *pubertade* > *puberdade*, *anfitriona* > *anfitrioa*, *rector* > *reitor*, *servicio* > *servizo*, *pao* > *pau*, *tribu* > *tribo*, *esto* > *isto*, *nembargantes* > *porén*, *a penas* > *apenas*, *tal vez* > *talvez*, *alomenos* > *polo menos*, etc. Tagging for this type of automatic normalization substitutes the old form of the Galician term by its new normative equivalent, leaving the `[MX1]` tag on its left. Human post-editing process, for this category, is limited to supervising the possible mistakes in automatic normalization due to misspellings or unexpected homographs, and to removing the `[MX1]` tag.
  Example 1 shows this process for a TEI fragment (*sub voce* acaso). Item *a.* shows that TEI fragment in its original state previous to automatic normalization, item *b.* shows the same piece of TEI after the automatic process of normalization, and finally item *c.* shows the remaining fragment after human post-editing. The same convention is used for all examples in what follows.

(1) *s. v.* acaso

a. <def n="2">Quizabes, quizais, se cadra, seica, tal vez.</def>

b. <def n="2">Quizabes, quizais, se cadra, seica, [MX1]talvez.</def>

c. <def n="2">Quizabes, quizais, se cadra, seica, talvez.</def>

- [MX2] Morphological and lexical normalization with exceptions according to [10]. This category includes only three terms which must be changed on most occasions, but not always: *estudio* > *estudo* (unless with the meaning of room), *vocal* > *vogal* (unless relating to the voice), and *flota* > *frota* (only as noun and not as verb). The procedure used for the tagging of this category is the same as that used in the previous type: the process of automatic normalization replaces the old form of the Galician term with its new normative equivalent, leaving a [MX2] tag on its left. Human post-editing now must confirm or reverse the automatic substitution in each case, and remove the tag.

  (2) *s. v.* armada
  a. <def n="1">Escuadra, flota. </def>
  b. <def n="1">Escuadra, [MX2]frota. </def>
  c. <def n="1">Escuadra, frota. </def>

- [MX3] Morphological normalization stated in [10] which cannot be accomplished in an automatic way because it requires the gender identification of the term. This category includes two terms: *triple* > *triplo* (or *tripla*), and *cuádruple* > *cuádruplo* (or *cuádrupla*). Again, the procedure used for the tagging of this category is the same as that used in the previous types, using the masculine gender for the substitution and leaving a [MX3] tag on its left. Human post-editing now must confirm or replace the term with the feminine form, and remove the [MX3] tag.

  (3) *s. v.* trino
  a. <def n="1">Ternario, triple. </def>
  b. <def n="1">Ternario, [MX3]triplo. </def>
  c. <def n="1">Ternario, triplo. </def>

- [*CC]/[*CT] Compulsory spelling reduction (*cc* > *c*, *ct* > *t*) of consonant groups before *i*/*u* vowels, taking into account the list of exceptions stated in [10]. The tagging of this type of automatic normalization replaces the consonant group with the Galician term by its new reduced normative spelling, leaving the [*CC] or the [*CT] tag on its left in each case. Human post-editing in this category is limited to supervising the possible unexpected mistakes in automatic normalization.

  (4) *s. v.* abducción
  a. <orth>abducción</orth>
  b. <orth>abdu[*CC]ción</orth>
  c. <orth>abdución</orth>

  (5) *s. v.* aboiar
  a. <def n="1">Flotar, fluctuar. </def>
  b. <def n="1">Flotar, flu[*CT]tuar. </def>
  c. <def n="1">Flotar, flutuar. </def>

- [INI] Removing of question and exclamation initial marks according to [10]. This kind of normalization only takes place in the examples present in the entries of the dictionary. Human post-editing in this category is limited to supervising the possible unexpected mistakes.

(6) *s. v.* aviado

a. <def n="2"><lbl>fig</lbl> Amolado, apañado, fastidiado <quote>(¡estouche aviada con esta febre!)</quote>

b. <def n="2"><lbl>fig</lbl> Amolado, apañado, fastidiado <quote>([INI]estouche aviada con esta febre!)</quote>

c. <def n="2"><lbl>fig</lbl> Amolado, apañado, fastidiado <quote>(estouche aviada con esta febre!)</quote>

- `[LX1=*mistake]` Non-dubious spelling, morphological and lexical normalization stated in [5]. This category includes more than 700 terms (mostly, but not only, castilianisms and anglicisms) which are marked with an asterisk in the listing of [5] and which are changed following its normative suggestion: *almíbar > caldo de azucre*, *altavoz > altofalante*, *armonía > harmonía*, *avantaxe > vantaxe*, *basoira > vasoira*, *coruxa > curuxa*, *fumo > fume*, *obscuro > escuro*, *reptil > réptil*, *pranchar > pasar o ferro*, *prohome > home de prol*, *playback > son pregravado*, *antidóping > antidopaxe*, *feed-back > retroacción*, etc. The tagging of this type of automatic normalization replaces the wrong form of the Galician term with its normative equivalent, leaving the `[LX1]` tag on its left along with the changed term. Human post-editing in this category is focused on supervising the possible mistakes in automatic normalization due to misspellings or unexpected homographs, confirming or reversing the automatic substitution in each case.

(7) *s. v.* adianto

a. <def n="3">Avantaxe, mellora, progreso. </def>

b. <def n="3">[LX1=*Avantaxe]vantaxe, mellora, progreso. </def>

c. <def n="3">Vantaxe, mellora, progreso. </def>

- `[LX2=*mistake]` Spelling, morphological and lexical substitutions based on the asterisked terms in the listing of [5], but which have two or more normative solutions (not always clearly synonyms) in this normative reference work. This category is formed by 41 terms, and the choice for the Perl program was the solution more frequent in its usage or more general in its meaning: *bucear > mergullarse*, *carcoma > caruncho*, *inasequible > inaccesible*, *rincón > recuncho*, etc. The tagging of this type of automatic normalization replaces the wrong form of the Galician term with the selected normative equivalent, leaving the `[LX2]` tag on its left along with the term changed. As in the previous type, human post-editing in this category is focused on supervising the possible mistakes in automatic normalization due to misspellings or unexpected homographs, confirming, modifying or reversing the automatic substitution.

(8) *s. v.* abstruso, a

a. <def n="1">Escuro, inasequible, recóndito. </def>

b. <def n="1">Escuro, [LX2=*inasequible]inaccesible, recóndito. </def>

c. <def n="1">Escuro, inaccesible, recóndito. </def>

- [**LX3=correction?**] Polysemic words which need correction according to [5] but only in some of their meanings. The wrong meaning is unusual so most times it should not be corrected. For this reason, the Perl program does not perform the substitution, but only marks the term suggesting the possible corrected form and its intended meaning: *bolo > birlo* (xogo), *cru > crup* (doenza), *racha > refacho* (de vento), *tanto > punto* (no xogo), *demais > de máis* (de sobra), *berrón > verrón* (porco semental), *solar > soar* (terreo), *vencello* > *birrio* (ave), etc. In general, human post-editing in this category is limited to removing the tag. In some cases, it will be necessary to adopt the corrected form, or even providing a better one.

(9) *s. v.* abstruso, a

a. <def n="5"><lbl>fig</lbl> Cáustico, corrosivo, cru, esgueiro, ferinte, mordaz, ofensivo, punxente, punzante, sedizo.

b. <def n="5"><lbl>fig</lbl> Cáustico, corrosivo, [LXE3=crup (doenza)?]cru, esgueiro, ferinte, mordaz, ofensivo, punxente, punzante, sedizo.

c. <def n="5"><lbl>fig</lbl> Cáustico, corrosivo, cru, esgueiro, ferinte, mordaz, ofensivo, punxente, punzante, sedizo.

4 Applications: Galnet and Dicionário Aberto

The process described was performed not just to create a new and valuable resource (a Galician dictionary in a semantic-rich format) but also with some applications of this dictionary in mind.

We have two case studies where we want to take advantage of the Galician dictionary: extracting synonyms for Galnet project, and adding a new language to Dicionário-Aberto. This section will describe how we intend to perform these two ideas.

4.1 Extracting Synonyms for Galnet

The aim of the Galnet project [2] is building a WordNet for Galician aligned with the ILI (the inter-lingual index) generated from the English WordNet 3.0. WordNet [6] is a lexical knowledge base structured as a semantic network. In this lexical-semantic network, each node is a concept, and the edges which connect them are the semantic relations (hyponymy, meronymy, etc.) that are established between the concepts. Each concept in the network is represented by the group of synonymic lemmas that can express this concept. In terms of WordNet, each group of synonyms is a *synset*, and each synonym part of this group is a variant (or a lexical variation of the same concept).

Today, WordNet is, probably, the most important computational resource with lexical-semantic information, especially in the field of natural language processing (NLP), where it is used in tasks of automatic semantic disambiguation, information retrieval, automatic text classification and automatic summarization, among others.

Most of the versions of WordNet in languages other than English follow the design model of EuroWordNet [16], where *synsets* that are part of the WordNet for one of the languages, are linked to the *synsets* from other languages, through an ILI that is unique to each concept and which is mainly based on the *synsets* of the English WordNet. Therefore, the set of WordNet lexicons in different languages allows the connection between the *synsets* of any two languages via the ILI, thus constituting a very useful feature in applications of linguistic technologies which deal with multilingual processing, such as automatic translation or cross-language information retrieval.

Galnet is distributed under a Creative Commons license[4] (CC BY 3.0) as part of the Multilingual Central Repository, currently available in version 3 (MCR 3.0). The MCR 3.0 integrates the WordNet for English, Spanish, Catalan, Basque and Galician in the framework

[4] Details on the Creative Commons licenses can be found at `http://creativecommons.org/`.

of EuroWordNet. The ILI index allows the connection between words which are equivalent in different languages. The current version of ILI corresponds to the English WordNet 3.0 developed at Princeton University. The MCR also integrates the WordNet Domains, new versions of the Base Concepts and the Top Ontology, and the AdimenSUMO ontology. Thus, the MCR is a multilingual semantic resource of broad range suitable for use in language processing tasks that require large amounts of multilingual knowledge [4].

In its current state, Galnet reaches a lexical coverage of about one-fifth of the English WordNet, as shown in detail in Table 1.

Table 1 Galnet current state.

	WN30		Galnet	
	Vars	Syns	Vars	Syns
N	117798	82115	18949	14285
V	11529	13767	1416	612
Adj	21479	18156	6773	4415
Adv	4481	3621	0	0
TOTAL	155287	117659	27138	19312

The goal of the Galnet project is to reach a lexical coverage similar to the English WordNet, in order to facilitate language technologies for Galician. One of the methodologies used to extend that coverage is lexical information acquisition from human-oriented electronic dictionaries and thesaurus. In fact we have yet applied that methodology in a previous phase of the project, using the WN-Toolkit [9] to expand Galnet from two existing bilingual English-Galician resources: Wikipedia and the English-Galician CLUVI Dictionary [3].

The automatic extraction techniques applied to these two lexical resources had two distinct objectives: on one hand, expand Galnet with proper names spelled in the same way in English and Galician from the material provided by Wikipedia; on the other hand, expand Galnet with the Galician variants included in Wikipedia and in the CLUVI Dictionary as translations of English words included in the *synsets* of WordNet and not coded yet in Galnet. As for the *Diccionario de sinónimos da lingua galega*, we aim to expand Galnet with the Galician variants included as synonyms in entries whose lemma or companion synonyms are part of Galnet.

4.2 Adding Galician Definitions to Dicionário Aberto

Dicionário-Aberto[5] is a website that allows the user to query a continuously updated version of a Portuguese dictionary from 1913. The details on its construction and the main goals for this project can be read in [14, 13, 12]. In order to improve its funcionality, Dicionário-Aberto has been learning new features in the last few years.

Given the proximity between the Galician language and the Portuguese language, and the fact that researchers from these languages often look into the other language resources to check for definitions, terminological solutions, idiomatic expressions or word origins, the Dicionário-Aberto team is interested in using the Galician dictionary presented in this article to enrich the user experience:

- for each dictionary entry, show a list of possible Galician translations,
- for each translation, present the Galician dictionary entry.

[5] Available at `http://dicionario-aberto.net`.

Although the second goal is easy to achieve, the first is quite difficult, namely because there are no freely available good translation dictionaries between Portuguese and Galician, and the ones available cover only a small percentage of the domain of the dictionary (see Simões and Guinovart, this proceedings, for some work on this subject).

Taking advantage of the strong connection available between these two languages, the Galician dictionary entries will also be subject to relation extraction (synonyms, hypernyms, hyponyms, etc) that can be used to enrich the base ontology of Dicionário Aberto, and enhance the user experience when searching or browsing the dictionary.

5 Final Remarks

At the beginning of the 21st century we have a lot of useful information that cannot be used because it is encoded in paper or, more recently, in non semantic-rich formats. The definition of procedures to retread these documents into structured documents that can be easily processed by computer programs is relevant. In this paper we present the work done with a Galician dictionary stored in a Microsoft Word file, and how it was processed and converted into a structured format (Text-Encoding Initiative schema subset).

The process used for the dictionary conversion is not universal, and cannot be applied blindly to any dictionary in Microsoft Word format. Nevertheless, the approach is universal, and can be easily adapted for other dictionaries, just by adjusting the rewriting rules for the specific formatting details of the original files.

At the end of the conversion process we obtained a dictionary with more than 24571 entries (41923 meanings or groups of synonyms), written in modern Galician ortography, and annotated using a semantic-rich format, that can be easily explored for different tasks. At the moment the dictionary is not available for complete download, but its content will soon be available for querying using a simple web interface, as an independent resource, or complementing the Dicionário-Aberto dictionary of the Portuguese language.

Acknowledgments This work was partially supported by Grant TIN2012-38584-C06-04, supported by the Ministry of Economy and Competitiveness of the Spanish Government on "*Adquisición de escenarios de conocimiento a través de la lectura de textos: Desarrollo y aplicación de recursos para el procesamiento lingüístico del gallego (SKATeR-UVIGO)*"; and by the Xunta de Galicia through the "*Rede de Lexicografía (Relex)*" (Grant CN 2012/290) and the "*Rede de Tecnoloxías e análise dos datos lingüísticos*" (Grant CN 2012/179).

References

1 José João Almeida and José Carlos Ramalho. XML::DT a Perl down-translation module. In *XML-Europe'99, Granada - Espanha*, May 1999.

2 Xavier Gómez Guinovart, Xosé María Gómez Clemente, Andrea González Pereira, and Verónica Taboada Lorenzo. Galnet: WordNet 3.0 do galego. *Linguamática*, 3(1):61–67, 2011.

3 Xavier Gómez Guinovart, Alberto Álvarez Lugrís, and Eva Díaz Rodríguez. *Dicionario moderno inglés-galego*. 2.0 Editora, Ames, 2012.

4 Aitor González, Egoitz Laparra, and German Rigau. Multilingual central repository version 3.0: upgrading a very large lexical knowledge base. In *6th Global WordNetConference*, Matsue, Japan, 2012.

5 Manuel González González and Antón Santamarina Fernández. *Vocabulario Ortográfico da Lingua Galega (VOLGa).* Real Academia Galega/Instituto da Lingua Galega, A Coruña/Santiago, 2004.

6 George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. Wordnet: An on-line lexical database. *International Journal of Lexicography*, 3:235–244, 1990.

7 Camiño Noia Campos, Xosé María Gómez Clemente, and Pedro Benavente Jareño. *Diccionario de sinónimos da lingua galega.* Galaxia, Vigo, 1997.

8 Hugo Gonçalo Oliveira and Paulo Gomes. Onto.PT: automatic construction of a lexical ontology for Portuguese. In *5th European Starting AI Researcher Symposium (STAIRS 2010)*, August 2010.

9 Antoni Oliver González. WN-Toolkit: un toolkit per a la creació de wordnets a partir de diccionaris bilingües. *Linguamática*, 4(2):93–101, 2012.

10 Real Academia Galega. *Normas ortográficas e morfolóxicas do idioma galego.* Editorial Galaxia, Vigo, 2004.

11 Alberto Simões and José João Almeida. Processing XML: a rewriting system approach. In Alberto Simões, Daniela da Cruz, and José Carlos Ramalho, editors, *XATA 2010 — 8^{a} Conferência Nacional em XML, Aplicações e Tecnologias Aplicadas*, pages 27–38, Vila do Conde, May 2010.

12 Alberto Simões, José João Almeida, and Rita Farinha. Processing and extracting data from Dicionário Aberto. In Nicoletta Calzolari et al., editor, *Seventh International Conference on Language Resources and Evaluation (LREC2010)*, pages 2600–2605, Valletta, Malta, May 2010. European Language Resources Association (ELRA).

13 Alberto Simões and Rita Farinha. Dicionário Aberto: Um novo recurso para PLN. *Vice-Versa*, 16:159–171, December 2011.

14 Alberto Simões, Álvaro Iriarte Sanromán, and José João Almeida. Dicionário-aberto – a source of resources for the portuguese language processing. *Computational Processing of the Portuguese Language, Lecture Notes for Artificial Intelligence*, 7243:121–127, April 2012.

15 TEI Consortium, editor. *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, chapter 9. Dictionaries. TEI Consortium. `http://www.tei-c.org/release/doc/tei-p5-doc/en/html/DI.html` (January 2012), Version 2.0.1 edition, December, 22nd 2011.

16 Piek Vossen. Wordnet, eurowordnet and global wordnet. *Revue française de linguistique appliquée*, 7:27—-38, 1990.

Part IV

Learning Environment Languages

A Flexible Dynamic System for Automatic Grading of Programming Exercises

Daniela Fonte[1], Daniela da Cruz[1], Alda Lopes Gançarski[2], and Pedro Rangel Henriques[1]

1 **Department of Informatics, University of Minho**
Braga, Portugal
`{danielamoraisfonte,danieladacruz,pedrorangelhenriques}@gmail.com`
2 **Institute Telecom, Telecom SudParis**
Paris, France
`alda.gancarski@telecom-sudparis.eu`

Abstract

The research on programs capable to automatically grade source code has been a subject of great interest to many researchers. Automatic Grading Systems (AGS) were born to support programming courses and gained popularity due to their ability to assess, evaluate, grade and manage the students' programming exercises, saving teachers from this manual task.

This paper discusses semantic analysis techniques, and how they can be applied to improve the validation and assessment process of an AGS. We believe that the more flexible is the results assessment, the more precise is the source code grading, and better feedback is provided (improving the students learning process).

In this paper, we introduce a generic model to obtain a more flexible and fair grading process, closer to a manual one. More specifically, an extension of the traditional Dynamic Analysis concept, by performing a comparison of the output produced by a program under assessment with the expected output at a semantic level. To implement our model, we propose a Flexible Dynamic Analyzer, able to perform a semantic-similarity analysis based on our Output Semantic-Similarity Language (OSSL) that, besides specifying the output structure, allows to define how to mark partially correct answers. Our proposal is compliant with the Learning Objects standard.

1998 ACM Subject Classification K.3.2 Computer and Information Science Education, D.3.1 Programming Languages - Formal Definitions and Theory

Keywords and phrases Automatic Grading Systems, Domain Specific Languages, DSL, Dynamic Analysis

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.129

1 Introduction

When learning a new programming language, students need to solve a large number of programming exercises to practice the new language syntax and semantics. Teacher's feedback about the mistakes that they made on those exercises is crucial to improve their knowledge. However, it is hard for teachers to manually manage all the students' solutions.

The manual grading of programming exercises can involve a lot of work and be a time consuming task, since each program must be tested and its source code must be analyzed by a teacher. This task is neither simple nor mechanical: it is often a complex and arduous process, prone to faults. Different human graders may assign different evaluations to the same exercise, due to several factors like fatigue, favoritism or even inconsistency [51].

To minimize such problems, the research on tools capable to reduce the amount of work for instructors and improve the students learning experience led to the development of several *Automatic Grading Systems* (AGS), specialized on grading student's programs; they gained popularity in the field of teaching and learning programming languages [2] as Learning Support tools.

Aside their educational role, AGS are also used by the programming communities for *programming contests*. These contests can vary slightly in rules, but all of them are intended to assess the competitor skills concerning the ability to solve problems using a computer.

In a typical programming contest competitors participate in teams to solve a set of problems. For each problem, the team submits the source code of the program developed to solve the problem. Many well known programming contests in the world — such as ACM-ICPC[1] — are based on the automatic grading of the proposed solutions. This means that the submitted code will be immediately evaluated by an AGS. This process normally involves tasks like running the program over a set of predefined tests (actually a set of input data vectors), and comparing each result (the actual output produced by the submitted code) against the expected output value. Time and memory space consumptions are also usually measured during the program execution and taken into account in the final grade. This evaluation is typically complemented by the action of a human judge, who takes the final grade decision according to the specific rules for each contest.

1.1 Automatic Grading Systems as Competitive Learning Tools

Programming contests gained popularity in programming courses as a competitive learning tool in the form of exercises to stimulate the students' ability to solve practical problems in a competitive environment. An example of this was born with Mooshak[2], a system originally developed for managing programming contests [24]. Mooshak is at moment used as an e-learning tool in several universities in programming courses and is a reference tool for competitive learning in Portugal. In [26], the authors present an overview of this experience, evidencing the characteristics of competitive learning that stimulates students to work harder on problem solving using the subjects taught in each course. Students participate in several "contests" where they have to solve one or more problems, receive immediate feedback on their attempts and are able to compare their own progress with the progress of their colleagues. By providing immediate feedback to students, they are encouraged to improve their skills and to submit a new solution. The challenge associated with these competitive environments provides a meaningful way to learn and easily acquire practical skills on programming.

1.2 Assessment methodologies: Static versus Dynamic Analysis

AGS are classified concerning the methodology followed to evaluate the submitted program. This assessment can be done using two different techniques: *static* or *dynamic*.

Dynamic approaches depend on the output results, after running the submitted program with a set of predefined tests. The final grade depends on the comparison between the expected output and the output actually produced.

Static approaches take profit from the technology developed for compilers and language-based tools. Unlike dynamic analysis, this method is able to gather information about the source code without executing it.

[1] International Collegiate Programming Contest: `http://cm.baylor.edu/welcome.icpc`
[2] `http://mooshak.dcc.fc.up.pt/`

In Section 2 we detail each approach and present an overview of the existing systems that fall in each type.

1.3 Automatic Grading of Partially Correct Answers

As discussed in Section 1.2, existing dynamic-based AGS run the submitted program using a set of predefined tests and compare its output with the expected output. However, this comparison is sometimes done using a simple string comparison between the outputs, which does not allow formatting differences between them. A simple example of this situation may be a program that lists all the possible subsets of a giving set. If a specific order is not explicitly asked, different valid listing options can appear; and, of course, the formatting of the subsets can vary slightly on spacing and punctuation. However, such AGS do not support any variant of the expected output vector.

A possible solution, followed by some AGS (such as Mooshak [24]), is to allow the manual codification of a script to validate output differences. This option consumes time and effort; moreover the insertion of a new test in the test set may forces a change in that script, if it does not include all the valid options. This enables a flexibilization of the comparison process, but it is can be restricted to the programming languages supported by the AGS or by the host environment.

Consider now a situation where a student is asked to codify a program that outputs the divisors list of a given number (in ascending order). A submitted program may actually output the correct divisors, but not respect the asked order or even not output the complete list. In classrooms, a manual grading of these answers should consider them partially correct, allowing a score based on the severity level of the errors found. However, traditional AGS only grade these answers if they respect the structure defined on the comparison script — they do not support the individual evaluation of the partially correct answers.

The notion of grading partially correct answers is explored in [52], but only focused on submitted programs with *syntax* errors. In this work, the authors propose an automatic grading algorithm that combines dynamic and static marking, based on compiler theory and matching of knowledge points [52], capable to grade programs with syntax errors. However, based on the surveyed research work, there is no grading system capable of assessing programs whose produced output has *semantic* errors, regarding the expected output.

1.4 Our Contribution

We propose an output semantic-similarity based analysis that allows the comparison between the meaning of the actually produced output and the meaning of the expected output. Moreover, we aim to allow not only the specification of which parts of the generated output can differ from the expected output, but also to define how to mark partially correct answers.

More specifically, we intend to extend the traditional dynamic analysis concept by exploiting the use of a Domain Specific Language (DSL) for an output structure specification. This leads to a more flexible and fair grading process, closer to a manual one, by not restricting the output comparison process. To this end, we explore how to enrich similarity-based techniques with semantic annotations, in order to specify rules about how the outputs should be given and compared.

1.5 Article Structure

This document is organized as follows. Section 2 surveys the related work in AGS and presents the state of the art, describing their evolution in terms of the techniques applied to assess the submitted program. Section 3 is devoted to the exposition of our proposal. Section 4 closes the document with some conclusions and directions for future work.

2 Automatic Grading Systems for Program Evaluation

The earliest report about systems capable to automatically grade programs was published in 1960 in CACM by Hollingsworth [15], describing a "grader program" used to assess students in machine language at Rensselaer. This grader was completely automatic and did not require user special intervention or knowledge.

In 1965, Forsythe [11] introduced a system that follows the fundamental principle of the modern grading systems by validating the submitted solutions with a set of tests.

In 1969, BAGS [14], a system developed at University of Sydney, was used to test the submitted programs with two data sets. The system gives points for each of five activities: successful compile, complete run, data set 1 correct, data set 2 correct, and time sufficiently short. The program penalizes each extra submission after the first attempt.

Later, in 1988, Ceilidh [4] was the first computer-based assessment coursework system. Its first release only supports C language but, in 1992, a major release that supports C and C++ languages became available to all educational institutions. This version could be accessed either via a command line interface or a text based terminal menu interface. From its implementation in 1988, it had an important impact on the research and implementation of related grading systems, including CourseMarker [12], which is its direct descendant.

Kassandra [47] was developed in 1994 at ETH Zurich. It was designed to automatically mark Maple and Matlab exercises, implemented as a network service. After students submit their programs, Kassandra tests them according to two test cases and gives credit if both answers are correct. It also provides students with a complete assessment report.

With the evolution of computers, AGS increased in complexity, diversifying the tests made to the subject programs and introducing tools for monitoring the grading process. As referred, they can be distinguished according to the approach followed to evaluate the submitted program. This assessment can be done employing two major different techniques: *static* and *dynamic*. Next subsections are devoted to the analysis of this two approaches; also a more recent hybrid technique is introduced.

2.1 Dynamic Assessment

Dynamic approaches focus on the execution of the program through a set of predefined tests, comparing the generated output with the expected output (provided in the set of tests). It is the most obvious approach to verify the program correctness.

There are in the literature many systems that adopt this approach, such as Ceilidh [4], BAGS [14], TRY [40], Kassandra [47], PSGE [22], HoGG [33], Mooshak [25], JEWL [9], Quiver [8], Infandango [17], and the tools referred in [11, 15], among many others.

Some of the dynamic-based systems, such as Better Programmer[3], were developed as Web-based submission tools, where users can exercise and evaluate their programming skills by picking-up a problem from their repository, coding a solution and submitting

[3] http://www.betterprogrammer.com

it for assessment. This concept is also largely explored by several universities over the world to support automatic grading of their programming courses, encouraging the so called Competitive Learning (CL). Examples of CL projects are Practice-It[4], Marmoset[5], CodingBat[6], UVa Online Judge[7], CodeLab[8] and CodeWrite[9], among others.

Dynamic approaches are usually based in a simple string comparison between the expected output and the output actually produced to determine if both values are equal. Thus, the submitted program is considered correct if and only if this condition is true, which can be a limitation. Besides this, another important drawback of this approach is the incapability to produce an assessment when a program does not successfully compile, does not produce an output or does not end its execution (infinite loop).

2.2 Static Assessment

Unlike *Dynamic approaches*, which are incapable of analyzing the way source code is written, *Static approaches* take benefit from the technology developed for compilers and language-based tools, to gather information about the programming code without executing it. They are supported by source code analysis, which allows to detect situations where the submitted solution does not comply with the exercise rules.

As an example, consider a typical C programming exercise that asks the student to implement a Graph using *adjacency lists* to print the shortest-path between two given nodes. If the final output is equal to the expected one, Dynamic AGS will consider correct a solution implemented with an *adjacency matrix*. However, this solution is not acceptable because it does not satisfy all the assignment requirements – a static approach can be useful to help detecting the used data types. Or, even more dramatic, if the user computes by hand the shortest-path and the submitted program only prints it, the solution is also accepted using a dynamic approach, because it is not able detect such implementation faults.

The most popular method used in this approach is based on software metrics analysis. Metrics, such as lines of code, number of variables, statements and expressions or even the code complexity are used as the program grading base. They are easy to calculate, though the semantics of a program can not be analyzed. Examples of such systems are STYLE [23], Knots [18], CAP [43], Style Checker [32] and Verilog Logiscope [30].

Besides software metrics, there are other techniques that fit on static analysis approach such as the programming style assessment [1], syntax and semantics errors detection [45, 16], structural similarity analysis [3, 46, 34, 49, 51], non-structural similarity analysis [50], keyword detection [42] or even plagiarism detection [35], allowing static analysis to be a powerful approach to evaluate how well source code is written.

2.3 Hybrid Assessment

Static approaches can not be used for testing the correctness of programs with input and output operations. However, traditional dynamic grading systems leave aside one important aspect when assessing programming skills: the source code quality. These assumptions

[4] http://Webster.cs.washington.edu:8080/practiceit/
[5] http://marmoset.cs.umd.edu/
[6] http://codingbat.com/
[7] http://uva.onlinejudge.org
[8] http://turingscraft.com/
[9] http://codewrite.cs.auckland.ac.nz/

led to the construction of systems such as Web-CAT [44, 7], WebBot [5], Scheme-Robe [42] or BOSS[10], that combine the best of both approaches, by improving the dynamic testing mechanism with static techniques like metrics or style analysis. This symbiosis keeps providing immediate feedback to the users (students/competitors or instructors/judges), but enriched by a quality analysis – which is obviously a relevant extra-value.

The first system combining both approaches was Ceilidh [13] by introducing semantic error detection. It statically detects suspicious never-ending loops, a crucial feature to avoid breaks during the dynamic evaluation process. This system completes the dynamic analysis with a static verification of the program layout and structure – its indentation, identifiers, comments; it also measures readability and complexity metrics.

Another example is ASSYST [20], used to automate some aspects of grading in introductory Ada classes, as well as a second-year C-programming course. It gives weighted grades to students, based on the correctness (output), efficiency (run time), style and complexity of their answers, and also based on the adequacy of the submitted tests (student self-test data).

Used in Java introductory programming courses, eGrader [3] produces detailed feedback reports, showing to students the model solution provided by the teacher. Additionally, specific comments on syntax and semantic errors (if any) are also provided. Its static analysis process consists of two parts: the structural similarity, which is based on the graph representation of the program; and the quality analysis, which is measured by software metrics.

A more recent system, AutoLEP [48], improves the traditional static grading mechanisms with dynamic code testing, by enriching source code static analysis with a comparison of the similarity degree. Summing up, it evaluates the program construction and how close the source code is from the correct solution.

Another example of hybrid assessment is Quimera [10], a Web-based application able to evaluate source code written in C language, which provides a full management system for programming contests. Quimera allows to create and manage programming exercises both in competitive learning and programming contest environments. Besides the traditional dynamic approach, this system provides a static analysis of the program by measuring the source code quality. Thus, the final grade is based not only in the source code capability of producing the expected output, but also on its quality and accuracy.

3 Flexible Dynamic Analysis

Our proposal, a Flexible Dynamic Analyzer (FDA), is based on the traditional dynamic analysis which is, as referred, supported by the execution of the submitted answer (the program under assessment) over a set of predefined tests. We aim at extending this concept to allow a more flexible comparison between the output produced by the program under assessment and the expected output.

We propose to compare the meaning of both outputs performing a semantic-similarity analysis to achieve a more flexible grading process. A Domain Specific Language (DSL), specially designed to specify the output structure and semantics, will be used as the basis for the desired semantic comparison. The DSL's design will also support the mark of partially correct answers.

Next subsections are devoted to detail this proposal, a flexible system able to interpret the output meaning, concerning a predefined structure. This system produces a complete grading report, designed to be easily integrated with a traditional dynamic analyzer. Section 3.1

[10] `http://www.dcs.warwick.ac.uk/boss/about.php`

presents the proposed architecture. Section 3.2 introduces the Output Semantic-Similarity Language (OSSL), the proposed DSL used by the FDA. OSSL grammar is also presented and discussed; the subsection ends with illustrative examples.

3.1 Architecture

Our FDA was designed as an independent module that can be easily integrated with any AGS that uses an external evaluator. We just require that the AGS compiles the submitted source code, and provides a compilation report and (if the compilation is successful) the compiled program.

Moreover, to follow the trend of the evolution of systems that perform the automatic assessment of programming exercises [6], as well as to ensure the interoperability with other systems [36], the FDA uses the concept of Learning Object (LO) to describe a programming exercise, its assessment instructions and the associated resources. LOs are content components, organized in context independent and reusable digital pieces of information – a standard in the learning domain [37]. Since the standard LO cannot be used for complex evaluation domains such as programming exercise evaluation [28], we propose an extension of the Programming Exercises Interoperability Language (PExIL) [38] to include the OSSL description of the set of tests. PExIL aims at consolidating all the data required to cover the programming exercise life-cycle, since it is created until it is graded. The associated PexilUtils generator produces a IMS CC[11] LO package, allowing the definition of Specialized Learning Objects. The use of LOs components also ensures compatibility with specialized LOs Repositories such as CrimsonHex [27], allowing the reuse of programming exercises among different systems.

Therefore, the FDA architecture is composed of three modules: the OSSL Processor, the Flexible Evaluator and the Grader, as depicted in Figure 1.

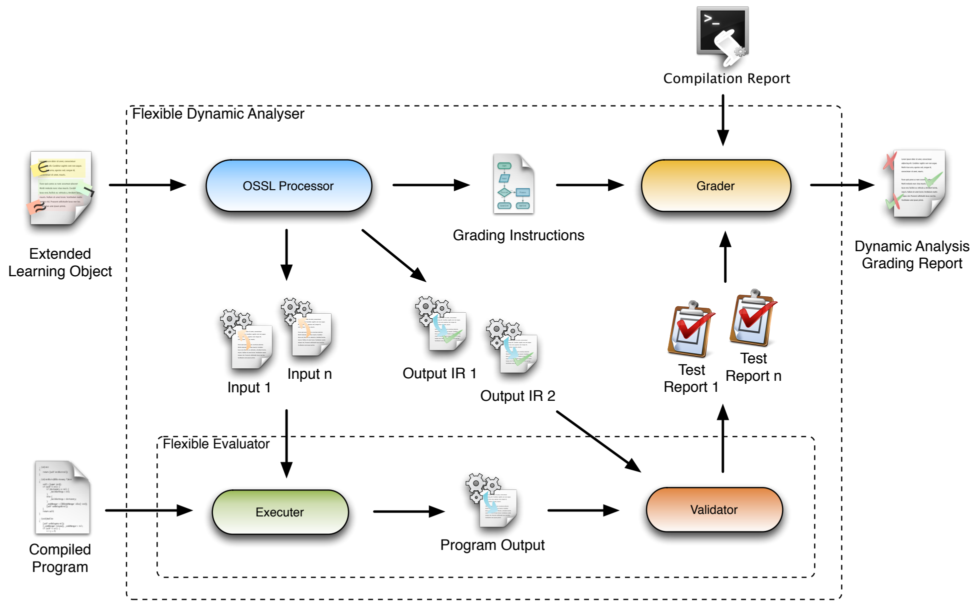


Figure 1 Flexible Dynamic Analyzer Architecture.

[11] A package standard that assembles educational resources and publishes them as reusable packages.

The OSSL Processor is the central piece of the FDA, being the responsible for producing the set of resources required to execute, validate and grade the submission under assessment. It receives an Extended Learning Object containing the problem description, the associated metadata and the OSSL specification, and generates the set of Inputs, the set of the Expected Outputs (through an intermediate representation) and the Grading Instructions.

The Flexible Evaluator is responsible for the execution and validation of the submissions. It receives the set of Inputs, extracted from the OSSL specification by the OSSL Processor, and executes the Compiled Program. If an execution is successful, the Flexible Evaluator module produces a Program Output file that is validated against the respective Output IR – the intermediate representation of the OSSL specification of the expected output, generated by the OSSL Processor. This output intermediate representation allows to compare (at the semantics level) the meaning of the expected output with the output actually produced. This validation process produces a Test Report for each test performed, containing the details about time and memory consumptions and the test results.

The Grader module produces a Grading Report resulting from the dynamic evaluation performed, concerning the set of Test Reports produced by the Flexible Evaluator and the Grading Instructions provided by the OSSL Processor. This Grading Report is composed of the details about each individual test report and the submission assessment, which is calculated regarding time and memory consumptions, the weight and score for each test and the number of successful tests. Moreover, if the submission under assessment fails the compilation phase, this grading process is based on the Compilation Report provided by the compiler, in order to give feedback about the program under assessment.

3.2 OSSL: Output Semantic-Similarity Language

We believe that the development of a DSL is the most flexible approach for: (i) the extension of an AGS dynamic analyzer to interpret different output values with the same meaning; and (ii) to support partial grading. DSLs are programming languages adapted to a specific application domain, which offer substantial gains in expressiveness and ease of use, when compared with general-purpose programming languages [31]. Rather than being for a general purpose, a DSL captures the *semantics* of its domain. Examples of DSLs include *lex* [29] and *yacc* [21], used for program lexical analysis and parsing, HTML [39], used for document markup, or even VHDL [19], used for hardware descriptions. Concerning the DSLs scope and our goals, we propose a DSL that, given a programming exercise, allows to define:

- The program output meaning;
- Partially correct solutions and their penalties;
- Mandatory and optional output components;
- The support of case sensitive text;
- The delimiter and punctuation characters used to produce the output;
- Output patterns.

As discussed along Section 1, the grade of programing exercises is a complex task that usually involves executing a program to assess its ability to produce the expected output concerning the given input. The OSSL language aims at supporting the output structure specification, in order to allow an easy and clear way of describing the instructions for the automatic grading of the output produced by the program under evaluation. This allows the interpretation of the output meaning and perform its grading, based on a semantic-similarity specification strategy. This description follows the traditional manual assessment process, determining partially correct answers and their respective grade.

To be able to automatically interpret the meaning of a program output, it is essential to categorize in a simple but formal way the format used to model that output. To represent the abstraction of the possible output values, we divide them into two main types: the *atomic* and the *compound* values. The *Atomic* category includes numeric, character, identifier and string values. The other category – *Compound values* – represents tuples, unions, sets, sequences, trees, graphs and mappings. Concerning this categorization, we propose the grammar in Listing 1.

Listing 1 OSSL Grammar Core.

```
Output   -> Value
Value    -> Atomic  |  Compound
Atomic   -> number | character | identifier | string
Compound -> Tuple | Union | Set | Seq | Mapping | Tree | Graph
```

An *Output* is defined by its *Value*, which can be an *Atomic* or a *Compound* value. As referred, *Atomic* values can represent *numbers*, *characters*, *identifier* or *strings*. *Tuples*, *Unions*, *Sets* and *Sequences* are represented by lists of elements that are *Values*. A *Mapping* is composed of a *Key* and an associated *Value*. *Trees* are represented by their *Root*, composed of an *Atomic* value and its *Descendants*. *Graphs* are represented by a list of arcs, being each arc a triple composed of source and destination *Nodes* and its *Weight*. Since this is a recursive definition, the elements of a *Compound* value can be atomic or compound.

In order to allow the specification of partially correct answers and also the association of a grade, the grammar axiom *Output* was redefined as can be seen in Listing 2.

Listing 2 OSSL Grammar extension for support partially correct answers.

```
Output -> Correct PartiallyCorrect
Correct -> Value Grade
PartiallyCorrect -> (Value Percent) *
```

An *Output* is composed of a *Correct* answer and a list of *Partially Correct* answers (if any). *Correct* answers have an associated *Grade*. This grade is the base of the *Partially Correct* answers assessment, which is a *Percent* of the correct answer grade.

To enable the automatization of each test, a new axiom was defined, including the specification of the input value, as can be seen in Listing 3.

Listing 3 OSSL Grammar support for test automatization.

```
Test -> Input Output
Input -> Atomic +
```

In order to permit the definition of all the tests in the same specification, the grammar is extended again with a new axiom and a new production, shown in Listing 4.

Listing 4 OSSL Grammar support for a set of tests.

```
TestSet -> Test +
```

To define completely OSSL, the abstract grammar presented above shall be transform into a concrete one, adding some syntactic sugar. Listing 5 shows our final choice. In the final version of the grammar, a new axiom was introduced, *Ossl*, composed of two elements. The *Header*, that allows to identify the problem, define the number of tests included and the total grading of the set of tests. The *TestSet* represents all the tests (the pairs of input/output descriptions). Notice that the sum of the grade corresponding to each output must be equal to the *Total* value defined in the *Header*.

Listing 5 OSSL Grammar.

```
Ossl -> Header TestSet
Header -> PROBLEM ":" identifier TESTS ":" number TOTAL ":" number
TestSet  -> Test +
Test     -> Input Output
Input    -> INPUT ":" Value
Output   -> OUTPUT ":" Correct PartiallyCorrect
Correct  -> Value "(" Grade ")"
PartiallyCorrect -> ( ALSO Value "(" Percent ")" )*
Value    -> Atomic | Compound
Atomic   -> number | character | identifier | string
Compound -> TUPLE Elems | Pair
           | UNION Elems | MAP Entries
           | SET Elems | SEQ  Elems
           | TREE Root | GRAPH Arc +
Elems    -> "<" Values ">"
Entries  -> Entry +
Entry    -> Key "->" Value
Values   -> Value ("," Value)*
Key      -> Atomic
Root     -> Node Descs
Node     -> Atomic
Descs    -> Root *
Percent  -> number
Grade    -> number
Arc      -> "(" Node "," Node Weight ")"
Weight   -> & | "," Atomic
Pair     -> "(" Value "," Value ")"
```

Let us now introduce some examples of OSSL grammar usage, presenting the OSSL specification for two simple programming exercises.

Example 1

Consider the following problem statement: *Given a positive integer, compute:*

a) *The sequence of its divisors, in ascending order;*
b) *The set of its divisors.*

Consider now that the set of tests defined for the proposed problem statement is composed of two tests, the first one with the number 10 as input, and the second one with the number 18. The correct divisors are 1, 2, 5 and 10 for the first test, and 1, 2, 3, 6, 9 and 18 for the second test. Concerning question a), the correct answer would be the sequence of the respective divisors. Using OSSL language, the set of tests and their corresponding grades would be defined as described in Listing 6.

When comparing the expected output with the effectively produced one, the FDA compares each value according to the defined order. Thereby, this specification ensures that only a sequence of the correspondent divisors will be considered a correct answer. For instance, the sequence <1, 2, 10, 5> is not accepted as a correct answer.

Consider now the question b). The set of tests is defined in OSSL language as described in Listing 7.

Listing 6 OSSL definition for question 1b).

```
PROBLEM: SeqDivisors TESTS: 2 TOTAL:3

INPUT: 10
OUTPUT: SEQ <1,2,5,10>  (1)

INPUT: 18
OUTPUT: SEQ <1,2,3,6,9,18> (2)
```

Listing 7 OSSL definition for the concerning question.

```
PROBLEM: SetDivisors TESTS: 2 TOTAL:3

INPUT: 10
OUTPUT: SET <1,2,5,10>  (1)

INPUT: 18
OUTPUT: SET <1,2,3,6,9,18> (2)
```

This definition ensures that any combination of the specified numbers is considered a correct answer – the FDA will verify, for each value of the produced output, if it is a member of the accepted set. So, <3, 18, 9, 2, 6, 1> is one of the possible correct answers for the second test (input 18).

Consider again question a). Listing 8 illustrates how to specify that incomplete sequences, missing their extreme values, should be accept as partially correct answers.

Listing 8 OSSL definition with partially correct answers.

```
PROBLEM: SetDivisors TESTS: 2 TOTAL:3

INPUT: 10
OUTPUT: SEQ <1,2,5,10>  (1)
        ALSO SEQ <2,5,10> (0.5)
        ALSO SEQ <1,2,5> (0.5)

INPUT: 18
OUTPUT: SEQ <1,2,3,6,9,18> (2)
        ALSO SEQ <2,3,6,9,18> (0.5)
        ALSO SEQ <1,2,3,6,9> (0.5)
```

The OSSL specification in Listing 8 allows to accept answers where the first or the last value of the correct sequence is not outputted. In such situations, the final grade will be 50% of the total grade.

Example 2

Consider now the following problem statement: *Write a program that allows to find all the possible paths to solve a given maze. A maze is represented through a 6 x 5 matrix, where the Lines are numbered from 1 to 6 and the Columns are identified from A to E, as depicted in Figure 2. Each cell represents a Position in the maze, expressed by a pair (Line, Column) (e.g., (1,A) represents the first cell of the maze, on its left top corner).*

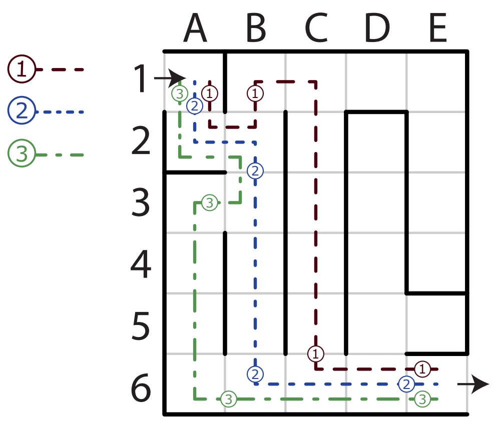


Figure 2 The three possible paths that solve the maze.

The maze Walls are represented by a list of Coordinates indicating the positions where each wall is in the maze. Each Coordinate is represented by a tuple (Position, Limit), where the Limit represents the side of the cell where the wall is located. The different positions are represented according to the following notation convention: Left (L), Top (T), Right (R) and Bottom (B) (e.g., ((1,A),R) represents a wall on the right side of cell (1,A)).

The program receives three parameters: the start Position, the end Position, and the Walls list, and will output the set of the correspondent possible paths.

Listing 9 represents the OSSL specification for the given problem, concerning the maze definitions depicted in Figure 2. The input is defined by a tuple with the start position and the end position, followed by the wall list of the maze. The output is composed of a set of Position sequences, representing each of the three possible paths. It is also considered that, if the program produces two of the three possible paths, it will receive 50% of the total input grade. Moreover, if the program only outputs one of the three possible paths, it will receive 25% of the total input grade.

4 Conclusion

Along this document, the problem of automatically grading the solutions submitted by students to programming exercises was introduced and characterized. This contextualization gave the motivation for the research topic of this work: improve a traditional dynamic grading system with the ability to interpret the meaning of the output, instead of a strict syntactic comparison. Moreover, the capability of marking partially corrected solutions was also considered. The deep study of the state of the art on AGS has shown that there is no other system supporting both requirements that we consider crucial for the successful use of such systems in learning environments.

We proposed an architecture for the Flexible Dynamic Analyzer (FDA) module to achieve the identified objectives. Also, we proposed a DSL, named OSSL, to support the output semantic specification. OSSL grammar was presented in the paper, and its use illustrated.

We strongly believe that the proposed approach is user-friendly (OSSL allows to specify the output meaning in a simple way) and is easy to implement. We also argue that it effectively improves the role of AGS as Learning Support Tools, ensuring the interoperability with existent programming exercise evaluation systems that support Learning Objects.

As this is an undergoing project, obviously the future work is concerned with the proposal

Listing 9 OSSL definition for the maze exercise.

```
PROBLEM: FindPaths TESTS: 1 TOTAL: 4

INPUT: TUPLE < (1,A) , (6,E),
              SET  < ((1,A),R),((2,A),B),((2,B),R),((2,C),R),
                     ((2,D),T),((2,E),R),((3,B),R),((3,C),R),
                     ((3,D),R),((4,A),R),((4,B),R),((4,C),R),
                     ((4,D),R),((4,E),B),((5,A),R),((5,B),R),
                     ((5,C),R),((5,E),R) > >

OUTPUT: SET < SEQ < (1,A),(2,A),(2,B),(1,B),(1,C),(2,C),
                     (3,C),(4,C),(5,C),(6,C),(6,D),(6,E) >,
               SEQ < (1,A),(2,A),(2,B),(3,B),(4,B),(5,B),
                     (6,B),(6,C),(5,C),(6,C),(6,D),(6,E) >,
               SEQ < (1,A),(2,A),(2,B),(3,B),(3,A),(4,A),
                     (5,A),(6,A),(6,B),(6,C),(6,D),(6,E) > > (4)
       ALSO SET < SEQ < (1,A),(2,A),(2,B),(1,B),(1,C),(2,C),
                         (3,C),(4,C),(5,C),(6,C),(6,D),(6,E) >,
                   SEQ < (1,A),(2,A),(2,B),(3,B),(4,B),(5,B),
                         (6,B),(6,C),(5,C),(6,C),(6,D),(6,E) > > (0.5)
       ALSO SET < SEQ < (1,A),(2,A),(2,B),(1,B),(1,C),(2,C),
                         (3,C),(4,C),(5,C),(6,C),(6,D),(6,E) >,
                   SEQ < (1,A),(2,A),(2,B),(3,B),(3,A),(4,A),
                         (5,A),(6,A),(6,B),(6,C),(6,D),(6,E) > > (0.5)
       ALSO SET < SEQ < (1,A),(2,A),(2,B),(3,B),(4,B),(5,B),
                         (6,B),(6,C),(5,C),(6,C),(6,D),(6,E)>,
                   SEQ < (1,A),(2,A),(2,B),(3,B),(3,A),(4,A),
                         (5,A),(6,A),(6,B),(6,C),(6,D),(6,E) > > (0.5)
       ALSO SEQ < (1,A),(2,A),(2,B),(1,B),(1,C),(2,C),
                   (3,C),(4,C),(5,C),(6,C),(6,D),(6,E) > (0.25)
       ALSO SEQ < (1,A),(2,A),(2,B),(3,B),(4,B),(5,B),
                   (6,B),(6,C),(5,C),(6,C),(6,D),(6,E) > (0.25)
       ALSO SEQ < (1,A),(2,A),(2,B),(3,B),(3,A) (4,A),
                   (5,A),(6,A),(6,B),(6,C),(6,D),(6,E) > (0.25)
```

implementation. The example presented in Listing 9 illustrates well on how partially correct answers can be mathematically seen. When the expected output is a *set* of values, we can see the associated partially correct answers as *subsets* of the correct set. We believe that it could be a benefit to extend the OSSL grammar to allow not only *subset* definitions, but also to support ranges and other subtype definitions like subsequences or subgraphs, concerning the compound values currently supported and their meaning in terms of partially correct output definition. Besides that, this example also shows what is the main benefit of the proposed output typification: support the definition of output patterns for describing both correct and partially correct answers. These extensions will allow a simpler definition of the correct answers and improve the readability of the output definition.

Moreover, and regarding the literature studied [41, 36], the support for automatic test data generation is not a closed option in the future. The proposed architecture is able to support it by exploiting PExIL functionalities and with the implementation of an OSSL generator. However, in this initial phase of the project, it is irrelevant how the set of tests is defined – it is not the focus of this paper. A manual definition of the set of tests will not interfere with the OSSL language and its main features.

After extending OSSL, we will use Quimera [10] system to integrate and test the FDA. As soon as the new system is available, we intend to test it with real users. We plan to design and implement an experiment in real learning environments to assess the usability and performance of the proposed system. This experiment will also allow us to evaluate the benefits of the learning approach defended along this document.

Acknowledgements The authors are in debt to the anonymous Referees for their valuable comments that have clearly contribute for the progress of our proposal as well as for the improvement of the paper. We also acknowledge the numerous fruitful discussions with Nuno Oliveira and Ismael Vilas Boas – Quimera's co-author and a permanent project contributor.

References

1 Kirsti Ala-mutka, Toni Uimonen, and Hannu matti Järvinen. Supporting students in C++ programming courses with automatic program style assessment. *Journal of Information Technology Education*, 3:245–262, 2004.

2 Kirsti M. Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102, 2005.

3 F. AlShamsi and A. Elnagar. An automated assessment and reporting tool for introductory Java programs. In *Innovations in Information Technology (IIT), 2011 International Conference on*, pages 324 –329, april 2011.

4 S D Benford, E K Burke, E Foxley, and C A Higgins. The Ceilidh system for the automatic grading of students on programming courses. In *Proceedings of the 33rd annual on Southeast regional conference*, ACM-SE 33, pages 176–182, New York, NY, USA, 1995. ACM.

5 Don Colton, Leslie Fife, and Andrew Thompson. A Web-based Automatic Program Grader. *Information Systems Education Journal (ISEDJ)*, 4(114), November 2006.

6 Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), September 2005.

7 Stephen H. Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.*, 3(3), September 2003.

8 Christopher C. Ellsworth, James B. Fenwick, Jr., and Barry L. Kurtz. The Quiver system. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 205–209, New York, NY, USA, 2004. ACM.

9 J. English. Automated assessment of GUI programs using JEWL. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, page 137–141, Leeds, United Kingdom, 2004. ACM.

10 Daniela Fonte, Ismael Vilas Boas, Daniela da Cruz, Alda Lopes Gançarski, and Pedro Rangel Henriques. Program analysis and evaluation using quimera. In *ICEIS'2012 — 14th International Conference on Enterprise Information Systems*, pages 209–219. INSTICC, June 2012.

11 G. E. Forsythe and N. Wirth. Automatic Grading Programs. Technical report, Stanford University, 1965.

12 E. Foxley, C. Higgins, T. Hegazy, P. Symeonidis, and A. Tsintsifas. The CourseMaster CBA System: Improvements over Ceilidh. *Fifth International Computer Assisted Assessment Conference*, 2001.

13 Eric Foxley, Colin Higgins, Edmund Burke, and Cleveland Gibbon. The Ceilidh system. *Asian Technology Conference in Mathematics*, pages 430–441, 1997.

14 J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5):272–275, May 1969.

15 Jack Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, October 1960.

16 Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, SIGCSE '03, pages 153–156, New York, USA, 2003. ACM.

17 Mike Hukk, Dan Powell, and Ewan Klein. Infandango: Automated Greding for Student Programming. In *ITiCSE 2011*, page 330, Darmstadt, Germany, 2011. Association for Computing Machinery.

18 S. Hung, L. Kwok, and R. Chan. Automatic program assessment. *Computers and Education*, 20(2):183–190, 1993.

19 IEEE. IEEE standard VHDL language reference manual. *IEEE Std 1076-1987*, 1988.

20 David Jackson and Michelle Usher. Grading student programs using ASSYST. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, SIGCSE '97, pages 335–339, New York, NY, USA, 1997. ACM.

21 Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, , 1975.

22 Edward L. Jones. Grading student programs - a software testing approach. In *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, pages 185–192, USA, 2000. Consortium for Computing Sciences in Colleges.

23 Al Lake and Curtis Cook. Style: an automated program style analyzer. *SIGCSE Bull*, 22(3):29–33, August 1990.

24 José Paulo Leal. Managing programming contests with Mooshak. *Software—Practice & Experience*, 2003.

25 José Paulo Leal and Fernando Silva. Mooshak: a Web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, May 2003.

26 José Paulo Leal and Fernando Silva. Using Mooshak as a Competitive Learning Tool. *The 2008 Competitive Learning Symposium*, 2008.

27 José Paulo Leal and Ricardo Queirós. CrimsonHex: A Service Oriented Repository of Specialised Learning Objects. In *Enterprise Information Systems*, volume 24 of *Lecture Notes in Business Information Processing*, pages 102–113. Springer Berlin Heidelberg, 2009.

28 José Paulo Leal and Ricardo Queirós. Defining Programming Problems as Learning Objects. In *International Conference on Computer Education and Instructional Technology (ICCEIT)*, 2009.

29 M.E. Lesk and E. Schmidt. Lex — A Lexical Analyzer Generator. *Unix Time-sharing system: Unix programmer's manual*, 2B, July 1975.

30 S. A. Mengel and J. Ulans. Using Verilog LOGISCOPE to analyze student programs. In *Proceedings of the 28th Annual Frontiers in Education - Volume 03*, FIE '98, pages 1213–1218, Washington, DC, USA, 1998. IEEE Computer Society.

31 Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

32 G. Michaelson. Automatic analysis of functional program style. In *Australian Software Engineering Conference, 1996., Proceedings of 1996*, pages 38 –46, jul 1996.

33 D.S. Morris. Automatically grading Java programming assignments via reflection, inheritance, and regular expressions. In *Frontiers in Education, 2002. FIE 2002. 32nd Annual*, volume 1, pages T3G–22, 2002.

34 Kevin A. Naudé, Jean H. Greyling, and Dieter Vogts. Marking student programs using graph similarity. *Comput. Educ.*, 54(2):545–561, February 2010.

35 Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, , 2000.

36 Ricardo Queirós and José Paulo Leal. Programming Exercises Evaluation Systems - An Interoperability Survey. In *Proceedings of the 4th International Conference on Computer Supported Education (CSEDU)*, pages 83–90, 2012.

37 Ricardo Queirós and José Paulo Leal. A Survey on eLearning Content Standardization. In *Information Systems, E-learning, and Knowledge Management Research*, volume 278 of *Communications in Computer and Information Science*, pages 433–438. Springer Berlin Heidelberg, 2013.

38 Ricardo Queirós and José Paulo Leal. Making Programming Exercises Interoperable with PExIL. In *Innovations in XML Applications and Metadata Management: Advancing Technologies*, chapter 3, pages 38–56. IGI Global, 2013.

39 Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*, Dec. 1999.

40 Kenneth A. Reek. The TRY system -or- how to avoid testing student programs. In *Proceedings of the twentieth SIGCSE technical symposium on Computer science education*, SIGCSE '89, pages 112–116, New York, NY, USA, 1989. ACM.

41 R. Romli, S. Sulaiman, and Kamal Zuhairi Zamli. Automatic Programming Assessment and Test Data Generation: a review on its approaches. In *2010 International Symposium in Information Technology (ITSim)*, volume 3, pages 1186–1192, 2010.

42 Riku Saikkonen, Lauri Malmi, and Ari Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education*, ITiCSE '01, pages 133–136, New York, USA, 2001. ACM.

43 Tom Schorsch. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education*, SIGCSE '95, pages 168–172, New York, USA, 1995. ACM.

44 Anuj Shah. Web-CAT: A Web-based Center for Automated Testing. Technical report, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 2003.

45 Nghi Truong, Peter Bancroft, and Paul Roe. A web based environment for learning to program. In *Proceedings of the 26th Australasian computer science conference*, volume 16 of *ACSC '03*, pages 255–264, Darlinghurst, Australia, 2003. Australian Computer Society.

46 Nghi Truong, Paul Roe, and Peter Bancroft. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education - Volume 30*, ACE '04, pages 317–325, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

47 Urs Von Matt. Kassandra: the automatic grading system. *SIGCUE Outlook*, 22(1):26–40, January 1994.

48 Tiantian Wang, Xiaohong Su, Peijun Ma, Yuying Wang, and Kuanquan Wang. Ability-training-oriented automated assessment in introductory programming course. *Comput. Educ.*, 56:220–226, January 2011.

49 Tiantian Wang, Xiaohong Su, Yuying Wang, and Peijun Ma. Semantic similarity-based grading of student programs. *Information and Software Technology*, 49(2):99 – 107, 2007.

50 N. Zamin, E. E. Mustapha, S. K. Sugathan, M. Mehat, and E. Anuar. Development of a Web-based Automated Grading System for Programming Assignments using Static Analysis Approach. In *International Conference on Technology and Operations Management (ICTOM'06)*, Institute Technology Bandung, Indonesia, December 2006.

51 K. Zen, D.N.F.A. Iskandar, and O. Linang. Using Latent Semantic Analysis for automated grading programming assignments. In *2011 International Conference on Semantic Technology and Information Retrieval (STAIR)*, pages 82 –88, june 2011.

52 Xiao Zhao, Liu Xuefeng, and Hou Yumo. Research and Implementation of Automatic Scoring System about Programming. In *International Conference on Computer Science Service System (CSSS)*, pages 225 –227, aug. 2012.

CodeSkelGen – A Program Skeleton Generator

Ricardo Queirós

CRACS & INESC-Porto LA & DI-ESEIG/IPP, Porto, Portugal
ricardo.queiros@eu.ipp.pt

Abstract

Existent computer programming training environments help users to learn programming by solving problems from scratch. Nevertheless, initiating the resolution of a program can be frustrating and demotivating if the student does not know where and how to start. Skeleton programming facilitates a top-down design approach, where a partially functional system with complete high-level structures is available, so the student needs only to progressively complete or update the code to meet the requirements of the problem.

This paper presents CodeSkelGen - a program skeleton generator. CodeSkelGen generates skeleton or buggy Java programs from a complete annotated program solution provided by the teacher. The annotations are formally described within an annotation type and processed by an annotation processor. This processor is responsible for a set of actions ranging from the creation of dummy methods to the exchange of operator types included in the source code.

The generator tool will be included in a learning environment that aims to assist teachers in the creation of programming exercises and to help students in their resolution.

1998 ACM Subject Classification D.3 Programming Languages; D.3.4 Processors; Code generation

Keywords and phrases Code Generation, Programming Languages, Annotation

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.145

1 Introduction

Several studies [2, 3] reported experiences where the students were tested on a common set of program-writing problems and the majority of students performed more poorly than expected. It was not clear why the students had difficulties to write the required programs. One possible explanation is that students, mainly novice students, lacked knowledge of fundamental programming constructs. Another explanation is that students despite being familiar with the constructs lacked the ability to “problem solve” [6].

One of the approaches mentioned in these studies was the delivery of skeleton code that the students should complete to meet the problem requirements. This approach was validated successfully and students found it a good choice. Other approach used was the definition of buggy programs. In this case the students would have to find logic errors in the program thus stimulating valences as debugging and testing. The rationale is simple: with the delivery of skeleton or buggy programs, the "problem-solving" issue is softened and the students' working memory is free to build a new mental model of the problem to solve.

This paper presents CodeSkelGen as a scaffolding tool to generate Java programs from an annotated solution program provided by the teacher. The generation process is based on annotations scattered throughout the code. These annotations are formally described through an annotation type that includes all the possible actions to make in the source code.

When Java source code is compiled, annotations are processed by a compiler plug-in called annotation processor. This type of processor will produce additional Java source files as versions of the solution program created by the teacher. These versions can be of two types: skeleton and buggy.

Skeleton programs will accelerate the beginning of exercises resolution by the students and facilitate their problem understanding. With the structure included, students can now focus on the core of the problem and abstract their foundations.

Buggy programs include logic and/or execution errors. These type of programs can stimulate students to debug and test their programs. Often this is a forgotten practice which leads to malfunctioning programs.

The motivation for the creation of this tool came from the need to integrate a code generation facility on an Ensemble instance. Ensemble is a conceptual tool[1] to organise networks of e-learning systems and services based on international content and communication standards. Ensemble is focused exclusively on the teaching-learning process. In this context, the focus is on coordination of educational services that are typical in the daily lives of teachers and students in schools, such as the creation, resolution, delivery and evaluation of assignments. The Ensemble instance for the computer programming domain relies on the practice of programming exercises to improve programming skills. This instance includes a set of components for the creation, storage, visualisation and evaluation of programming exercises orchestrated by a central component (teaching assistant) that mediates the communication among all components.

The remainder of this paper is organised as follows: Section 2 presents CodeSkelGen with emphasis on its two components: annotation type and annotation processor. Then, we present an explanation on how to use the annotations formally described on a source code. Then, to evaluate the generation tool, we present its integration on a network for the computer programming learning. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2 CodeSkelGen

CodeSkelGen is a code generator tool that generates Java partial programs. The teacher starts by creating the solution program for a specific problem and annotates the code based on the CodeSkelGen annotation type. Upon compilation the Java compiler uses the CodeSkelGen annotation processor to produce several sources files based on the annotations found in the solution program. The architecture of the generator tool is depicted in Figure 1.

The generated source files can be of two types: skeleton or buggy. In the former some methods are replaced by dummy methods. In the latter, operators and values are exchanged to produce buggy programs (with logic/execution errors).

2.1 Annotation Type

Annotations, in the Java computer programming language, are a form of syntactic metadata that can be added to Java source code. At runtime, annotations with runtime retention policy are accessible through reflection. At compile time, annotation processors (compiler plug-ins) will handle the different annotations found in code being compiled.

Java defines a set of annotations that are built into the language. The compiler reserves a set of special annotations (including @Deprecated, @Override and @SuppressWarnings) for syntactic purposes. However it is possible to create your own annotations by means of the

[1] `http://ensemble.dcc.fc.up.pt/`

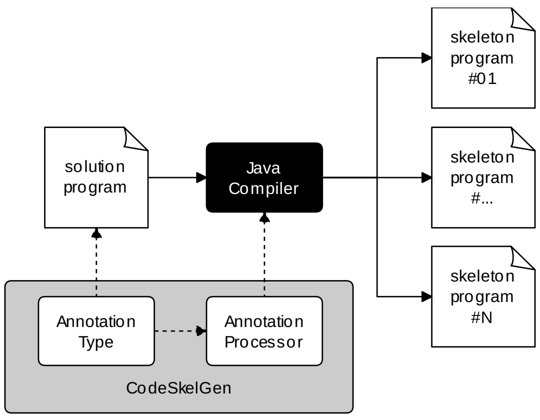


Figure 1 CodeSkelGen Architecture.

Listing 1 CodeSkelGen Annotation Type.

```
package CodeSkelGen;
@Retention(RetentionPolicy.SOURCE)
public @interface CSG {
    String changeOperator() default "";
    String changeValue() default "";
    String changeVariable() default "";
    String comment() default "";
    boolean removeBody() default false;
    ...
}
```

creation of annotation types. Annotation type declarations are similar to normal interface declarations. An at-sign (@) precedes the interface keyword. Each method declaration defines an element of the annotation type.

In annotation declarations, you can also specify additional parameters, for instance, what types of elements can be annotated (e.g. classes, methods) and how long the marked annotation type will be retained (CLASS – included in class files but not accessible at run-time; SOURCE - discarded by the compiler when the class file is created; and RUNTIME available at run-time through reflection).

In the CodeSkelGen, the interface CSG was created with a set of methods enumerated at Listing 1. The interface is composed by several methods. The most important are:

- *changeOperator()* - replaces the operator (arithmetic, relational or logic) by another;
- *changeValue()* - replaces a specific value (literal) by another;
- *changeVariable()* - replaces a specific variable by another existent one;
- *changeVariableType()* - change the variable types;
- *removeParameters()* - remove parameters from a method;
- *comment()* - defines generic messages;
- *removeBody()* - removes all the lines of code of an existing method and includes a return statement according to the method return type.
- *removeBodySection()* - removes all the lines of code applied to a while instruction or to a conditional instruction;
- *removeRefVariable()* - remove all the instructions that use a specific variable;

In order to process the CSG annotations you need to create an annotation processor.

2.2 Annotation Processor

Starting with Java 6, annotation processors were standardized through JSR 269 and incorporated into the standard libraries. Also the Annotation Processing Tool (apt) was integrated with the Java Compiler Tool (javac).

The annotation processor will be the responsible to process the annotations found in the source code. Listing 2 shows an excerpt of the foundations of the CodeSkelGen annotation processor.

A processor will "process" one or more annotation types. First, we need to specify what annotation types that our annotation processor will support by using *@SupportedAnnotationTypes* (in this case all) and the version of the source files that are supported by using *@SupportedSourceVersion* (in this case the version is JDK 6).

Then, we need to declare a public class for the processor that extends the *AbstractProcessor* class from the *javax.annotation.processing* package. *AbstractProcessor* is a standard superclass for concrete annotation processors that contains necessary methods for processing annotations. Inside the main class a *process()* method must be created. Through this method the annotations available for processing are provided. Note that through *AbstractProcessor*, you also access the *ProcessingEnvironment* interface. In the environment the processor will find everything it needs to get started, including references to the program structure on which it is operating, and the means to communicate with the environment by creating new files and passing on warning and error messages. More precisely, with this interface annotation processors can use several useful facilities, such as:

- *Filer* - a filer handler that enables annotation processors to create new files;
- *Messager* - a way for annotation processors to report errors.

The final step to finish the annotation processor is to package and register it so the Java compiler or other tools can find it. The easiest way to register the processor is to leverage the standard Java services mechanism:

1. Package your Annotation Processor in a Jar file;
2. Create in the Jar file a directory META-INF/services;
3. Include in the directory a file named *javax.annotation.processing.Processor*.
4. Write in the file the fully qualified names of the processors contained in the Jar file, one per line.

The Java compiler and other tools will search for this file in all provided classpath elements and make use of the registered processors.

2.3 Program annotation

After the creation of the annotation type and processor one must code the solution program that will use the annotation type previously created. The following excerpt at Listing 3 shows an annotated solution program coded by the teacher for the factorial problem.

Upon compilation the Java Compiler with the help of the registered annotation processors will generate several source files accordingly with the syntax of the annotations found in the source code and the associated semantic in the annotation processor. Listing 4 shows a possible source file.

Note that due to presentation purposes the program generated combines both program types supported by CodeSkelGen (skeleton and buggy).

Listing 2 CodeSkelGen Annotation Processor.

```
SupportedAnnotationTypes( "CodeSkelGen.CSG" )
@SupportedSourceVersion( SourceVersion.RELEASE_6 )
public class CSGAnnotationProcessor extends AbstractProcessor {
 public CSGAnnotationProcessor() {
  super();
 }
 @Override
 public boolean process(Set<? extends TypeElement> annotations,
    RoundEnvironment roundEnv) {
  //For each element annotated with the CSG annotation
  for (Element e : roundEnv.getElementsAnnotatedWith(CSG.class)) {
   //Check if the type of the annotated element is not a field.
   //If yes, return a warning.
   if (e.getKind() != ElementKind.FIELD) {
    processingEnv.getMessager().printMessage(Diagnostic.Kind.WARNING,
        "Not a field", e);
    continue;
   }
   //Define the following variables: name and clazz.
   String name = capitalize(e.getSimpleName().toString());
   TypeElement clazz = (TypeElement) e.getEnclosingElement();

   //Generate a source file with a specified class name.
    try {
        JavaFileObject f = processingEnv.getFiler().
            createSourceFile(clazz.getQualifiedName() + "Skeleton");
        processingEnv.getMessager().printMessage(Diagnostic.Kind.NOTE,
            "Creating " + f.toUri());
        Writer w = f.openWriter();
        //Add the content to the newly generated file.
        try {
            PrintWriter pw = new PrintWriter(w);
            pw.println("...");
            pw.flush();
        } finally {
            w.close();
        }
    } catch (IOException x) {
        processingEnv.getMessager().printMessage(
                        Diagnostic.Kind.ERROR, x.toString());
    }
   }
   return true;
  }
}
```

Listing 3 Program Annotation.

```
public class Program {
 @CSG(comment = "Calculate the factorial of the sum of 2 numbers")
 public static void main(String[] args) {
  long num1 = Long.parseLong(args[0]);
  long num2 = Long.parseLong(args[1]);
  long total = sum(num1,num2);
  System.out.println("Factorial of " + total + " is " + fact(total));
 }
 public static long fact(long num) {
  @CSG(changeValue=">")
  if (num <=1 )
    return 1;
  else
    @CSG(changeOperator)
  return num * fatorial(num - 1);
 }
 @CSG(comment="Complete the method!", removeBody=true)
 public static long sum(long num1, long num2) {
  return num1+num2;
 } }
```

Listing 4 Skeleton program generated.

```
public class Program {
 //Calculate the factorial of the sum of 2 numbers received by stdin
 public static void main(String[] args) {
  long num1 = Long.parseLong(args[0]);
  long num2 = Long.parseLong(args[1]);
  long total = sum(num1,num2);
  System.out.println("Factorial of " + total + " is " + fact(total));
 }

 public static long factorial(long num) {
  if (num >1 )
    return 1;
  else
        return num * fatorial(num + 1);
 }

 //Complete the method!
 public static long sum(long num1, long num2) {
    return 1;
 }
}
```

3 Integration into an Educational Setting

The motivation for the creation of this tool came from the need to integrate a code generation facility on an Ensemble instance. Ensemble is a conceptual tool to organise networks of e-learning systems and services based on international content and communication standards. Ensemble is focused exclusively on the teaching-learning process. In this context, the focus is on coordination of educational services that are typical in the daily lives of teachers and students in schools, such as the creation, resolution, delivery and evaluation of assignments. The Ensemble instance for the computer programming domain relies on the practice of programming exercises to improve programming skills. This instance includes a set of components for the creation, storage, visualisation and evaluation of programming exercises orchestrated by a central component (teaching assistant) that mediates the communication among all components.

Skeleton programs will be generated during the exercises authoring process (Figure 2) in Petcha [4]. Petcha is a teaching assistant component of an Ensemble instance for the computer programming domain. In the authoring process, the teacher fulfils a set of metadata regarding the exercise, codes the correct solution, annotates it, automatically generates skeleton programs and test cases and finally packages all these files in a IMS Common Cartridge (IMS CC) file. An IMS CC object is a package standard that assembles educational resources and publishes them as reusable packages in any system that implements this specification (e.g. Moodle LMS).

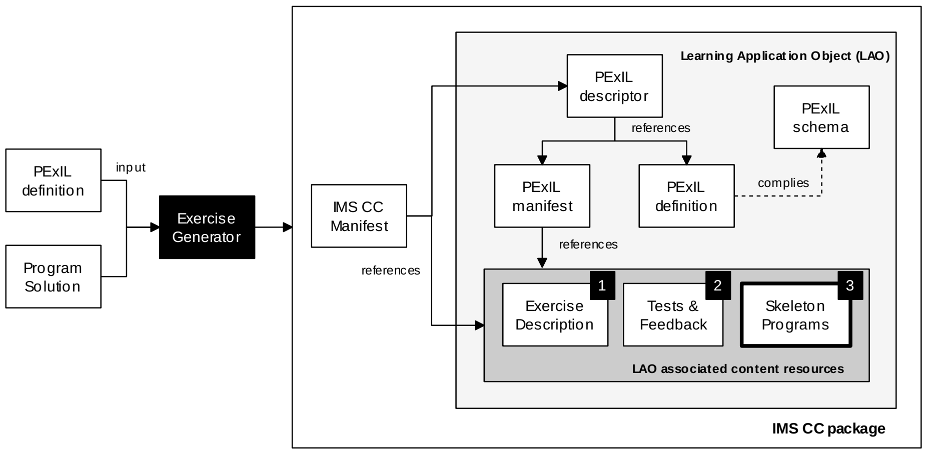


Figure 2 Programming Exercise Package.

Since the specification is insufficient to fully describe a programming exercise, an interoperability language was created called PExIL [5]. PExIL describes the life-cycle of a program exercise since its creation until its evaluation. The generation of a learning object (LO) package is straightforward as depicted in Figure 2. The Generator tool uses as input a valid PExIL instance and an annotated program solution file and generates 1) an exercise description in a given format and language, 2) a set of test cases and feedback files and 3) a set of skeleton programs. The PExIL data model depicted in Figure 3 accommodates all these files formalized through the creation of a XML Schema.

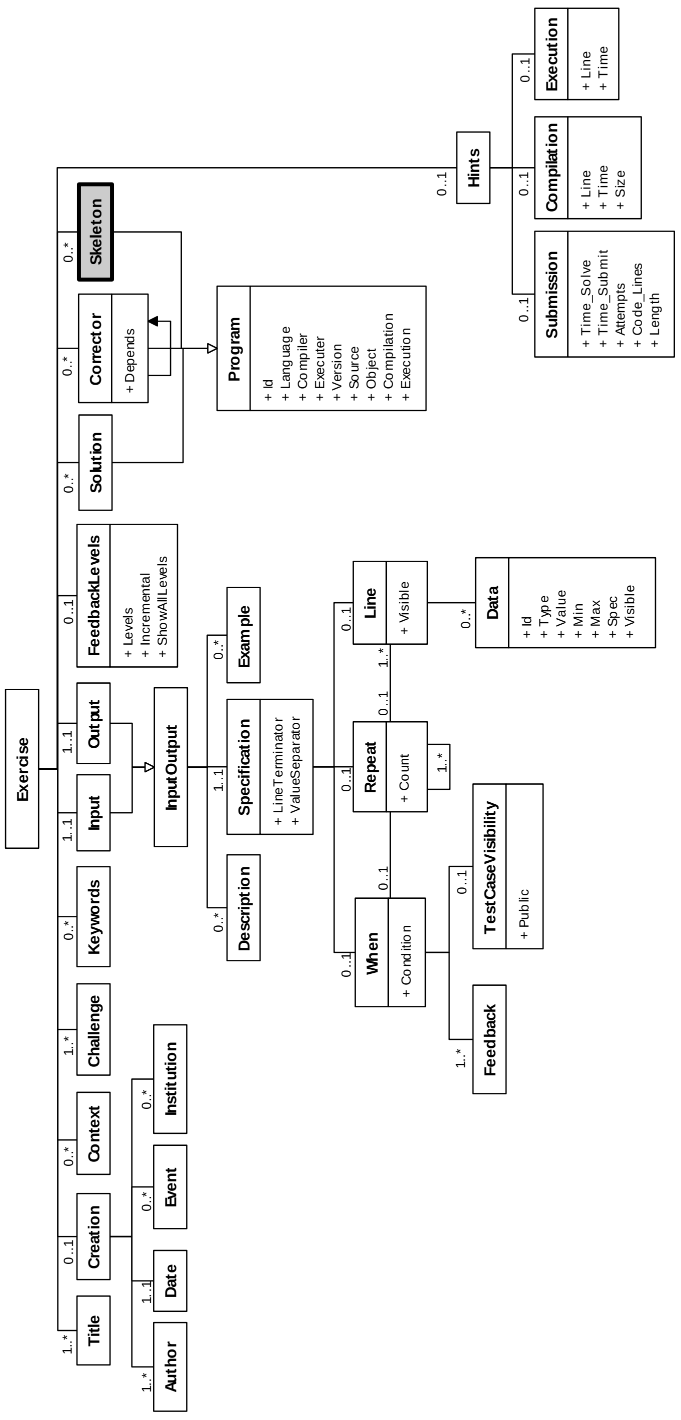


Figure 3 PExIL data model.

The PExIL schema is organized in three groups of elements:

1. Textual - elements with general information about the exercise to be presented to the learner. (e.g. title, date, challenge);
2. Specification - elements with a set of restrictions that can be used for generating specialized resources (e.g. test cases, feedback);
3. Programs - elements with references to programs as external resources (e.g. solution program, correctors, skeleton files) and metadata about those resources (e.g. compilation, execution line, hints).

Then, a validation step is performed to verify that the generated tests cases meet the specification presented on the PExIL instance and the manifest complies with the IMS CC schema. Finally, all these files are wrapped up in a ZIP file and deployed in a Learning Objects Repository (e.g. CrimsonHex [1]).

4 Conclusions and Future Work

This paper presents CodeSkelGen as a code generator tool. Despite not yet implemented most of the design and implementation details were enumerated. The tool is based on annotations. Firstly an annotation type was created to describe the type of operations that can be made on the annotated solution programs provided by the teacher. Secondly, an annotation processor was made to parse these annotations and process them. Finally, an example of how annotate a source file and a possible output was shared to understand the goal of the tool. The tool can produce two types of files: skeleton or buggy (or a combination of both). Based on some studies [2, 3] we think that these types of files will engage novice students on initiating the resolution of exercises and on stimulating them to test more effectively their solutions while using in a regular basis the debugger tools.

The main contribution of this paper is the approach used to generate partial programs. This can be helpful for other people that deal with similar problems. This approach has advantages and disadvantages:

- Advantages:
  - the processor is external to the source code;
  - the annotations processing is at compile time (not runtime);
  - the same annotated solution program can be the base for several different versions.
- Disadvantages:
  - language dependent (Java);
  - teacher must learn the elements of the annotation type.

As future work, it is expected to implement the CodeSkelGen and enrich the CSG interface with more pertinent constructs. Other research path will be find a language-independent approach to address the main issue of the approach presented in this paper.

References

1 José Paulo Leal and Ricardo Queirós. Crimsonhex: a service oriented repository of specialised learning objects. In *ICEIS 09 - 11th International Conference on Enterprise Information Systems, Milan, Italy*, volume 24 of *Lecture Notes in Business Information Processing*, pages 102–113. Springer-Verlag, LNBIP, Springer-Verlag, LNBIP, May 2009.

2 Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice

programmers. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '04, pages 119–150, New York, NY, USA, 2004. ACM.

3 Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, ITiCSE-WGR '01, pages 125–180, New York, NY, USA, 2001. ACM.

4 Ricardo Queirós and José Paulo Leal. Petcha - a programming exercises teaching assistant. In *ACM SIGCSE 17th Anual Conference on Innovation and Technology in Computer Science Education*, Haifa, Israel, July 2012 2012. ACM.

5 Ricardo Queirós and José Paulo Leal. Pexil: Programming exercises interoperability language. Conferência - XML: Aplicações e Tecnologias Associadas (XATA), 2011.

6 Jacqueline L. Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P. K. Ajith Kumar, and Christine Prasad. An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pages 243–252, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

Choosing Grammars to Support Language Processing Courses

Maria João Varanda Pereira[1], Nuno Oliveira[2], Daniela da Cruz[2], and Pedro Rangel Henriques[2]

1 **Polytechnic Institute of Bragança**
Bragança, Portugal
`mjp@ipb.pt`

2 **Universidade do Minho**
Braga, Portugal
`{danieladacruz,nunooliveira,prh}@di.uminho.pt`

Abstract

Teaching Language Processing courses is a hard task. The level of abstraction inherent to some of the basic concepts in the area and the technical skills required to implement efficient processors are responsible for the number of students that do not learn the subject and do not succeed to finish the course.

In this paper we intend to list the main concepts involved in Language Processing subject, and identify the skills required to learn them. In this context, it is feasible to identify the difficulties that lead students to fail. This enables us to suggest some pragmatic ways to overcome those troubles. We will focus on the grammars suitable to motivate students and help them to learn easily the basic concepts. After identifying the characteristics of such grammars, some examples are presented to make concrete and clear our proposal. The contribution of this paper is the systematic way we approach the process of teaching Language Processing courses towards a successful learning activity.

1998 ACM Subject Classification K.3.2 Computer and Information Sciences Education, D.3.1 Programming Languages - Formal Definitions and Theory

Keywords and phrases Teaching Language Processing, Domain Specific Languages, Grammars

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.155

1 Introduction

Learning was, is and will be difficult. The student has to interpret and understand the information he got, and then he has to assimilate the new information merging it with his previous knowledge to generate new knowledge.

However teaching is becoming more and more difficult as new student generations are no more prepared to absorb information during traditional classes.

Both statements are true in general, but they are particulary significant in domains that require a high capability for abstraction and for methodological analysis and synthesis. This is the case of Computer Science (CS), in general, and of Language Processing (LP) in particular.

As we will show in the next subsection, many other authors, researching and teaching in LP domain, have recognized the difficulties faced by both students and teachers. To overcome these difficulties, that frequently lead to the nonsuccess and nonsatisfaction of all the participants in the learning activity, and keeping in mind that higher education should focus on improving students' problem solving and communication skills, three main approaches can be identified:

- exploring different teaching methodologies;
- choosing motivating and adequate languages to illustrate concepts and to create project proposals;
- resorting to specific tools tailored to support the development of grammars and language processors in classroom context.

In this paper, we are interested in the second approach. Considering that *a person just learns when involved in a process*, we argue that motivation is a crucial factor to engage students in the course work allowing them to achieve the required knowledge acquisition. In this context, we intend to show that motivation is highly dependent on the languages used to work on during the course. We will discuss the characteristics that a language should have to be a motivating case study. We think that LP teachers should be very careful in their choices and be astute in the way they explore the underlying grammars along the course evolution.

1.1 Related Work

The next paragraphs describe contributions that intend to tackle the problem resorting to different teaching methodologies and techniques.

Li, in [9], states that most topics in a compiler course are quite theoretical and the algorithms covered are more complex than those in other courses. Usually the course content contribute to the lack of students motivation, giving rise to the students unsuccess and to the teacher frustration. The author also thinks that to improve teaching and learning, there are some effective approaches such as concept mapping, problem solving, problem-based learning, case studies, workshop tutorials and eLearning. In particular Problem-based Learning enables students to establish a relation between abstract knowledge and real problems in their learning. It can increase their interest in the course, their motivation to learn science, make them more active in learning, improve their problem solving skills and lifelong learning skills. The problem-based learning is a student-centered teaching approach; it was shown that the approach gets better results when enrolling students that are not at the first year.

Several authors advocate the use of Project-based Learning approaches to teach compilers. Although similar, Project-based and Problem-based Learning are distinct approaches. In Problem-based, the teacher prepares and proposes specific problems (usually focussed in a specific course topic, and smaller in size and complexity than a project) and the students work on each one, over a given period of time, to find solutions to the problems; after that, the teacher provides feedback to the students. In Project-based Learning the students, more than solve a specific problem, have to control completely the project; usually the project covers more than one topic and run over a larger period of time.

Islam et al, in [8], also agree with the complexity of the compiler course and consequently with the students difficulties in this subject. They propose an approach based on templates. Since the automatic construction of compilers is a systematic process, the main idea is to give students templates to produce compilers. The students just have to fill the parts necessary to implement the syntax and the semantics of the language.

Some other authors deal with the problem choosing carefully the language they use for the illustration of concepts or for exercises/projects, as we describe bellow.

Henry has published a paper [7] about the use of Domain Specific Languages for teaching compilers. He says that building a compiler for a domain specific language can engage students more than traditional compiler course projects. In this paper we defend a similar idea. In the cited paper, Henry proposes the use of a new programming language GPL

(Game Programming Language). GPL and the tools provided can be used to create exercises or projects that keep the students motivated because they can define, compile and test video games.

Years ago (1996), Aiken introduced in [2] the Cool Project that was based in an academic programming language used to teach compiler construction topics. Cool (Classroom Object-Oriented Language) is the name for both a small programming language and its processor. Two years later, a language called Jason (Just Another Simple Original Notion) was created by Siegfried [11]. It is a small language based in ALGOL that is used just for academic purposes. Although small, it contains all the important concepts of procedural programming languages that allow the students to extrapolate how to design larger-scale compilers.

Adams and Trefftz propose, in [1], the use of XML to teach compiler principles. They argue that XML processing or Programming Language processing are quite similar tasks, and that a compiler course can be a good place in a Computer Science curriculum to introduce at the same time the main concepts associated to both domains. According to that proposal, the students develop their own grammar and test their project using the tool XMLlint. The authors also describe their experience following that approach.

At last, the next paragraphs refer works that envisage to handle the problem resorting to adequate supporting tools. For that purpose, some compiler construction tools were developed to be used in classrooms.

One of the most significative examples is the work of Mernik et al [10] on LISA system. Using LISA it is possible to use a friendly interface to process Attribute Grammars and generate Compilers (lexical, syntactic and semantic components can be exercised solely or in a whole); useful visualizations are available for each compiler development/execution phase. These visualizations are the key point of LISA; they help students to understand easily the process or the internal structures involved in each phase.

Other examples can be seen in [5]. Demaille et al, introduce in this paper a complete compiler project based on Andrew Apple's Tiger language and on his famous book *Modern Compiler Implementation* [3, 4]. They augmented Tiger language and chose C++ as the implementation language. Considering a compiler as a long pipe composed of several modules, the project is divided in several steps, and students are requested to implement one or two modules. In particular the authors have invested efforts in tools to help students develop and improve their compiler and make the maintenance easier to teachers.

Barrtrada et al [6] combine theoretical and practical topics of the course using diverse modern technologies such as mobile learning, web-based learning as well as adaptive or intelligent learning. They develop a software tool that allows to create learning material for the compiler course to be executed in different learning environments.

The rest of the paper is organized as follows. Section 2 presents the topics that should be taught in an introductory Language Processing course building the correspondent Concept Map, and identifies the requirements that a student must satisfy for achieving the course goals. Section 3 discusses the main difficulties faced by students when attending a LP course. Section 4 introduces our proposal to overcome the difficulties, and defines the characteristics of a language to be considered adequate to support the course being two fold, motivating and enabling to progress incrementally the teaching activity. Section 5 illustrates our proposal, introducing a few examples. Any of those examples are suitable to explain both syntactic or semantic concepts; all of them can be used to support the course evolution, i.e. the introduction of new and more complex concepts, in an incremental mode. The purpose of this section is just to reinforce the approach and offer different alternatives. Section 6 closes the paper with a synthesis of our contribution.

2 Building a LP Course

In this section we define the subjects that should be taught in a *introductory*, *one semester*, **Language Processing course** (also called many times, a Compiler course) that is supposed to appear in the second or third year of an university degree on Computer Science or Software Engineering.

Before identifying the concepts that should be introduced and understood by the apprentices, it is mandatory to define the **learning objectives**.

Learning Objectives

At the end of the course unit the student is expected to be able to work with techniques and tools for formal specification of programming languages and automatic construction of language processors.

More than that, the student should understand the language processing tasks—the main approaches and strategies available for language analysis and translation—as well as the associated algorithms and data structures.

Course Contents

Now we can list the main topics that must be included in the contents of any LP course:

- Programming Language: concept, formal definition, syntax versus semantics, general purpose (GPL) versus domain specific (DSL) languages; examples; Language Design.
- Formal specification of Languages using Regular Expressions (RE) and Grammars (Gr): basic concepts like symbols or tokens of an alphabet, derivation rule or production, derivation tree, abstract syntax tree, contextual condition, attribute evaluation, etc...
- Language Processor: objectives, requirements and tasks; automatic generation tools, like Compiler Generators.
- Lexical Analysis using Regular Expressions and Reactive Automata (coping with symbol names and values).
- Syntactic Analysis using Context-Free Grammars (CFG) and Parsers:
  - Top-Down Parsing, TD (Recursive-Descendant, and LL(1));
  - Bottom-Up Parsing, BU (LR(0), LR(1), SLR(1), LALR(1)).
- Semantic analysis using Translation Grammars (TG) and Syntax Directed Translation (SDT): evaluating and sharing symbol-values, static semantic validation, and code generation using hash-tables and other global variables.
- Semantic analysis using Attribute Grammars (AG) and Semantic Directed Translation (SemDT): attribute evaluation, static semantic validation, and code generation using Abstract Syntax Trees and Tree Traversals.

Notice that part of these topics—those concerned with languages, grammars, and processing approaches or strategies—is more theoretical and will be introduced resorting to formal definitions and algorithms, while the other part—concerned with the implementation of language processors and their automatic generation— is more practical and can be supported by the development of exercises and projects, either manually from the scratch or recursing to tools.

Examples of problems that can be the subject of the above mentioned projects are: text filters; compiler for small or medium size programming languages; or translators for domain specific languages.

2.1 Topics to Learn in a LP Course: a Concept Map

To formalize the knowledge that a student is suppose to acquire in order to achieve the course objectives, we intend to build a Concept Map, or an ontology, describing the Language Processing domain.

The main concepts that we can infer from the course contents presented above are:

- PL – Programming Language;
- GPL – General Purpose Language; DSL – Domain Specific Languages;
- RE – Regular Expression;
- Gr – Grammar; Terminal and Non-Terminal Symbols, Start-symbol, Productions;
- CFG – Context Free Grammar; TG – Translation Grammar; AG – Attribute Grammar;
- LA – Lexical Analysis;
- SynA – Syntactic Analysis (or Parsing)
- TD – Top-Down Parsing; BU – Bottom-Up Parsing;
- SemA – Semantic Analysis;
- CG – Code Generation;
- SDT – Syntax Directed Translation; SemDT – Semantic Directed Translation;
- LP – Language Processor;
- LPG – Language Processor Generator; CG – Compiler Generator
- Interpreter; Analyzer; Compiler; Translator.

Figure 1 is a simplified version of the Concept Map that relates the concepts above in order to describe the knowledge domain under consideration.

2.2 Student skills required to learn LP

From the Concept Map introduced in the previous subsection, we can list the minimum programming skills that a student should have to understand the basic notions and learn the topics involved in a Language Processing course. They are

- knowledge about the basics of computer programming, at least in a imperative (procedural) programming language;
- knowledge about the basic iterative and recursive algorithms;
- knowledge about standard data structures (properties and operations) like list, sets, trees, graphs, tables (matrix) and hash-tables;
- basic knowledge about operating systems and computer architecture.

3 Difficulties faced by Students

When we deal with first year students attending introductory programming courses we know that we need several months to teach a programming language like C, C++ or Java. This happens because students have usually difficulties to interpret the problem statement, to analyze it, to translate what they want to do into an algorithm or a sequence of basic commands or operations. Besides the high level of abstraction required by those tasks, another difficulty arise from the fact that there are several ways to describe the same task in an algorithmic or programming language and the beginner needs to choose the more convenient one. Moreover, to code an algorithm, the student must pay careful attention to all lexical, syntactic and semantic details of the programming as, for instance, the use of semicolons at the end of each statement. There are an high amount of functions and methods spread along a big set libraries or classes that they have to use in an appropriate way. Moreover the students have usually lots of difficulties in algorithm understanding and

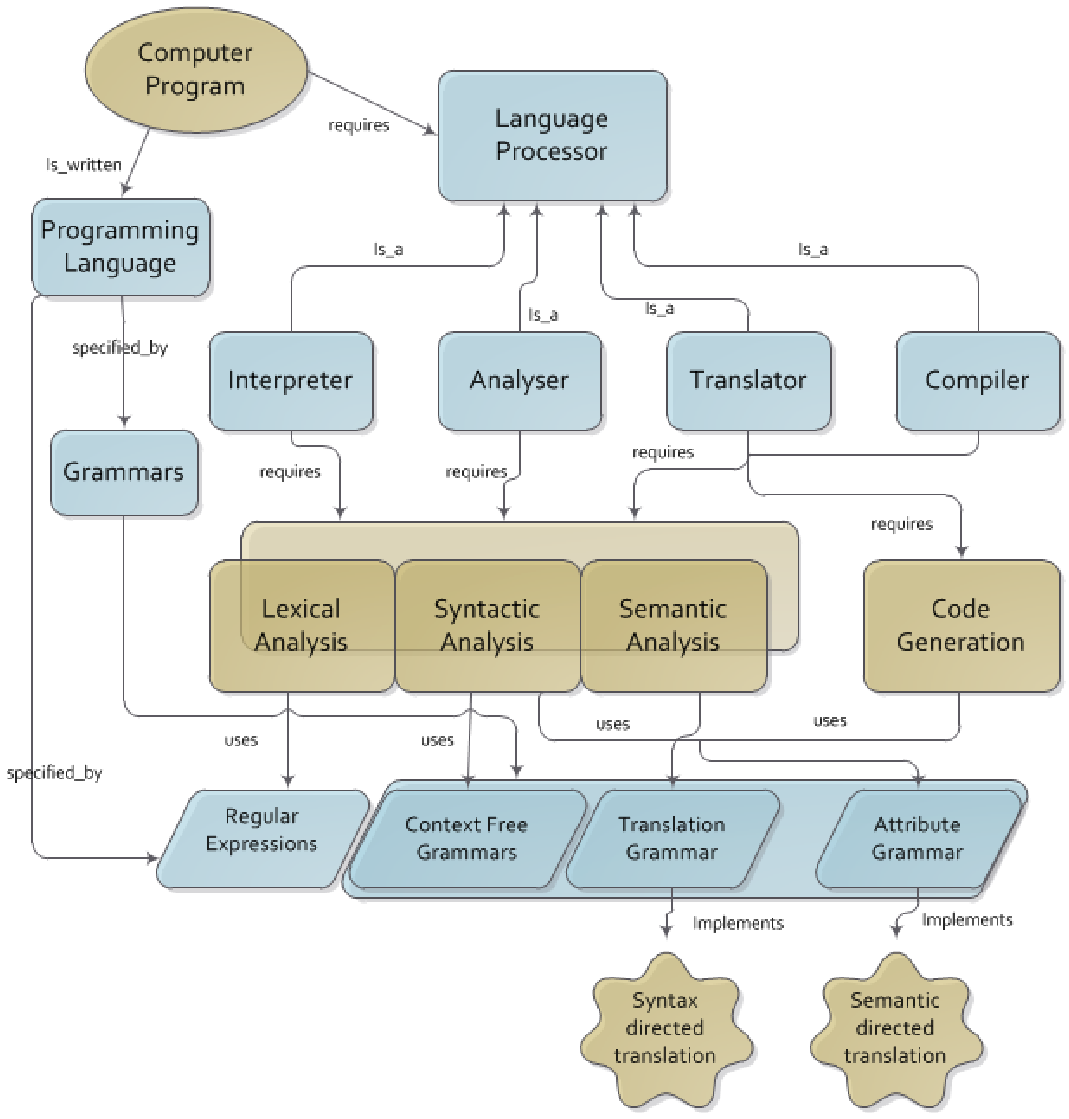


Figure 1 Concept Map describing the Language Processing Domain.

they can not see clearly the relation between the problem and the implementation of the program that is supposed to solve it. There are also data structures that are complex to define and to use.

These are the skills that are at last required for following successfully a Language Processing (LP) course.

In particular, and has remembered before, in LP courses the objective is to teach language principles and compiler construction techniques. For that, we must focus in presenting lexical, syntactic and semantic techniques. These techniques are complex and the students must understand the abstract concepts involved in the problem domain and be able to map them into the program domain concepts.

For instance, the students have difficulties in defining regular expressions since they have a strong expressive power using short specifications. Also the next steps are not easy. Parsing algorithm, attribute evaluation, bottom-up and top-down processes are subjects difficult to teach and difficult to understand.

There are lots of students that when are faced with such difficulties give up. The lack of motivation, due to the field of application traditionally not interesting for most of the students, is responsible for them not go deeply on studding and discontinue the course work.

4 Overcoming the Difficulties: Languages to Support Learning

We have identified the main topics and related concepts that must be taught in a Language Processing course (LPc), and the competences or abilities required to assure students success in such a course. We also identified the common struggles faced by LP learners. In this section we introduce our proposal to overcome the negative factors that lead apprentices to fail.

We assume that the permanent search for new pedagogical methods and techniques, that can be used alone or combined with traditional approaches, is a duty of every teacher in the context of any course. *Problem-based learning* or *Project-based learning* are two examples, discussed in section 1.1, of new methods introduced to improve the students' engagement. Also the resort to eLearning instruments, like forums or collaborative work platforms, is another example of that principle.

We also recognize the relevant role of didactic tools to support LPc. *Grammar Editors*, *Compiler Generators*, *Visualizers and Animators* that allow to follow the generation or compilation processes, are important examples of tools that shall be adopted to ease the students task and help them in understanding the basic concepts.

However our goal is to devise a strategy *to improve students' motivation* as the safest way to get them involved in the course activities helping them to learn with success LP concepts, methods, techniques and tools. With that in mind, we advocate the use of specially tailored languages that will be employed: (i) to illustrate concepts introduced in theoretical classes; (ii) to create exercises to solve in practical classes; and (iii) to elaborate project proposals for students homework.

Based on many years of teaching experience, we believe that this is the most effective approach to overcome the mentioned difficulties, ending up with a high ratio *students-approved/attendants*.

On one hand, we argue that those languages shall be *small* and *simple*. Small is measured in terms of the underlying grammar; a language is said small if the number of non-terminal and terminal symbols is small, as well as the number of grammar productions (or derivation rules). Simple is a twofold characteristic: the objects described by the language shall not be sophisticated and must be familiar for most of the students; and the tasks involved in the required processing shall be natural and not too complex for understanding or implementing. More than that, we believe that those languages shall possess an incremental character. This is, it shall be possible and straightforward to extend gradually the core language (the language initially proposed) in order to cover more objects in the language domain, or to add requirements concerning the processor output.

On the other hand, we argue that the chosen support languages shall be defined over special domains, instead of being programming languages. These domains must be instinctive for the apprentices, this is, well defined and closed to their common knowledge. In such context, the programs that students are supposed to develop, instead of being traditional *compilers*, will be *translators*—that, for a given input text, produce an output text in a different language—or *generic processors*—that extract data from the source text and compute information to be outputed.

Summing up, we propose the choice of appealing, small and simple, Domain Specific Languages (DSLs), by opposition to the recourse of General Purpose programming Languages (GPLs).

The approach here defended consists in choosing one friendly domain and a simple processing task and then write the grammar for the intended DSL and develop the respective processor. This step will cover the basic lexical, syntactic and semantic concepts. To teach more complex concepts or methods, or to discuss alternative strategies and techniques, the grammar shall evolve covering more domain components or performing more processing tasks. After this stage, other similar and equivalent DSLs shall be used to reinforce all the ideas so far presented.

Concerning project proposals, it is crucial that the language domain is attractive for the students and the project statement is opened enough to give room for their creativity, regarding both the language definition and the processing requirements.

5 Illustrating the Proposal: Examples

In this section we present some language examples to instantiate the approach proposed in the previous section. The examples introduce similar languages than can be used as alternatives to teach grammars (definition and variants, lexical and syntactic issues, static and dynamic semantic aspects of language processing).

Any of these languages are appropriate for an incremental approach enabling the teacher to start with a short and simple problem statement, asking the students to write the grammar (CFG and RE for terminals) and build manually some derivation trees. Then he can elaborate the statement covering more concepts in problem domain in order to extend the grammar. After dealing with the basic lexical and syntactic topics, the teacher can enrich the problem statement adding now some requirements for the desired output leading to the introduction of semantic actions writing the correspondent translation grammar (or, if it is the course objective, to the introduction of attributes, evaluation and translation rules and the correspondent attribute grammar). The requirements can be successively incremented with semantic constraints to introduce validation in semantic actions and error handling (or to introduce contextual conditions in attribute grammars).

All these steps can, and shall, be complemented with practical exercises supported by generating tools.

Each example presented in this section represents a different knowledge domain but in each domain it is possible to create a complete set of exercises tunning the language concerning the desired task.

5.1 1st Example: Lavanda

Lets, then, introduce a domain to work with (and within). Informally, lets think of a big launderette company that has several distributed facilities (collecting points) and a central building where the launder is made. The workflow on this company is as follows: each collecting point is responsible of receiving laundry bags from several clients, and send them to the central building, in a daily basis. The bags are dispatched to the central building with a *ordering note* that identifies the collecting point and describes the content of each bag.

Going deeply, each bag is identified by a unique identification number, and the name of the client owning it. The content of each bag is separated in one or more items. Each item is a quantified set of laundry of the same type, that is, with the same basic characteristics, for an easier distribution at washing time. The collecting points workers should always typify the laundry according to a class, a kind of tinge and a raw-material. The class is either *body cloth* or *household linen*; the tinge is either *white* or *colored* and finally, the raw-material is one of *cotton*, *wool* or *fiber*.

Once in the central building, the ordering notes are processed for several reasons: enter the notes' information into a database, calculate the number of bags received, produce statistics about the type of cloth received, define the value that each client must pay and so on.

Doing such processing by hand is risky because humans are easily error-prone. Therefore, an automatic and systematic way of processing the information in the notes is desirable. A reasonable way of achieving this is use the computer to do the job. In this context, the design of a computer language to describe the contents of an ordering note along with its suitable amount of rules (the grammar) that may be *taught* to a computer is the way to go.

Lavanda is the Domain Specific Language defined in the context of the domain described above, whose main application is to describe the ordering notes that the collecting points of the launderette company daily send to the central building.

Writing grammars according to the domain description requires that the domain concepts and the relations between such concepts are well understood. A good starting exercise is to outline an ontology where the relations between the several domain concepts are expressed. Notice that this approach is feasible due to the *domain size* and consequently, this happens because the domain is a specific one.

Once the domain is studied and internalized writing the grammar is much about giving a concrete shape to the relation between the domain concepts. This shape defines the syntax of the language Figure 2 presents the Context Free Grammar that formalizes the syntax of the language Lavanda.

A valid sentence written according to that grammar is presented below.

```
DAY 2013-03-20 CP Lidl
  BAG 1 CLI ClientA:
       (BODY-COLOUR-COTTON 1 , HOME-COLOUR-COTTON 2);
  BAG 2 CLI ClientB:
       (BODY-WHITE-FIBRE 10)
```

The grammar in figure 2 has enough complexity to propose exercises concerned with recursivity (lists), lists of lists, keywords and alternative productions.

```
p1:         Lavanda   →  Header  Bags
p2:         Header    →  DAY date CP IdCP
p3,p4:      Bags      →  Bag  |  Bags ';' Bag
p5:         Bag       →  BAG num  CLI IdCli ':' '(' Items ')'
p6,p7:      Items     →  Item  |  Items ',' Item
p8:         Item      →  Type  Quantity
p9:         Type      →  Class '-' Tinge '-' Material
p10:        IdCP      →  id
p11:        IdCli     →  id
p12:        Quantity  →  num
p13,14:     Class     →  BODY |  HOME
p15,16:     Tinge     →  WHITE    |  COLOUR
p17,18,19:  Material  →  COTTON   |  WOOL    |  FIBER
```

Figure 2 Lavanda Grammar.

Some examples of output requirements that can be formulated in this context are:

- compute the total of items delivered by each client;
- compute the total of bags in the order;
- compute the total of items in the class body clothes and total in the class household line;
- verify that there are not client identifiers occurring more than once.

It is also possible to add more productions to grammar in order to cope with some other concepts like prices, washing times and scheduling. This allows to add more complexity to the exercise and more tasks can be proposed like: compute the amount to be paid by each clients, consult the daily scheduler and the processing state of each bag, generate the invoices for each client, generate html code to construct a web page with the information involved, and so on.

5.2 2nd Example: Genea

This second example shows how in a completely different, but still common sense, domain we can define a language with characteristics similar to the previous one. We believe that students can be engaged in the exercises proposed around this subject.

Lets, then, introduce a second domain to work with (and within). A research organization devoted to demography and history has a complex application that constructs genealogical trees from simple specifications of families and offers a lot of statistics, computations and relation-based information. Family records consist of the basic part of each family which is the parents and their children. As it is obvious, dates play an important role in history, therefore born, death and wedding dates are important in this domain.

All persons are identified with their first name, but only the parents (a father or a mother) have their family names (as before marrying). Children hire their family name from the father. In contrast with the parents, children must have their gender defined.

Although small and very well defined, this domain is full of common-sense restrictions and relations that need to be respected. The most important ones concern chronological order, and age-related issues.

Regarding this simple domain, the researchers decided to build a language capable of specifying each family. The language should be processed by the application to construct the tree and to make information and computations available to the users of the application.

```
p1:      Genea     →  Families
p2,p3:   Families  →  Family  | Families ‘;’ Family
p4:      Family    →  Parents WED Wedding CHILDREN Children
p5:      Parents   →  Parent Parent
p6:      Parent    →  Type ‘:’ Name Name Life
p7,p8:   Children  →  &  |  Children Child
p9:      Child     →  Gender Name Life
p10:     Life      →  ‘(’ Born ‘-’ Death ‘)’
p11,12:  Type      →  FATHER | MOTHER
p13,14:  Gender    →  MALE | FEMALE
p15:     Born      →  date
p16,17:  Death     →  date | ‘?’
p18:     Wedding   →  date
p19:     Name      →  id
```

Figure 3 Genea Grammar.

Genea was the language defined based on this domain. Its concrete context free grammar is presented in Figure 3. Notice that the empty string symbol is denoted by &.

A sentence of the grammar is expressed below to show a concrete and correct source text.

```
FATHER : Herman Einstein (1847.08.30 - 1902.10.10)
MOTHER : Pauline Koch (1858.02.08 - 1920.02.20)
WED 1876.08.08
  CHILDREN
  MALE Albert (1879.03.14 - 1955.04.18)
  FEMALE Maja (1881.11.18 - 1951.06.25)
FATHER : Albert Einstein (1879.03.14 - 1955.04.18)
MOTHER : Mileva Maric (1875.12.19 - 1948.08.04)
WED  1903.01.06
  CHILDREN
  MALE Hans (1904.5.14 - 1973.07.26)
  MALE Eduard (1910.07.28 - 1965.10.25)
```

The grammar in figure 3 uses recursivity to represent lists of lists but the inner lists can be empty. Some interesting outputs can be produced from these lists. This language also uses some keywords, special characteres and non−literal terminals like date that would allow to propose some exercises related with their intrinsic value.

Some examples of output requirements that can be formulated in this context are:

- compute the number of children in each family, total and separated by gender;
- compute the total of families in the description;
- compute the average age at death;
- compute the mother’s age at the first birth (average);
- generate an SQL statement to insert each child in a database; the child’s family name is obtained concatenating the mother’s surname with the father’s surname;
- generate dot specifications in order to visualize the family tree;
- verify that the death date is greater than the birth date;
- verify that the wedding date lies within the birth and death interval.

5.3 3rd Example: Orienteering Paths Planner

Foot Orienteering is a widely developed sport in Portugal. Basically, an athlete receives a map with a marked path; in that path there are signaled control points that must be visited in the required order; at the end of the course, the athletes return to the start point and are scored according to control points visited and also according with the time spent. In each contest, competitors are divided by age class. A different path is given to each age class.

In order to help the organization of competition, we propose a new DSL to specify the list of paths (each path will be, therefore, a list of control points), so that the distance can be calculated and the course be visualized.

The required language should start by identifying all the control points of a given area where the competition takes place. Each point will be identified with an acronym and its Cartesian coordinates. Also, the language should enable us to define each path, indicating its name, age class, and list of points (described by acronyms). The order in the list establish the visiting order.

OPPL was the language defined based on this domain. Its concrete context free grammar is presented in Figure 4.

A sentence of the grammar is expressed below to show a concrete and correct source text.

```
POINTS
  A(3,5)
  B(4,2)
  C(5,5)
  D(9,9)
  E(5,15)
PATHS
  soft (>10) (A,B,C)
  medium (>20) (A,C,B,D)
  hard (>20) (A,E,C,D,B)
```

The grammar in figure 4 uses recursivity to represent two indepent lists. This allows to propose different exercises for each list. There are also some keywords and special characteres as in the other examples.

Some examples of output requirements that can be formulated in this context are:

- compute the total number of points and/or paths;
- compute the number of points in each path;
- compute the distance between two points;
- compute the length of a path;
- generate dot code to visualize the paths.

```
p1:      OPPL    →  POINTS Points PATHS Paths
p2,p3:   Points   →  Point  | Points Point
p4:      Point    →  letter '(' num ',' num ')'
p5,p6:   Paths    →  Path | Paths Path
p7:      Path     →  name Age '(' List ')'
p8:      Age      →  '(' '>' num ')'
p9,p10:  List     →  List ',' letter | letter
```

Figure 4 OPPL Grammar.

It is also possible to add more productions to the grammar in order to cope with the athlete information. In this context new symbols must be created representing names, numbers, paths, time spent and scores of each athlete. More exercises can be proposed with this new information; new output results can be required, like athletes ranking, partial scores, historical results and so on.

6 Conclusion

The use of DSLs in teaching methodologies allows to chose a knowledge domain appropriated to the students. When students are aware of the domain, its main concepts and relations, it is much easier to explain and discuss the processing of a language in that domain. In this sense, the efforts made to explain a subject like language processing do not dependent any more on the complexity of GPL grammars.

The usual grammar size of DSLs is more appropriate for teaching when compared with GPLs. Smaller grammars allows the students to understand better the concepts involved. Moreover, these kind of languages can be easily changed, adapted or incremented depending on the complexity of the example that the teacher desires to show and discuss with students.

Working within these small and common sense domains we can hope that the students quickly and easily guess the processing results expected for given source text samples. This allows to check if the language processing is well done or if there something that must be tuned.

Our proposal differs from the others in the sense that we do not create a special language to support our teaching activities. Instead we present the characteristics that a language, and its grammar, should exhibit to be helpful. Besides that, we systematize how to take profit of the toy languages chosen to introduce different topics and evolve from a concept to the next concept, in a smooth and challenging way in order to keep students interested and engaged.

We did not perform an evaluation experiment but we have been using this approach during the last 10 or 15 years and the results are truly positive. We achieved an average of 80% of students approved over students assessed. The grades reached by the students in the practical works prove that they are motivated, they adquired the basic language processing concepts, they were able to apply them and they had an opportunity to use their criativity.

References

1 D. Robert Adams and Christian Trefftz. Using XML in a compiler course. *ACM Sigcse Bulletin*, 36:4–6, 2004.

2 Alexander Aiken. Cool: A portable project for teaching compiler construction. *Sigplan*, 1996.

3 Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, 2004.

4 Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, 2002.

5 Akim Demaille, Roland Levillain, and Beroit Perrot. A set of tools to teach compiler construction. *ACM SIGCSE*, 40(3):68–72, 2008.

6 M.L. Barron Estrada, Ramon Zatarain Cabada, Rosalio Zatarain Cabada, and Carlos A. Reyes Garcia. A hybrid learning compiler course. *Lecture Notes in Computer Science*, 6248:229–238, 2010.

7 Tyson R. Henry. Teaching compiler construction using a domain specific language. *ACM SIGCSE*, 37(1):7–11, 2005.

8 Md. Zahurul Islam and Mumit Khan. Teaching compiler development to undergraduates using a template based approach. *Bangladesh*.

9 ZhaoHui Li. Exploring effective approaches in teaching principles of compiler. *The China Papers*, 2006.

10 Marjan Mernik and V. Zumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68, 2003.

11 Robert M. Siegfried. The jason programming language, an aid in teaching compiler construction. *ESCCC*, 1998.

Part V

Domain Specific Languages

Role of Patterns in Automated Task-Driven Grammar Refactoring*

Ján Kollár[1] and Ivan Halupka[2]

1 **Department of Computers and Informatics**
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
`Jan.Kollar@tuke.sk`

2 **Department of Computers and Informatics**
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
`Ivan.Halupka@tuke.sk`

Abstract

Grammarware engineering, and grammar-dependent software development has received considerable attention in recent years. Despite of this fact, grammar refactoring as a significant cornerstone of grammarware engineering is still weakly understood and little practiced. In this paper, we address this issue by proposing universal algorithm for automated refactoring of context-free grammars called mARTINICA, and formal specification language for preserving knowledge of grammar engineers called pLERO. Significant advantage of mARTINICA with respect to other automated refactoring approaches is that it performs grammar refactoring on the bases of user-defined refactoring task, rather then operating under some fixed objective of refactoring process. In order to be able to understand unified refactoring process of mARTINICA this paper also provides brief insight in grammar refactoring operators, which in our approach provide universal refactoring transformations for specific context-free grammars. For preserving of knowledge considering refactoring process we propose formalism based on patterns which are well-proven method of knowledge preservation in variety of other domains, such as software architectures.

1998 ACM Subject Classification F.4.3 Formal Languages (D.3.1)

Keywords and phrases Automated grammar refactoring, refactoring operators, grammar refactoring patterns, evolutionary algorithms, task-driven transformations

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.171

1 Introduction

Our work in the field of automated grammar refactoring derives from the fact that two or more equivalent context-free grammars may have different forms. Although two equivalent grammars generate the same language, they do not necessarily share some other specific properties that are measurable by grammar metrics [3]. The form in which a context-free grammar is written may have a strong impact on many aspects of its future application. For example, it may affect the general performance of the parser used to recognize the language generated by the grammar [4], or it may influence, and in many cases limit, our choice of parser generator for use in implementing the syntactic analyzer [4].

* This work was supported by project VEGA 1/0341/13 *Principles and methods of automated abstraction of computer languages and software development based on the semantic enrichment caused by communication.*

2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 171–186
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The ability to transform one grammar to another equivalent grammar becomes the capability to shift between domains of the possible application of grammars. Although this ability makes each context-free grammar more universal in the scope of its application, its practical advantages may easily be overwhelmed by the difficulties that this approach can introduce. The problem is that grammar refactoring is in many cases a non-trivial task, and if done manually it is prone to errors, especially in the case of larger grammars. This is an issue, because there is in general no formal way of proving that two context-free grammars generate the same language, since this problem is undecidable.

In our previous work [13], we addressed this issue by proposing an evolutionary algorithm for automated task-driven grammar refactoring. The algorithm is called mARTINICA (metrics Automated Refactoring Task-driven INcremental syntactIC Algorithm). The main idea behind this algorithm is to apply a sequence of simple transformation operators on a chosen context-free grammar in order to produce an equivalent grammar with the desired properties. Each refactoring operator transforms arbitrary context-free grammar into equivalent context-free grammar which may have different form than original grammar. Purpose of mARTINICA is to find sequence of refactoring operator instances that transforms specific context-free grammar into equivalent grammar whose form satisfies user-defined requirements. The current state of development of the algorithm requires that the grammar's production rules be expressed in BNF notation.

Refactoring operators with respect to diversity of possible requirements on qualitative properties of context-free grammars provide relatively universal grammar transformations. Although relative universality of refactoring operators contributes to versatility of refactoring algorithm, it also may lead to high computational complexity and in some specific cases inability of mARTINICA to fulfill refactoring task. In our current research, we propose solution to these issues, based on patterns, which in this context we consider to be a problem-specific refactoring operators.

Pattern in general is a problem-solution pair in given context [14, 15]. Christopher Alexander argues that each pattern can be understood as an element of reality, and as an element of language [14]. Pattern as an element of reality is a relation between specific context, certain system of forces recurring in given context and certain spatial configuration that leads to balance in a given system of forces [14]. Pattern as an element of language is an instruction, which shows how certain spatial configuration can be repeatedly used in order to balance certain system of forces wherever specific context makes it relevant [14].

As such, patterns are means for documenting of existing, well proven design knowledge, and they support creation of systems with predictable properties and quality attributes [15]. In our view, role of patterns in the field of grammar refactoring is to preserve knowledge of language engineers about when and how to refactor context-free grammars and to support process of grammar refactoring by providing this knowledge. In order to incorporate patterns in the process of automated grammar refactoring we have coined new term – grammar refactoring patterns. Each grammar refactoring pattern describes a way in which a context-free grammar can be transformed, with preserving of language that it generates, specific situation in which this transformation is possible and consequences of this transformation on specific quality attributes of a context-free grammar. Description of situation in which transformation provided by specific pattern can be applied on specific grammar defines refactoring problem that pattern addresses. Grammar transformation provided by pattern defines solution of refactoring problem. Description of consequences of applying transformation provides context in which pattern should be used.

This paper is organized as follows. Section 2 provides motivation considering our research and briefly discusses possible domains of our approach's application. Section 3 discusses related work, while our refactoring algorithm is described in section 4. Section 5 presents some experimental results considering our refactoring approach, while grammar refactoring patterns are discussed in section 6.

2 Motivation

Grammarware engineering is an up-and-rising discipline in software engineering, which aims to solve many issues in grammar development, and promises an overall rise in the quality of grammars that are produced, and in the productivity of their development [1]. Grammar refactoring is a process that may occur in many fields of grammarware engineering, e.g. grammar recovery, evolution and customization [1]. In fact, it is one of five core processes occurring in grammar evolution, alongside grammar extension, restriction, error correction and recovery [5]. The problem is that, unlike program refactoring, which is well-established practice, grammar refactoring is little understood and little practised [1].

If there is a clear purpose for which the grammar is being developed, its specification for an experienced grammar engineer is usually not an issue. Problems arise when a grammar is being developed for multiple purposes [5], or when a grammar engineer lacks knowledge about the future purpose of the grammar. In the first case, the problem is usually solved by developing multiple grammars of one language [5]. This need to develop multiple grammars could be replaced by developing a single grammar generating a given language and automatically refactoring it to another form suited to satisfy certain requirements, thus increasing the productivity of the grammar engineer. In fact, this is one of the main objectives of our work in the field of grammar refactoring. Ability to algorithmically change form in which context-free grammar is expressed makes it in this context more abstract and widens the scope of grammars possible application.

In cases when the grammar engineer lacks knowledge about some aspect of the future purpose of the grammar, its final shape may not satisfy some of the specific requirements, even if it generates correct language. Example of such situation would be development of left-recursive grammar which should be parsed by LL(k) parser, in which case form of the grammar would not satisfy requirements considering parser implementation. In this case, the grammar must either be refactored or be rewritten from scratch, thus draining valuable resources. An automated or even semi-automated way of refactoring the grammar could produce significant savings in this redundant consumption of resources. These are not the only two scenarios where an efficient refactoring tool is needed. In fact, an automated approach can be useful in all cases where we have a grammar with a form that needs to be changed while preserving the language that it generates. In this case, we see two domains for applying our algorithm, i.e. adaptation of legacy grammars, and grammar inference.

Parser generators and other implementation platforms for context-free grammars develop over time. Newly-established platforms and other tools operating with context-free grammars may require a form in which the grammar should be expressed that differs from the tools for the previous technological generation, or that operate with unequal efficiency over the same grammar forms. Kent Beck states that programs have two kinds of value: what they can do for today, and what they can do for tomorrow [6]. When we take this principle into the account, we can say that the ability to refactor a context-free grammar in order to adjust it to the requirements of current platforms is in fact the ability to add value to the legacy formalization of the language.

Grammar inference is defined as recovering the grammar from a set of positive and negative language samples [7]. Grammar inference focuses on resolving issues of over-generality and over-specialization of the generated language [8], while the form of the grammar is only a secondary concern. Grammar recovery tools in general do not allow their users enough fine-grained tuning options for recovering a grammar in the desired form, making it in many cases difficult to comprehend, and not useful until it has been refactored [9].

Sequence of refactoring operator instances provides transformation from some context-free grammar into another equivalent context-free grammar, and thus this sequence provides unidirectional formal relation between two equivalent grammars. In this context, a set of refactoring operators forms a universal vocabulary of grammar refactoring. On the other hand grammar refactoring patterns can be viewed as problem-specific refactoring operators and as such they form more abstract, domain-specific vocabulary of grammar refactoring. Sequence of grammar refactoring pattern instances does not only preserve relation between two equivalent grammars, but also captures rationale behind each refactoring decision and thus enables us to more deeply analyze and understand specific refactoring process.

3 Related Work

We were able to find very little reported research in the field of automated grammar refactoring. The small amount of work that we did find is mostly concerned with refactoring context-free grammars in order to achieve some fixed domain-specific objective.

Kraft, Duffy and Malloy developed a semi-automated grammar refactoring approach to replace iterative production rules with left-recursive rules [9]. They present a three-step procedure consisting of grammar metrics computation, metrics analysis in order to identify candidate nonterminals, and transformation of the candidate non-terminals. The first and third step of this procedure are fully automated, while the process of identifying non-terminals to be transformed by replacing iteration with left recursion is done manually. This approach is called metrics-guided refactoring, since the grammar metrics are calculated automatically, but the resulting values must be interpreted by a human being, who uses them as a basis for making decisions necessary for resuming the refactoring procedure. The work also provides an exemplary illustration of the benefits of grammar refactoring, since left-recursive grammars are more useful for some aspects of the application of a grammar [10] and are also more useful to human users [11] than iterative grammars.

The procedure for left-recursion removal is a well-known practice in the field of compiler design. An algorithm for automated removal of direct and indirect left recursion can be found in Louden [12]. This approach is further extended by Lohmann, Riedewald and Stoy [11], who present a technique for removing left-recursion in attribute grammars and semantic preservation while executing this procedure.

4 Background

In this section we discuss refactoring operators, as a basis for understanding of grammar refactoring patterns and core idea of our approach. We also discuss our refactoring algorithm as a background related to implementation of mARTINICA and interpretation of experimental results. This section also briefly introduces reader to method of describing properties of context-free grammar via formalism of objective function, which in context of our approach is used as a specification of refactoring objective.

4.1 Refactoring Operators

Formally, a grammar refactoring operator is a function that takes some context-free grammar $G = (N, T, R, S)$ and uses it as a basis for creating a new grammar $G' = (N', T', R', S')$ equivalent to grammar G. This function may also require some additional arguments, known as operator parameters. We refer to each assignment of actual values to the required operator parameters of the specific grammar refactoring operator as refactoring operator instantiation, and an instance of this refactoring operator is referred to as a specific grammar refactoring operator with assigned actual values of its required operator parameters.

At this stage of development, we have experimented with a base of eight grammar refactoring operators (Unfold, Fold, Remove, Pack, Extend, Reduce, Split and Nop), the first three of which have been adopted from Ralf Läammel's paper on grammar adaptation [2], while the others are proposed by us.

Nop is operator of identical transformation, and as such it does not impose any changes on context-free grammar. Unfold replaces each occurrence of specific non-terminal within some subset of production rules with right side of production rules whose left side is this non-terminal, and in BNF notation this transformation can lead to increase in number of production rules. Fold replaces some symbol sequences on the right side of some subset of grammar's production rules with specific non-terminal, whose right side is this sequence of symbols, and as such this operator provides inverse function to unfold operator. Remove operator removes specific non-terminal and all production rules containing this non-terminal on their right or left sides from grammar, but only in case when this transformation does not impose changes on language that grammar generates. Pack replaces specific sequence of symbols within right side of certain production rule with new-non-terminal, and creates production rule whose left side is this non-terminal and whose right side is equivalent to this sequence. Extend introduces new non-terminal, creates production rule whose left side is this non-terminal and right side is some other non-terminal, and replaces all occurrences of this other non-terminal within some subset of grammar's production rules with this new non-terminal. Reduce operator removes multiple production rules with equivalent right sides, but only in case when this transformation preserves language that grammar generates. In case when there are multiple production rules whose left side is specific non-terminal, split creates new grammar in which each of such rules will have different non-terminal on its left side. More detailed description of individual refactoring operators can be found in [2, 13].

In our approach, we use grammar refactoring operators as a tool for incremental grammar refactoring. We tend to keep the number of operators as small as possible, and we try to keep the refactoring operators as universal as possible. This is mainly because, as the base of refactoring operators grows, the state space of possible solution grammars also grows, and thus the size of the base of operators has a significant impact on the calculation complexity of the algorithm. However, lack of domain-specific refactoring processes is compensated by the overall openness of the base of operators, which means that it is a relatively trivial task to expand it or reduce it. In fact, the only refactoring operator required by the algorithm, which must reside at all times in the base of the operators, is the Nop operator.

In this work, we propose grammar refactoring patterns, as addition to base of refactoring operators. However, key difference between operators and patterns in this context is that growth in the number of refactoring patterns in base of refactoring operators does not have significant negative impact on calculation complexity of the algorithm, and in many cases opposite is the true. This is caused by their domain-specific orientation and relatively narrow scope of refactoring tasks for which individual patterns are applicable.

4.2 Objective Function

We adopt a somewhat modified understanding and notation of objective functions from mathematical optimization. In this case, the objective function describes the properties of the context-free grammar that we seek to achieve by refactoring. However, it does not describe the way in which refactoring should be performed, and the condition in which desired properties of the grammar are achieved.

In our view, the objective function consists of two parts: *objective* and *state function*. Our automated refactoring algorithm works with only two kinds of objectives, which are minimization and maximization of a state function. We define a state function as an arithmetic expression whose only variables are the grammar metrics calculable for any context-free grammar. As such, a state function is a tool for qualitative comparison of two or more equivalent context-free grammars.

Until now, we have experimented with some grammar size metrics [3], e.g. number of non-terminals (var) and number of production rules ($prod$). An example of an objective function defining the refactoring task to be performed on grammar G is (1).

$$f(G) = minimize\ 2 * var + prod \tag{1}$$

4.3 Refactoring Algorithm

The main idea behind our grammar refactoring algorithm is to apply a sequence of grammar refactoring operators to a chosen context-free grammar, in order to produce an equivalent grammar with a lower value of the objective function, when the objective is minimization, or a higher value of the objective function when the objective is maximization. Since it is an evolutionary algorithm, it also requires some other input parameters, in addition to the *initial grammar* and the *objective function*, in order to be executed. The algorithm requires three other input parameters: *number of evolution cycles*, *population size* and *length of life of a generation*. The first two of these parameters are characteristic for algorithms of similar type, while the third parameter is our own.

As shown in Fig. 1, which presents a white-box view of our algorithm, the central figure in mARTINICA is an abstraction called *population of grammars*. In our view, population of grammars is a set containing a constant number of grammar population entities. Its main property is that, after performing an arbitrary step in our algorithm, the number of elements in the population of grammars is always equal to the population size.

Further, we define a grammar population entity as an arranged triple of elements: *post-grammar*, *process chain of grammar generation*, and *difference in objective functions*. A post-grammar is a context-free grammar equivalent to the initial grammar. The process chain for grammar generation is a sequence of refactoring operator instances that was used to create the post-grammar from the corresponding post-grammar of the previous generation. The number of refactoring operator instances in each grammar generation process chain is always equal to the length of life of a generation. The difference in objective functions is the difference between the values of the objective function calculated for a post-grammar of the current population and the corresponding post-grammar of the previous generation.

4.3.1 Refactoring Operators Instantiation

All operator instances occurring in our algorithm are created automatically in one of three procedures, which are referred to as *random operator creation*, *random parameter creation*, and *identical operator creation*.

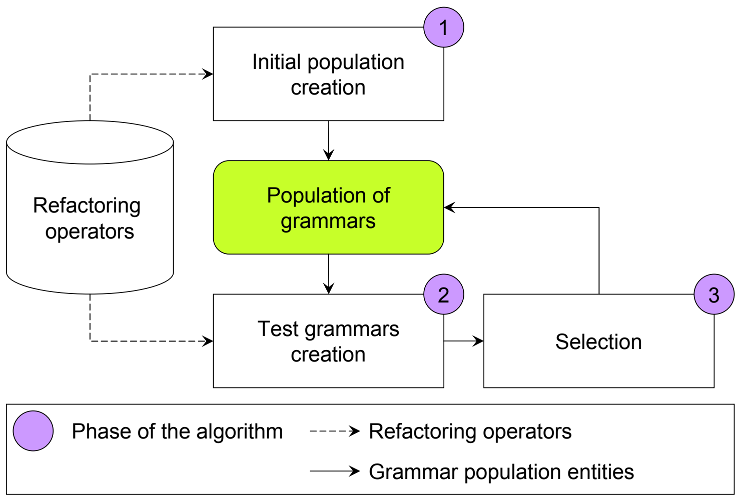


Figure 1 White-box view of mARTINICA.

Random operator creation creates instance of a random refactoring operator with random parameters. The first step in this procedure randomly selects an operator from the base of grammar refactoring operators. In this procedure, each grammar refactoring operator has the same probability of being selected. The second step in the procedure defines specific operator parameters for this operator on the basis of the grammar on which the operator instance will be applied. All possible combinations of operator parameters that respect the restrictions defined by a specific refactoring operator have the same probability of being generated in this procedure.

Random parameter creation creates an operator instance originating from some other operator instance. The two mentioned operator instances share the same refactoring operator, but their operator parameters may differ, since new operator parameters have been created in a procedure analogous to the second step of the random operation creation procedure. The only exception to this rule occurs when there is no acceptable combination of operator parameters for a given refactoring operator to be applicable to the given context-free grammar. In this, the random parameter creation procedure returns an instance of the Nop operator.

Identical operator creation creates an instance of the Nop grammar refactoring operator.

4.3.2 Creating an Initial Population

In the first phase of mARTINICA, the initial population of the grammars is created, and as such this phase is not repeated throughout the algorithm.

The first step in this phase is to create grammar generation process chains for each grammar population entity. All operator instances of each process chain created in this phase of the algorithm are created in the random operator creation procedure, except for one, whose operators are all created in the identical operator creation procedure. The reason for this exception is to guarantee that the initial grammar will be incorporated into the initial population of grammars. Since the sequence of operator instances contained in the process chain must be applicable to the grammar for which are they being generated, in exact order, we must consider all changes to the grammar performed by one refactoring operator instance in order to be able to generate the next operator instance of the process chain. We solve this issue by generating intermediate grammars after each random operator creation procedure by

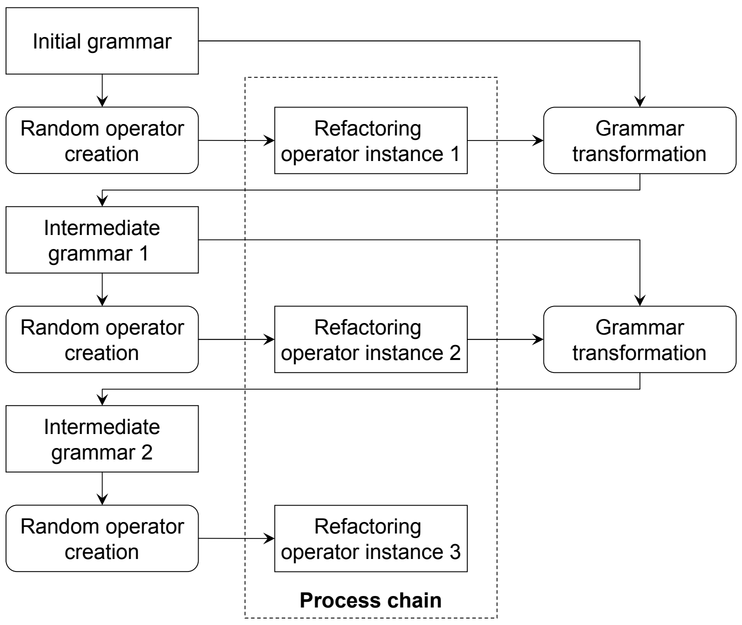


Figure 2 Creating a random process chain.

applying this operator instance on the grammar for which the random refactoring operator instance is being generated. We then generate the next random operator instance of the process chain on the basis of the intermediate grammar. In order to better understand the idea behind this approach, we provide an example of creating a process chain consisting of three random refactoring operators for the initial grammar. This example is shown in Fig. 2.

The second step in the first phase of the algorithm creates corresponding post-grammars for each grammar population entity by applying its process chain to the initial grammar, and finally the third step calculates the difference of the objective function calculated for the initial grammar and the post-grammar of the corresponding grammar population entity.

4.3.3 Creating Test Grammars

The second and third phase of the algorithm, called test-grammar creation and selection, are repeated in sequence for a number of evolution cycles. In test-grammar creation, we create three test grammar population entities for each grammar population entity. These entities are called *self-test grammar*, *foreign-test grammar*, and *random-test grammar*.

Self-test grammar is created on the basis of the corresponding grammar population entity and process chain, generated on the basis of the process chain of this entity. All refactoring operator instances in the newly generated process chain are created in the random parameter creation procedure, and the algorithm for creating them is analogous to the algorithm for creating a random process chain in the initial population of the grammar creation phase. Self-test grammar is therefore a grammar population entity containing a grammar that was created on the basis of the same refactoring operators as those on which original tested grammar was created, but these operators may have different process parameters.

Foreign-test grammar is created in a similar procedure as for self-test grammar, with the exception that the new population entity is not created on the basis of a tested grammar process chain, but on the basis of some other grammar population entity process chain. This population entity is randomly selected from the population of grammars.

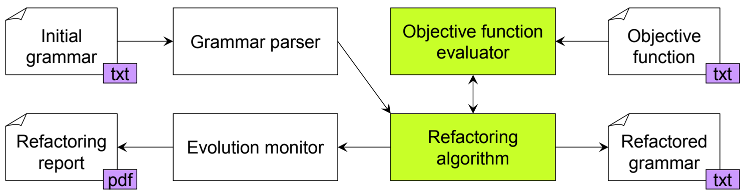


Figure 3 Architecture of Grammar Refactoring System.

Random-test grammar is created in a procedure analogous to the procedure for creating a random grammar population entity in the first phase of the algorithm, with the exception that the random process chain is not being generated for an initial grammar, but for a grammar contained within the tested grammar population entity.

4.3.4 Selection and Evaluation

In the selection phase of the algorithm, we compare the value of the objective function of each grammar within the population of grammars with the values of the objective function of the corresponding test grammars, and we choose the grammar with the best value of the objective function. This is the grammar which will be incorporated in the next generation of the population of grammars. When the chosen grammar is the tested grammar no changes occur, and the corresponding grammar population entity is preserved in the population of grammars. Otherwise, the tested grammar population entity is removed from the population of grammars and is substituted by the test grammar population entity with the best value of the objective function.

The fourth and final phase of the algorithm is performed after all evolution cycles have ended. In this phase, we compare the values of the objective function calculated for each grammar within the population of grammars, and we choose the grammar with the highest or lowest value, depending on our objective. This is the solution grammar, and as such is the result of automated refactoring.

5 Experimental Results

5.1 mARTINICA Implementation

In order to be able to perform experiments and demonstrate the correctness of our approach, we implemented a grammar refactoring system in which mARTINICA plays a central role. The entire system is implemented in Java, and its architecture is shown in fig 3.

The refactoring system takes the initial grammar to be refactored and the objective function from two different text files and, after refactoring has been performed, it creates two files. The first of these is a text file containing the resulting grammar, and the second is a pdf file containing the evolution report.

The core of the systems is divided into two coexisting entities: an automated refactoring algorithm, and an objective function evaluator. The automated refactoring algorithm contains the implementation of the entire mARTINICA algorithm, the refactoring process base and the interactive user interface for obtaining the number of evolution cycles, the population size and the length of life of the generation. The initial grammar is taken from the text file, parsed by the grammar parser, which creates a grammar model on the basis of which the refactoring is done. The objective function is parsed by the objective function evaluator, which calculates the values of the objective function for all grammars provided by the automated refactoring

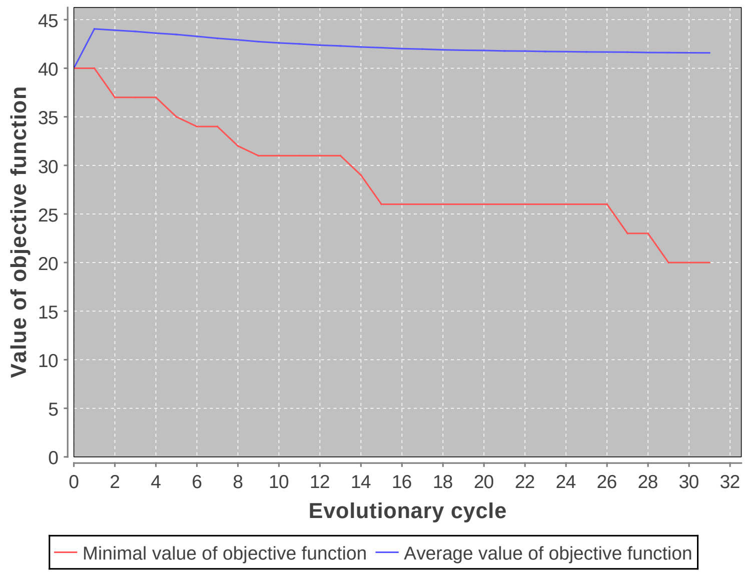


Figure 4 Values of the objective function through evolution.

algorithm. It does not provide an automated refactoring algorithm with a refactoring objective, since the algorithm always assumes that the objective is minimization, and if this is false the objective function evaluator transforms the state function so that it is equivalent to the native state function, but with the objective of minimization.

The entire refactoring process is monitored by the evolution monitor, which creates a report containing some analytical data concerning the specific refactoring process.

5.2 Refactoring Experiment

Until now, we have successfully performed refactoring tasks on small and medium-size grammars of Pascal-like languages and parts of the Algol-60 programming language grammar. In this section we present results of refactoring experiment performed on context-free grammar generating a simple assignment language. This grammar consisted of 11 non-terminals, 13 terminals and 18 production rules expressed in BNF notation. In our experiment, the refactoring task was described by an objective function (1), while mARTINICA iterated through 30 evolutionary cycles, with a population of 500 grammar population entities, and the length of life of a generation was set to 4. The value of the objective function evaluated for the initial grammar was 40, while the value of the objective function evaluated for the refactored grammar is 20, which means that mARTINICA managed to reduce the value of the objective function by 50%, and thus fulfilled the refactoring task. However, this interpretation is subjective, since in general there is no strict dividing line concerning reduction ratio at which refactoring task is fulfilled or not fulfilled, and this ratio may vary depending on aims of each individual refactoring process. The development of the value of the objective function through the evolutionary cycles is illustrated in fig 4.

We have chosen to present this experiment, because it explicitly illustrates the ability of mARTINICA to perform grammar refactoring tasks which have significant impact on values of grammar's size metrics. Other experiments however suggest that impact of our automated refactoring process on grammars structural metrics [3] is less significant. This is predominantly caused by incapacity of our refactoring operators to impose changes on grammar's structural metrics. On the other hand, refactoring patterns enable algorithm's users to extend base of refactoring operators, and create operators that suit their specific needs and requirements.

6 Grammar Refactoring Patterns

In our view, each grammar refactoring pattern provides equivalent transformation on context-free grammars and in this sense concept of grammar refactoring patterns is closely related to concept of refactoring operators. However there are some key differences between grammar refactoring patterns and refactoring operators. First of all, refactoring operators provide problem-independent transformations, while grammar refactoring patterns provide problem-specific transformations. This means that refactoring operators provide general transformations, whose usage is not bound by any specific class of refactoring tasks, while grammar refactoring patterns provide domain-specific transformations, intended for tackling the issues of particular class of refactoring problems. Secondly, each of our refactoring operators can be applied on arbitrary context-free grammar, including the situation when form of particular grammar does not allow specific transformation to occur, in which case original grammar form is returned as a result of a transformation. On the other hand, each grammar refactoring pattern prescribes some specific pre-conditions that context-free grammar must fulfill in order to be transformable by particular refactoring pattern.

In our approach, each grammar refactoring pattern is represented as specification consisting of three elements, which are context, problem and solution. Problem determines situation in which transformation provided by specific pattern can occur, context describes consequences of transformation on quality attributes of a context-free grammar, while transformation itself is specified in solution part of a pattern. In this notion of refactoring patterns, each refactoring operator is in fact refactoring pattern which lacks of explicit specification of a problem and a context. Problem part of a grammar refactoring pattern is described in the terms of grammar's quality attributes, and structural properties of grammar's production rules. Context part of a pattern is described in the terms of variations in values of grammar's quality attributes.

6.1 Specification of Grammar Refactoring Pattern

For purpose of expressing grammar refactoring patterns and in order to incorporate them in refactoring process of mARTINICA algorithm, we propose a language for formal specification of refactoring patterns called pLERO (pattern Language of Extended Refactoring Operators). Each refactoring pattern in pLERO is specified using schema in Listing 1, which consists of pattern name and three sub-specifications which describe context, problem and solution.

Context describes the effect that the grammar transformation provided by a solution has on chosen grammar metrics, in terms of increase or decrease in their values. Specification of context in pLERO is a set of metric-impact pairs, while each metric-impact pair describes impact of refactoring pattern on specific grammar metric. Purpose of context specification is to define a class of refactoring tasks to fulfillment of which can certain pattern contribute.

We can interpret Listing 2 as: Application of this pattern can lead to decrease in number of left recursive rules, and also increase in number of production rules

Problem defines structural properties that some production rules of a context-free grammar must and must not have, and also quality attributes that a context-free grammar must exhibit, in order to be transformed by instance of a refactoring pattern. By structure of grammar's production rule we mean order of specific terminal and non-terminal symbols on

Listing 1 Schema of grammar refactoring pattern specification.

```
PATTERN: [Pattern name]
        CONTEXT:
                [Context specification]
        END_CONTEXT
        PROBLEM:
                [Problem specification]
        END_PROBLEM
        SOLUTION:
                [Solution specification]
        END_SOLUTION
END_PATTERN
```

the right side of a production rule and occurrence of specific non-terminal on the left side of a production rule.

In order to specify this structure we have coined the term meta-structures of production rules. Each meta-structure of production rule is a structural model of specific sequence of arbitrary terminal and non-terminal symbols. Sequence of specific terminal and non-terminal symbols can be assigned to particular meta-structure of production rule only in case when this sequence exhibits structural properties prescribed by this meta-structure. To each assignment of sequence of specific terminal and non-terminal symbols to specific meta-structure of production rule we refer to as meta-structure instantiation, and to each meta-structure to which particular sequence of specific terminal and non-terminal symbols has been assigned we refer as to instance of meta-structure of production rule. We distinguish between two types of meta-structures of production-rules, namely primitive and composite meta-structures. Three kinds of primitive meta-structures of production rules are meta-non-terminals, meta-terminals and meta-symbols, while meta-non-terminal corresponds to arbitrary non-terminal symbol, meta-terminal corresponds to arbitrary terminal symbol and meta-symbol corresponds to arbitrary symbol of context-free grammar, regardless of the fact is this symbol terminal or non-terminal. Instance of meta-non-terminal is a specific non-terminal symbol, instance of meta-terminal is a specific terminal symbol and instance of meta-symbol is either a specific terminal symbol or a specific non-terminal symbol of a context-free grammar. Composite meta-structures are structural models made of primitive meta-structures. We propose one kind of composite meta-structure, which is meta-closure. Each meta-closure is a sequence of repeating primitive meta-structures, while instance of particular meta-closure is specific sequence of terminal and non-terminal symbols. Two meta-structure instances can be compared only if they were instantiated on the basis of a same meta-structure, and they are equivalent only when they refer to the same sequence of a specific terminal and non-terminal symbols, otherwise they are not equivalent.

Structure of right side of context-free grammar's arbitrary production rule can be described by a specific sequence of meta-structures, and each production rule can be viewed as sequence

Listing 2 Example of pLERO context specification.

```
CONTEXT
        minimizes countOfLeftRecursiveRules;
        maximizes prod;
END-CONTEXT
```

Listing 3 Example of pLERO declarations specification.

```
DECLARATIONS:
        Nonterminal1: NONTERMINAL;
        ArbitrarySequence1: CLOSURE ('ANY_SYMBOL', 0);
        ArbitrarySequence2: CLOSURE ('ANY_SYMBOL', 0);
        ArbitrarySequence3: CLOSURE ('ANY_SYMBOL', 0);
        ArbitrarySequence4: CLOSURE ('ANY_SYMBOL', 0);
END-DECLARATIONS
```

of meta-structure instances on its right side and an instance of meta-non-terminal on its left side.

Specification of a problem in pLERO consists of a four parts, which are declarations, positive-match, negative-match and forces.

In declarations part all meta-structure instances and composite meta-structures of grammar production rules are specified. In meta-structure instances specification we always assume that two or more different, but comparable meta-structure instances are always not equivalent. This means that if we want to allow situation where more meta-structure instances specify same sequence of terminal and non-terminal symbols, we have also to specify the number of composite meta-structures that is equivalent to the count of such meta-structure instances, and assign each meta-structure instance to different meta-structure.

We can interpret Listing 3 as: Declaration of one meta-non-terminal instance called Nonterminal1, Declaration of four composite meta-structures which can be matched against any number (including zero) of any grammar symbols, and declaration of one instance of each composite meta-structure. These are the only meta-structures which can be used for specification of structural properties of a context-free grammar in entire problem part of pLERO specification.

In positive-match part of pLERO problem specification, structural properties that some subset of grammar's production rules must exhibit are defined. Positive-match is defined as a set of production meta-rules. Each production meta-rule defines a structure of one production rule, and it consists of label, left side and right side of production meta-rule. Label is unambiguous identifier of production meta-rule and enables us to manipulate with whole grammar's production rule whose structure is represented by given meta-rule, while this manipulation occurs in solution part of a pLERO specification. Left side of production meta-rule is some meta-non-terminal and right side of production meta-rule is some sequence of meta-structure instances.

In order to some grammar exhibit required structural properties it must contain production rules whose structural properties match with each of production meta-rules specified in positive-match. To matching of production rule to production meta-rule we refer as to production rule labeling. Each grammar's production rule can be labeled with at most one production meta-rule, and each production-meta rule can be used for labeling at most one production rule. This however can lead to two kinds of non-determinism, non-determinism in selecting of the production rule which will be labeled, and non-determinism in selecting of production meta-rule by which production rule should be labeled. In case when there are multiple production rules which match one production meta-rule and in case when there are multiple production meta-rules to which one production rules matches, sophisticated strategy for conflict resolution is required. Specific context-free grammar exhibits required structural properties only in case when all production meta-rules have been used for labeling, but in this case not all production rules have to be labeled.

Listing 4 Example of pLERO positive-match specification.

```
POSITIVE-MATCH:
[Rule1] Nonterminal1 :: Nonterminal1 ArbitrarySequence1;
[Rule2] Nonterminal1 :: ArbitrarySequence2;
END-POSITIVE-MATCH
```

Listing 5 Example of pLERO negative-match specification.

```
NEGATIVE-MATCH:
Nonterminal1 :: ArbitrararySequence3 Nonterminal1 ArbitrarySequence4;
END-NEGATIVE-MATCH
```

We can interpret Listing 4 as: Grammar must contain at least two production rules whose left side is the same non-terminal. Right side of one of these rules must also start with this non-terminal followed by arbitrary sequence of symbols, while right side of other rule is an arbitrary sequence of symbols. Rule1, and Rule2 are labels of production meta-rules.

In negative-match part of pLERO problem specification, structural properties that some subset of grammar's production rules must not exhibit are defined. Negative-match is defined similarly as positive-match, and it is represented by a set production meta-rules, however in this case all production meta-rules are without labels. In order to grammar exhibit structural properties that prevent transformation provided by a pattern solution, similarly as in positive-match all production meta-rules in negative-match must be used for labeling.

Labeling of production rules with production meta-rules specified in positive-match, and labeling of production rules with production meta-rules specified in negative-match are two separate but interlinked processes. This means that some production rule can be labeled with one production meta-rule of positive-match and at the same time this production rule can be labeled with one production meta-rule of negative-match. However some meta-structure instance that occur in arbitrary production meta-rule of positive-match and same meta-structure instance occurring in arbitrary production meta-rule of negative-match represents in both cases same sequence of same symbols.

We can interpret Listing 5 as: Grammar does not contain rule whose left side is instance Nonterminal1 and whose right side containts this instance.

In forces part of pLERO problem specification additional quality attributes that grammar must posses are defined. This quality attributes are expressed using relational expressions, whose variables are grammar metrics, and logic operators between these expressions. In this part of specification meta-structure instances can also be used, as arguments of grammar metrics. From a global point of view pLERO specification of forces is one logic expression, which if evaluated as true, indicates that grammar possesses required quality attributes, and if evaluated as false, shows that grammar does not exhibit required quality attributes.

We can interpret Listing 6 as: Grammar contains exactly two production rules whose left side is non-terminal matched against meta-non-terminal instance Nonterminal1.

Solution in pLERO provides transformation on context-free grammar, in case that this

Listing 6 Example of pLERO forces specification.

```
FORCES:
        RulesLeftSide(Nonterminal1) = 2
END-FORCES
```

Listing 7 Example of pLERO solution specification.

```
SOLUTION:
INTRODUCE_NONTERMINAL (Nonterminal2);
REMOVE_PRODUCTION (Rule1);
REMOVE_PRODUCTION (Rule2);
INTRODUCE_PRODUCTION ([New1]
                Nonterminal1:: ArbitrarySequence2 Nonterminal2);
INTRODUCE_PRODUCTION ([New2]
                Nonterminal2:: ArbitrarySequence1 Nonterminal2);
INTRODUCE_PRODUCTION ([New3] Nonterminal2:: EMPTY_SYMBOL);
END-SOLUTION
```

grammar possesses structural properties defined in positive-match of problem specification, exhibits quality attributes defined in forces specification and does not posses structural properties defined in negative-match of problem specification. This transformation can be only related to meta-structure instances used in positive-match specification, and production rules which have been labeled. To specify this transformation we use relatively simple imperative language, which manipulates with meta-structures and production meta-rule labels as with constants, and allows execution of some operations on context-free grammar. Since transformation provided by solution is in fact refactoring operator, here we will not discuss it in detail.

We can interpret Listing 7 as: Introduce new non-terminal of grammar and match it against meta-non-terminal instance Nonterminal2, remove productions with labels Rule1 and Rule2 and introduce three new production rules. Along with Listing 2, Listing 3, Listing 4, Listing 5, Listing 6 this solution specifies a method of removing left-recursion in case when one grammar's production rule is left recursive and other is not.

7 Conclusion

In this paper, we discussed universal algorithm for grammar refactoring, as well as unified method for preservation and automated application of language engineer's knowledge. As such, our refactoring approach presents appropriate basis for creation of new theory concerning automated task-driven grammar refactoring, while provided experimental results explicitly demonstrate correctness and effectiveness of this approach. However, achievement of this goal also requires deeper understanding and intensified research in refactoring operators, as well as quality-based grammar metrics. Crucial part of this research are grammar refactoring patterns, since they operate with knowledge derived from experience of language engineers and thus they present appropriate tool for converging of state-of-art and state-of-practice in the field of grammar refactoring. Possibility to simply enlarge base of refactoring operators and grammar refactoring patterns greatly benefits to adaptability of our algorithm and also contributes to significant degree of our approach's flexibility.

In future we would like to focus on resolving some known issues concerning our approach, such as relatively slow propagation of positive changes within the population of grammars, which is indicated by the fact that gap between relative quality of a best found grammar and average quality of grammars within the population grows in each evolutionary cycle, which is the phenomena that can be clearly observed on Fig. 4. We would also like to focus on achieving greater abstraction power of pLERO language, since currently count of production rules on which refactoring pattern operates is limited by number of production meta-rules contained in positive-match part of pLERO problem specification.

However, our vision goes even far, since mARTINICA currently covers only one aspect of grammar adaptation, e.g. grammar refactoring, while ultimate goal is creation of universal approach covering other processes concerning grammarware engineering e.g. grammar construction and grammar destruction.

References

1 P. Klint, R. Lämmel and C. Verhoef. *Toward an engineering discipline for grammarware*. ACM Transactions on Software Engineering Methodology, Vol.14, No.3, 2005, pp. 331-380.

2 R. Lämmel.*Grammar Adaptation*. In Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME '01), 2001, J. Oliveira and P. Zave (Eds.). Springer-Verlag, London, UK, pp. 550-570.

3 J. Cervelle, M. Crepinsek, R. Forax, T. Kosar, M. Mernik and G. Roussel. *On defining quality based grammar metrics*. In Proceedings of IMCSIT '09. International Multiconference (IMCSIT '09), 2009, M. Ganzha and M. Paprzycki (Eds.). IEEE Computer Society Press, Los Alamitos, USA, pp. 651-658.

4 T. Mogensen. *Basics of Compiler Design*. University of Copenhagen, Copenhagen, DK, 2007.

5 T.L. Alves and J. Visser. *A Case Study in Grammar Engineering*. In Proceedings of 1st International Conferenceon Software Language Engineering (SLE' 2008), 2008, D. Gašević, R. Lämmel and E. Wyk (Eds.). Springer-Verlag, Berlin-Heidelberg, pp. 285-304.

6 M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley, Boston, USA, 1999.

7 M. Mernik, D. Hrncic, B.R. Bryant, A.P. Sprague, J. Gray, L. Qichao and F. Javed. *Grammar inference algorithms and applications in software engineering*. In Proceedings of ICAT 2009. XXII International Symposium (ICAT 2009), 2009, A. Salihbegović, J. Velagić, H. Šupić and A. Sadžak (Eds.). IEEE Computer Society Press, Los Alamitos, USA, pp. 14-20.

8 A. D'ulizia, F. Ferri, and P. Grifoni. *A Learning Algorithm for Multimodal Grammar Inference*. Trans. Sys. Man Cyber. Part B, Vol.41, No.6, 2011, pp. 1495-1510.

9 N.A. Kraft, E.B. Duffy and B.A. Malloy. *Grammar Recovery from Parse Trees and Metrics-Guided Grammar*. Software Engineering, Vol.35, No.6, 2009, pp. 780-794.

10 R. Läammel and C. Verhoef. *Semi-Automatic Grammar Recovery*. Software: Practice and Experience, Vol.31, No.15, 2001, pp. 1395-1438.

11 W. Lohmann, G. Riedewald and M. Stoy. *Semantics-preserving Migration of Semantic Rules During Left Recursion Removal in Attribute Grammars*. Electron. Notes Theor. Comput. Sci., Vol.110, 2004, pp. 133-148.

12 K.C. Louden. *Compiler Construction: Principles and Practice*. PWS Publishing, Boston, USA, 1997.

13 I. Halupka, J. Kollár. *Evolutionary algorithm for automated task-driven grammar refactoring*. In Proceedings of International Scientific Conference on Computer Science and Engineering (CSE 2012), 2012, pp. 47-54.

14 C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, USA, 1979.

15 F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. John Wiley & Sons, New York, USA, 1996.

Defining Domain Language of Graphical User Interfaces

Michaela Bačíková, Jaroslav Porubän, and Dominik Lakatoš

Department of Computers and Informatics
Technical University of Košice
Letná 9, 042 00 Košice, Slovakia
`{michaela.bacikova, jaroslav.poruban, dominik.lakatos}@tuke.sk`

Abstract

Domain-specific languages are computer (programming, modeling, specification) languages devoted to solving problems in a specific domain. The least examined DSL development phases are analysis and design. Various formal methodologies exist, however domain analysis is still done informally most of the time. There are also methodologies of deriving DSLs from existing ontologies but the presumption is to have an ontology for the specific domain. We propose a solution of a user interface driven domain analysis and we focus on how it can be incorporated into the DSL design phase. We will present the preliminary results of the DEAL prototype, which can be used to transform GUIs to DSL grammars incorporating concepts from a domain and thus to help in the preliminary phases of the DSL design.

1998 ACM Subject Classification D.2.11 Software Architectures, D.2.12 Interoperability, D.2.13 Reusable Software, H.1.2 User/Machine Systems, H.5.2 User Interfaces (D.2.2, H.1.2, I.3.6)

Keywords and phrases graphical user interfaces, domain analysis, formalization, domain-specific languages, DEAL

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.187

1 Introduction

Programming languages are used for human-computer interaction. Domain-specific languages (DSLs) such as Latex, SQL, BNF, are computer languages tailored to a specific application domain [15, 30, 48, 41, 21, 19]. In contrast, general-purpose languages (GPLs) such as Java, C and C# are designed to solve problems in any problem area.

When developing a new software, a decision must be made as to which type of programming language will be used: GPL or DSL. The issue is further complicated if an appropriate DSL does not exist. The reasons for using DSLs instead of GPLs are: easier programming, reuse of semantics, easier verification and understandability (and programmability) for end-users [15, 30]. However, the cost of developing a new DSL is usually high [15, 22] because it involves development of language parsers and generators along with the language.

A DSL should be developed whenever it is necessary to solve a problem that belongs to a problem family and when we expect more problems from the same family of problems to appear in the future. The implementation phase is well documented by many researchers [21] but the analysis and design phases are still dropped behind. The various DSL development phases and the tool support of each of them is very nicely described in the article by Čeh et al [10].

On the other hand, various methodologies for domain analysis (listed in the related work section) were developed such as DSSA, FODA, ODM. But they are often not used due to their complexity and the DA is done informally. This complicates the development od DSLs.

Even if a formal methodology is used for performing DA there are no clear guidelines on how the output from DA can be used in a language design process. Guillermo et al. [4] identified the following types of DA outputs:

a) *a domain model describing the target domain* - contains domain dictionary, terms, concepts, their relations and properties,
b) *a domain model describing how to develop systems* in the target domain.

The concrete implementations of the DA outputs vary when referring both to the first or the second type specifically. We identified the following DA output types:

- *simple domain dictionary* - contains terms, hierarchy of terms,
- *lexicographical dictionary* - terms, lexicographical relations (such as hyponymy, synonymy, etc.),
- *ontology* - terms, hierarchy, relations (such as is-a, part-of, has-part, regulates, exclusive-to relations, etc.),
- *feature diagram* - focuses on feature commonalities and variabilities in product lines, contains relations between the features representing the cases whether they are used in the product or not (and, or and alternative),
- *design documentation* - describing the system model or meta-model (class diagram, entity-relationship model, data-flow diagram, state-transition model, etc.),
- *reusable software artifacts* - fragments of code, libraries, reusable frameworks, etc.,
- *EBNF* - a grammar defining the domain language syntax.

Not each of these outputs can be used as an input to the DSL design phase. As proposed by Čeh et al. [10], it is possible to use ontologies. Their paper outlines the method of transformation of ontologies into DSL grammars and they have implemented the Ontology2DSL framework, which is able to demonstrate this transformation. By this they try to contribute to the first phases of DSL development, analysis and design.

However, the assumption for their solution is the existence of an ontology designed for the given specific domain. Which is actually an equally difficult problem compared to finding an existing DSL for the given specific domain.

In this paper we focus on a similar problem, but in the area of existing software systems. Many formal methods exist for analyzing the existing software systems with the goal of performing domain analysis (examples of them are listed and described in the section 6). The results in many cases represent a model or a design of a new software system (such as the class diagram); or a library of reusable software artifacts; both hardly usable for the DSL analysis or design.

The reason is that it is very difficult to filter out the implementation details from the target application so only the relevant domain information would remain. Therefore instead of trying to extract domain information, the existing methods rather benefit from the system's implementation details by extracting reusable code fragments or libraries and software documentation.

Thus the existing approaches for extracting domain-relevant information in a form of domain descriptions focus rather on existing informal documents and domain experts as sources of domain knowledge. The main drawback of such resources is their non-formal nature. Documents are mainly processed by complicated techniques such as Natural Language Processing (NLP) and it is necessary to verify the results manually. On the other hand, the information from domain experts has to be gained by personally meeting with them because often-times they are not willing to (nor they are capable of) writing the information in a semi-formal manner by themselves. The consecutive process of formalization of the gained data is even more tedious.

In our approach we do not focus on the extraction of domain information from the existing software systems in general. Our main points of interest are graphical user interfaces (GUIs) made of components. In the scope of supporting the early DSL development phases, our assumption is the existence of a user interface for the given specific domain, which nowadays is nothing unusual. Often-times there is a need to create a new software system and the obsolete version is thrown away without any regards to its reuse possibilities. There is almost an endless number of component-based applications for different domains and even a bigger number of web applications.

But the amount of existing software systems is *certainly larger* than the amount of existing ontologies. Therefore we claim that our approach is more efficient than the approach of Čeh et al. However the benefit resulting from their approach - no need to start from scratch but with a generated DSL grammar - is preserved in our approach.

In this paper we propose a formal design of a methodology for deriving DSL grammars from existing user interfaces. We also present the DEAL method for traversing GUIs and the preliminary results of the DEAL prototype, which was designed for the purpose of domain information extraction from GUIs.

The research thesis for this paper stands as follows: If an application exists for a specific domain with a GUI made of components and we have reflection available along with the possibility of identifying the component structure, then it is possible to design a tool, which uses reflection, and which can traverse the application GUI and create a DSL design from it. This DSL design can be then edited by a language designer.

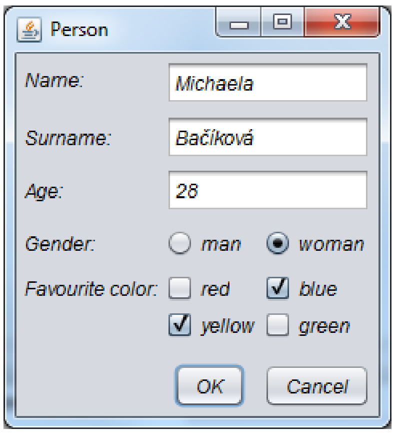
The presumption for using DSLs is an existence of a family of a repetitive problem. If the repetitive problem is a creation of a new DSL, creation of a new GUI or writing/performing commands in GUIs, then this paper tries to propose a partial solution for this problem.

The organization of this paper is as follows. Section 2 is intended to demonstrate an example of a GUI to DSL transformation. Section 3 presents the transformation rules used for generating a a DSL from a GUI and from different GUI components. Sections 4 and 5 present the DEAL method for traversing GUI components and the DEAL tool prototype which is an implementation of the DEAL method. The work related to this paper and the conclusion are summarized in Section 6.

2 GUI is a Language Definition

What do we see when we look at any user interface? Windows, dialogs, buttons, menus, labels, textfields, components. However a GUI is more than that, it is a definition of a DSL. We will try to demonstrate this fact on a simple example.

Let us look at fig. 1 with a form containing information about a person. Its concrete syntax could be specified using a grammar, where the non-terminals are noted by first capital letter and the terminals are noted in ⟨ ⟩ brackets. The elements ⟨STRING⟩ and ⟨NUMBER⟩ represent a terminal string or numeric value. The concrete syntax of the domain language



Name:	Michaela	
Surname:	Bačíková	
Age:	28	
Gender:	○ man	● woman
Favourite color:	☐ red	✓ blue
	✓ yellow	☐ green

Figure 1 An example of a person form.

for the person form can be defined as follows:

$$\begin{aligned}
Person &\rightarrow \langle\text{Person}\rangle Name\ Surname\ Age\ Gender\ FavouriteColor\\
Name &\rightarrow \langle\text{Name}\rangle\ \langle\text{STRING}\rangle\\
Surname &\rightarrow \langle\text{Surname}\rangle\ \langle\text{STRING}\rangle\\
Age &\rightarrow \langle\text{Age}\rangle\ \langle\text{STRING}\rangle\\
Gender &\rightarrow \langle\text{Gender}\rangle\ \langle\text{man}\rangle \mid \langle\text{woman}\rangle\\
FavouriteColor &\rightarrow \langle\text{Favourite colors}\rangle\ (\ \langle\text{red}\rangle?\ \langle\text{blue}\rangle?\ \langle\text{green}\rangle?\ \langle\text{yellow}\rangle?\ )
\end{aligned}$$

If the *age* text field was a formatted text field, or a spinner (fig. 2 on the right), we could extract the input type: string, number or date. The grammar rule for age could then look as follows:

$$Age \rightarrow \langle\text{Age}\rangle\ \langle\text{NUMBER}\rangle$$

We showed that the GUI defines a language. What is the sentence in this language? And where is the semantics?

A sentence in this language could look as follows:

$$Person\ Name\ Michaela\ Surname\ Bacikova\ Age\ 28\ Gender\ woman$$
$$Favourite\ colors\ (blue\ yellow)$$

And writing such sentences in the GUI language means filling the form with values.

The second question is the semantics, which is defined by the application. The semantic meaning of clicking the *OK* button is defined in the code. So it holds for every single component contained in a UI.

In the next section we will describe how the GUI language specification can be automatically created from an existing user interface.

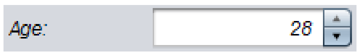


Figure 2 The *age* input field represented by a spinner with a numeric value.

3 Method for DSL Specification Derivation

For a better explanation of the process of the domain language specification derivation, we present a sketch of transformation for basic Java components. This is to be a formal description of our extraction method called *DEAL*. The transformation is defined as a semantic function, which expresses the domain meaning with the goal of creating domain grammar. The transformation function is defined as follows:

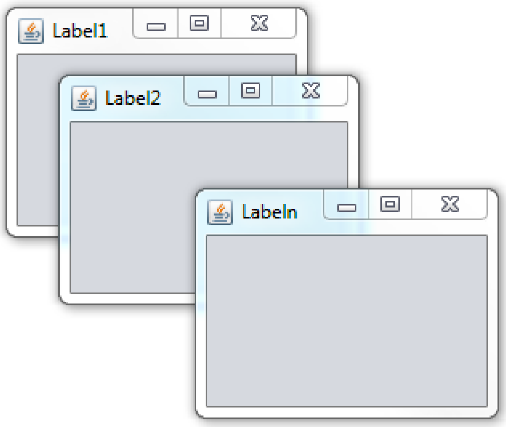
$$\mathcal{T} : Component \times ComponentTree \longrightarrow Production$$

where $\mathcal{T}$ is the transformation function, *Component* is the semantic domain of components and *Production* is the semantic domain of EBNF production rules. In order to extract any information from components we also have to know their location in the tree, to get the information about the component's parents and child components. Therefore the input of the semantic function also involves the *ComponentTree* semantic domain of component hierarchies in a form of trees. For example, in tabbed pane the tabs are located in the tabbed pane and their tab names (labels) are stored in the tabbed pane. Therefore we need to know the information from the parent component and store it to the child component.

Here we will present the transformation rules. Each of the rules is always labelled with the component name above the left semantic bracket.

We assume that the processing always starts with a GUI of an application which is component-based. Each UI should consist of one or more *scenes*[1]. Opening of a new scene can be initialized by starting the application or by performing action on a functional component.

$$\mathcal{T}\overset{GUI}{[\![}$$

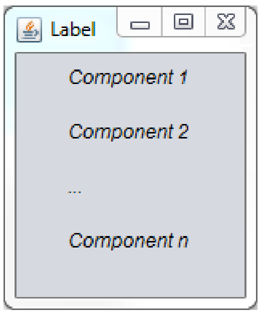


$$]\!] = GUI \rightarrow Label_1\ Label_2 \ldots Label_n$$

Each scene can be identified by its identifier (if there is any), e.g. a window title, a dialog title, a web page title. This holds for each component: it is identified by its name or description. We note the identifier of a scene or of a component generally as "*Label*" and it represents a string of characters. Each scene can contain a graphical content - a list of components.

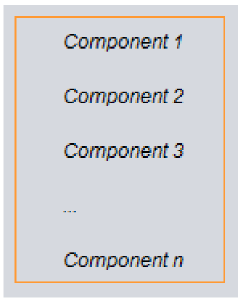
[1] As defined by Kösters in [23].

Scene

$\mathcal{T}$ $\Bigg[\!\!\Bigg[$  $\Bigg]\!\!\Bigg]$ $= \ Label \rightarrow \langle\text{Label}\rangle\ Component_1\ Component_2 \ldots Component_n$

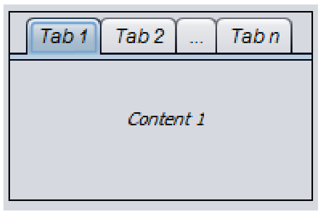
The components can be logically arranged in an unlimited number of containers. Containers are components which can contain other components (e.g. panel or a tabbed pane).

Panel

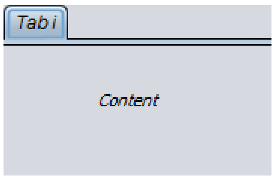
$\mathcal{T}$ $\Bigg[\!\!\Bigg[$  $\Bigg]\!\!\Bigg]$ $= \ Panel \rightarrow Component_1\ Component_2 \ldots Component_n$

From a tabbed pane, additional information can be extracted. Since the containers are stored in tabs and these tabs have a tab name (Tab_i), the tab name can be extracted as an identifier of the group of components contained in the given panel.

Tabbed pane

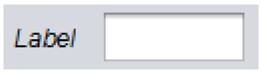
$\mathcal{T}$ $\Bigg[\!\!\Bigg[$  $\Bigg]\!\!\Bigg]$ $= \ TabbedPane \rightarrow Tab_1\ Tab_2 \ldots Tab_n$

Tab

$\mathcal{T}$ $\Bigg[\!\!\Bigg[$  $\Bigg]\!\!\Bigg]$ $= \ Tab \rightarrow \langle\text{Tab}\rangle\ Content$

Textfields are components which usually have no label integrated in their implementation. However the label can be (optionally) connected to them in the implementation phase by using the *for* attribute in the label component (the for attribute is supported e.g. by the Java language and by HTML) where the reference to the described component is inserted. This holds for any other type of component. Moreover, if the textfield is a formatted textfield, the type of the input value (string, numeric, date, etc.) and restrictions on the input text can be extracted (such as regular expression or min/max value).

$Textfield$

$\mathcal{T}\Bigg\llbracket$ 

$\Bigg\rrbracket = Label \rightarrow \langle\text{Label}\rangle \quad \langle\text{STRING}\rangle$

We realize that setting the label *for* attribute is an optional issue, however if the programmer does not set it, we perceive it as a usability issue. Once the label was assigned to the *for* component and saved, it has no meaning in the result of the domain analysis process therefore it can be thrown away. The important issue is the component the label describes.

$Label$

$\mathcal{T}\Bigg\llbracket$

$\Bigg\rrbracket = Label \rightarrow \epsilon$

The functional components (such as buttons, menu items, web links, etc.) have a similar rule. The reason for excluding functional components is that the labels are not important for the domain. The semantics of functional components is the execution of the action in GUI, which is defined by them and stored in the code. We see the execution of events in the user interface as a completely different chapter, which belongs rather to the areas of usability and automated GUI testing.

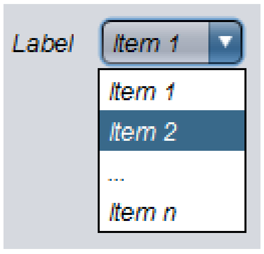
$Button$

$\mathcal{T}\Bigg\llbracket$

$\Bigg\rrbracket = Label \rightarrow \epsilon$

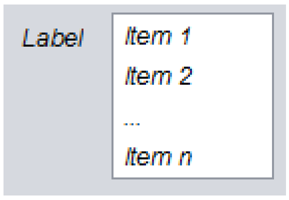
The comboboxes, lists and checkbox and radio button lists are examples of components (or lists of components) representing *lists of terms*. Depending on whether the component (or a group of components) has a single selection or multiple selection set, the relations between the terms are derived. Mutual exclusive relation (single selection) is represented by alternatives, terms that are not mutually exclusive (multiple selection) are represented by one or zero occurence (*?*) of each term.

$Combobox$

$\mathcal{T}\Bigg\llbracket$ 

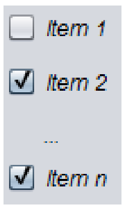
$\Bigg\rrbracket = Label \rightarrow \langle\text{Label}\rangle \ (\langle\text{Item}_1\rangle \mid \langle\text{Item}_2\rangle \mid \ldots \mid \langle\text{Item}_n\rangle)$

$List$

$\mathcal{T}\Bigg\llbracket$ 

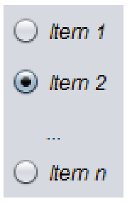
$\Bigg\rrbracket = Label \rightarrow \langle\text{Label}\rangle \ (\langle\text{Item}_1\rangle \mid \langle\text{Item}_2\rangle \mid \ldots \mid \langle\text{Item}_n\rangle)$

Checkbox list



$$\mathcal{T}\left[\!\left[\quad \right]\!\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle \quad \langle\text{Item}_1\rangle\,? \quad \langle\text{Item}_2\rangle\,? \ \ldots \ \langle\text{Item}_n\rangle\,?$$

Radio button list

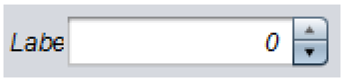


$$\mathcal{T}\left[\!\left[\quad \right]\!\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle \ \left(\langle\text{Item}_1\rangle \mid \langle\text{Item}_2\rangle \mid \ldots \mid \langle\text{Item}_n\rangle \right)$$

In the examples the list was initialized with single selection. If it would have been initialized with multiple selection the rule would be the same as for the checkbox list.

Spinners (like formatted textfields) in the Java language are instantiated with a model defining the restrictions on the data which are stored in them. The spinner model can be chosen of three basic types: List, Number and Date. Each of the model types has its particular type set and if it is a Number type, the data type of the number can be extracted (Float, Double, Integer, Binary, etc.).

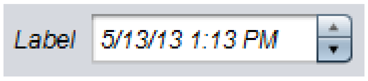
Spinner



$$\mathcal{T}\left[\!\left[\quad \right]\!\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle \quad \langle\text{NUMBER}\rangle$$

If the spinner has default values of a list of items, then it has the same rule as any component with single selection (alternatives). In the model, the minimum, maximum and default values are also set and this information can be extracted as additional information about the term.

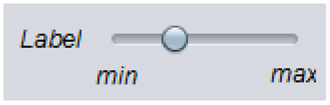
Date spinner



$$\mathcal{T}\left[\!\left[\quad \right]\!\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle \quad \langle\text{DATE}\rangle$$

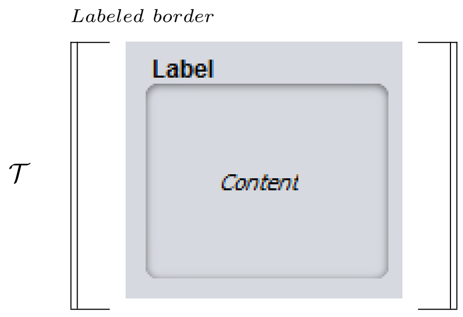
Sliders are similar to spinners, with the difference of displaying the chosen value not by textual representation, but on a graphical scale. Maximum and minimum values can be extracted.

Slider



$$\mathcal{T}\left[\!\left[\quad \right]\!\right] = \textit{Label} \rightarrow \langle\text{Label}\rangle \quad \langle\text{NUMBER}_{\text{min-max}}\rangle$$

In many types of containers (such as panels) the label (or title) attribute is optional. Therefore if a group of components (representing terms) is logically related, it is not clear what exactly they have in common, whether they belong to the same subdomain. However some containers have identifiers which give a clue about the subdomain of their content. The example of such a container is a panel with the labeled border. Each scene should have a label (window title, dialog title, webpage title, etc.) and this name gives a clue about the application domain or subdomain. Each component contained in the scene belongs to this domain. This way the hierarchy of terms is created.



$$= \quad \mathit{Label} \rightarrow \langle\text{Label}\rangle\ \ \mathit{Content}$$

According to the illustrated rules we can see that on various component types apply various extraction conditions and it is possible to derive basic relations between the terms and properties of terms such as default value, minimum and maximum value and restrictions. A hierarchy of domains and subdomains (a part-of or a belongs-to relation) is created by containers placed in other containers and in scenes. The result of the extraction method is a formalized domain model. The $\langle\text{STRING}\rangle$, $\langle\text{NUMBER}\rangle$ and $\langle\text{DATE}\rangle$ could be defined as follows:

$\langle\text{STRING}\rangle \rightarrow *$
$\langle\text{NUMBER}\rangle \rightarrow [0-9]+$
$\langle\text{DATE}\rangle \rightarrow [0-9][0-9]/[0-9][0-9]/[0-9][0-9]\ ([0-9][0-9]{:}[0-9][0-9]\ (\text{AM}|\text{PM}))?$

4 The DEAL Method

We have performed a research in the area of *automated domain analysis of user interfaces*, described in detail in [24] [8] and [6]. Our general goal was to prepare the first phase for automatized analysis of UI domain content by means of automatic extraction of domain content (terms), the properties of the concepts and structure and analysis of relations between the content units. The method of traversing GUIs and extraction of domain information from existing GUIs is called DEAL (Domain Extraction ALgorithm). The input of DEAL is an existing system programmed in a language, which provides the possibility of determining the component structure, reflection and/or aspect-oriented programming. The output of DEAL is a domain model displayed in fig. 3.

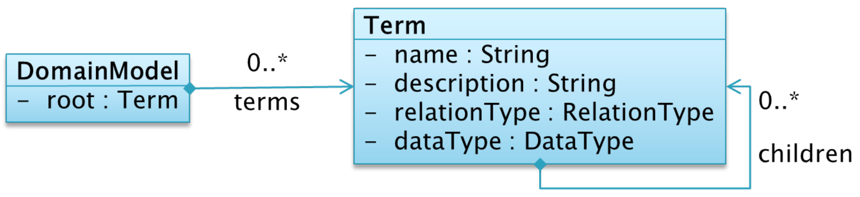


Figure 3 Domain model representation in DEAL.

The DEAL traversal algorithm was described in [6] and in the section 3 we described the method of transformation.

The domain content of the target UI is extracted as a graph of terms, their relations and properties. Some relations (such as the parent-son relation) are explicit, other relations (such as mutual exclusivity) are derived automatically based on the typology of components and, partially, based on their topology as outlined in this paper. From this domain model representation, different types of domain models, including grammars and ontologies, can be generated. We experimentally confirmed the possibility of extraction of a graph of domain terms including the derivation of relations between some terms on applications in Java language and we performed a feasibility analysis of web application analysis (HTML) [37, 6, 7]. Both languages meet the presumption to have the component nature, they provide the possibility to determine the component structure.

5 The DEAL Tool Prototype

The DEAL tool prototype[2] is a software solution for extracting domain content of existing user interfaces and it proves the possibility of using the DEAL method on Java applications. Currently, DEAL uses YAJCo[3] language processor to generate grammars. More about YAJCo can be found in [38].

The DEAL tool prototype proves that it is possible to:

- traverse the GUI of an application, which is made of components,
- extract domain information from an existing GUI in a formalized form,
- and to generate a DSL grammar based on the extracted information.

The DEAL prototype was tested on 17 open-source Java applications, all of which are included in the DEAL project online. It is however still in development and we are improving it based on the test results.

Some Java applications have their own class loaders, therefore we had to use AspectJ to be able to extract information from them. The tutorial, documentation and related references are all published online along with the tool[4].

An example of the DEAL output for the person form (fig. 1) is the grammar generated from the extracted model:

$$\begin{aligned}
Person &::= (\langle \text{Person}\rangle Name\ Surname\ Age\ Gender\ (Favourite\_color){*}) \\
Name &::= (\langle \text{Name}\rangle\ \langle \text{STRING\_VALUE}\rangle) \\
Surname &::= (\langle \text{Surname}\rangle\ \langle \text{STRING\_VALUE}\rangle) \\
Age &::= \langle \text{Age}\rangle\ \langle \text{STRING\_VALUE}\rangle \\
Gender &::= \langle \text{Gender}\rangle\ (\langle \text{man}\rangle \mid \langle \text{woman}\rangle) \\
Favourite\_color &::= \langle \text{Favourite color}\rangle\ (\langle \text{red}\rangle\ \langle \text{blue}\rangle\ \langle \text{green}\rangle\ \langle \text{yellow}\rangle\ )
\end{aligned}$$

The grammar is in YAJCo notation and the rules for not-mutually exclusive terms are supplemented by the $0 - n$ version because YAJCo does not support the ? operator. However the results show that generating DSL grammars from existing user interfaces is definitely possible.

[2] DEAL project can be found at: https://www.assembla.com/spaces/DEALtool
[3] YAJCo project can be found at: https://code.google.com/p/yajco/ and YAJCo maven project at: http://mvnrepository.com/artifact/sk.tuke.yajco
[4] DEAL project documentation and tutorials: https://www.assembla.com/spaces/DEALtool/wiki

6 Related Work

Here we briefly summarize the different approaches related to: domain analysis, ontology extraction, GUI modeling and semantic UIs and reverse engineering.

Domain Analysis The domain analysis was first defined by Neighbors [33] in 1980 and he stresses that domain analysis is the key factor for supporting reusability of analysis and design, not the code.

The most widely used approach for domain analysis is the *FODA* (Feature Oriented Domain Analysis) approach [17]. FODA aims at the analysis of software product lines by comparing the different and similar features or functionalities. The method is illustrated by a domain analysis of window management systems and explains what the outputs of domain analysis are but remains vague about the process of obtaining them. Very similar to the FODA approach, and practically based on it, is the *DREAM* (Domain Requirements Asset Manager) approach by Mikeyong et. al. [31]. They perform commonality and variability analysis of product lines too, but with the difference of using an analysis of domain requirements, not features or functionalities of systems. Many approaches and tools support the FODA method, for example Ami Eddi [12], CaptainFeature [1], RequiLine [2] or ASADAL [25]. Other examples of formal methodologies are ODM (Organization Domain Modeling) [45] and DSSA (Domain Specific Software Architectures) [47].

There are also approaches that do not only support the process of domain analysis, but also the reusability feature by providing a library of reusable components, frameworks or libraries. Such approaches are for example the early *Prieto-Díaz approach* [13] that uses a set of libraries; or the later *Sherlock* environment by Valerio et. al. [54] that uses a library of frameworks.

The latest efforts are in the area of *MDD* (Model Driven Development). The aim of MDD is to shield the complexity of the implementation phase by domain modelling and generative processes. The MDD principle support provides for example the Czarnecki project Clafer [5] and the FeatureIDE plug-in [50] by Thüm and Kästner.

ToolDay (A Tool for Domain Analysis) [27] is a tool that aims at supporting all the phases of domain analysis process. It has possibilities for validation of every phase and a possibility to generate models and exporting to different formats.

All these tools and methodologies support the domain analysis process by analysing data, summarizing, clustering of data, or modelling features. But the input data for domain analysis (i.e. the information about the domain) always comes from the users, or it is not specified where it actually comes from. Only the *DARE* (Domain analysis and reuse environment) tool from Prieto-Díaz [16] primarily aims at automatic collection and structuring of information and creating a reusable library. The data is collected not only from human resources, but also *automatically from existing source codes and text documents*. But as mentioned above, the source codes do not have to contain the domain terms and domain processes. The DARE tool *does not analyse the GUIs* specifically.

Last but not least, the approach most similar to ours is the one proposed by Čeh et al. [10]. They proposed a methodology of transforming existing ontologies into DSL grammars and they present the results of their Ontology2DSL framework. The disadvantage in comparing to our approach is the little amount of existing ontologies available when comparing to the amount of existing software systems.

Ontology Extraction Many approaches are targeted to ontology learning. Several methodologies for building ontologies exist, such as OTK, METHONTOLOGY or DILIGENT, but they target ontology engineers and not machines [9]. Many methods and different sources of analysis are used to generate ontologies automatically. Among the Results are almost always combined with a manual controlling and completing by a human and as an additional input, almost always some general ontology is present (a “core ontology”) serving as a “guideline” for creating new ontologies. Different methods are used to generate ontologies:

1. clustering of terms [44, 58, 39],
2. pattern matching [51, 58, 56, 39],
3. heuristic rules [51, 56, 39],
4. machine learning [34, 43],
5. neural networks, web agents, visualizations [39],
6. transformations from obsolete schemes [56],
7. merging or segmentation of existing ontologies [42, 58],
8. using fuzzy logic to generate a fuzzy ontology, which can deal with vague terms such as FFCA method ([40] and [11]) or FOGA method [49],
9. analysis of web table structures [35, 51, 26],
10. analysis of fragments of websites [57].

A condition for creating a good ontology is to use many sources as an input to analysis - structured, non-structured or semi-structured - and to use a combination of many methods [9]. Therefore as an additional mechanism for identifying different types of relationships (e.g. mutual exclusivity, hierarchical relations), WordNet web dictionary [35, 3, 14, 58] or other web dictionaries or databases available on the Web are used. A state of the art from 2007 can be found in [9].

GUI Modelling and Semantic User Interfaces Special models are designed specifically for modelling UIs or for modelling the interaction with UIs, whether they are older, such as CLASSIC language by Melody and Rugaber [32], or modern languages, such as XML, described in the review made by Suchon and Vanderdonckt in [46]. Paulheim [36] designed UI models of interaction with users. For UI configurations usually models such as configuration ontology designed for WebProtégé tool in [53] are used.

The most complex UI model was designed by Kösters in [23] as a part of the modelling process of the FLUID method for combined analysing of UIs and user requirements. A part of Kösters model is a domain model and model of UI (UIA-Model). Our model was slightly inspired by Kösters work - however we use domain-specific modelling without the relation to user requirements, therefore our model is simpler.

An interesting work was made in the area of semantic UIs by Porkoláb in [52].

Reverse Engineering Only a few works in the area of reverse engineering will be mentioned since our work is primarily focused on the area of domain analysis, not on reverse engineering. However, there are several works closely related to our work. Specifically, they are either reverse processes compared to ours (i.e. generate GUIs from domain models), or they produce other outputs than a DSL grammar:

- a GUI-driven generating of applications by Luković et al. in [29],
- generating of UIs based on models and ontologies by Kelshchev and Gribova in [18],
- deriving UIs from ontologies and declarative model specifications by Liu et al. in [28],
- program analysing and language inference [20].

A very interesting process is also seen in [55] where authors transform ontology axioms into application domain rules however the results are not as formal as our DSL grammar.

7 Conclusion

In this paper we presented the actual state of our research and introduced our DEAL method for extracting domain information from GUIs. We showed that the utilization of this method is not strictly in the area of the domain analysis, with the intent of creating a new software system. We argue that it is possible to define a domain-specific language. A GUI is a template that defines how the input accepted by the GUI should look like. According to this template it is possible to extract (generate) a GUI domain language specification, which, analogically, represents a *formalized* template for the GUI inputs – according to it, it is possible to determine whether a sentence belongs to the domain language defined by the GUI, or not.

The generated specification of the domain language defined by the UI can be utilized in further processes, such as automatic filling of forms. It happens in real cases, such as organizing conferences (or other events, where people need to register) where many people submit papers. The conference organizers get the information from a submitting system including the submitter names, surnames, personal information and information about their submitted papers. All this data have to be entered into the department software system and since there is no automated transfer system, they have to be entered manually. For each single submitter the whole form has to be filled manually again and again. And if there are 1000 submitters, we can imagine that it takes a lot of effort and time.

Using our approach, the conference organizer would automatically derive a grammar for the given language based on the GUI, and using this grammar it is possible to create a tool for automatic form filling. Moreover, this tool can be generated based on the grammar specification and it would also check the input for correctness.

Acknowledgements This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the Artifacts Written in Domain-specific Languages Driven by Language Evolution. Thanks to Milan Nosáľ for brainstorming and reviewing the paper.

References

1 Captainfeature, the webpage of captainfeature sourceforge.net project. https://sourceforge.net/projects/captainfeature, 2005. [Online 2013].
2 The webpage of requiline project. http://wwwlufgi3.informatik.rwth-aachen.de/TOOLS/requiline/, 2005. [Online 2013].
3 Y. J. An, J. Geller, Y. Wu, and S. A. Chun. Automatic generation of ontology from the deep web. In *Database and Expert Systems Applications, 2007. DEXA '07. 18th International Workshop on*, pages 470–474, Sept.
4 G. Arango. A brief introduction to domain analysis. In *Proceedings of the 1994 ACM symposium on Applied computing*, SAC '94, pages 42–46, New York, NY, USA, 1994. ACM.
5 K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: mixed, specialized, and coupled. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 102–122, Berlin, Heidelberg, 2011. Springer-Verlag.
6 M. Bačíková and J. Porubän. Analyzing stereotypes of creating graphical user interfaces. *Central European Journal of Computer Science*, 2:300–315, 2012.

7 M. Bačíková, J. Porubän, and Lakatoš D. Introduction to domain analysis of web user interfaces. In *Proceedings of the Eleventh International Conference on Informatics, INFORMATICS'2011*, pages 115–120, Rožňava, Slovakia, 2011.

8 Michaela Bačíková. Domain analysis with reverse-engineering for gui feature models. In *POSTER 2012 : 16th International Student Conference on Electrical Engineering*, volume 16, pages 1–5. Czech Technical University in Prague, May 2012.

9 I. Bedini and B. Nguyen. Automatic ontology generation: State of the art. Technical report, University of Versailles, December 2007.

10 I. Čeh, M. Crepinsek, T. Kosar, and M. Mernik. Ontology driven development of domain-specific languages. *Computer Science and Information Systems*, (2):317–342, 2011.

11 W. Chen, Q. Yang, L. Zhu, and B. Wen. Research on automatic fuzzy ontology generation from fuzzy context. In *Proceedings of the 2009 Second International Conference on Intelligent Computation Technology and Automation - Volume 02*, ICICTA '09, pages 764–767, Washington, DC, USA, 2009. IEEE Computer Society.

12 K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, GPCE '02, pages 156–172, London, UK, 2002. Springer-Verlag.

13 R. P. Díaz. Reuse Library Process Model. Final Report. Technical report start reuse library program, Electronic Systems Division, Air Force Command, USAF, Hanscomb AFB, MA, 1991.

14 Y. Ding, D. Lonsdale, D. W. Embley, and Li Xu. Generating ontologies via language components and ontology reuse. In *In Proceedings of 12th International Conference on Applications of Natural Language to Information Systems (NLDB'07*, 2007.

15 M. Fowler. *Domain-Specific Languages (Addison-Wesley Signature Series (Fowler))*. Addison-Wesley Professional, 1 edition, October 2010.

16 W. Frakes, R. Prieto-Diaz, and Ch. Fox. Dare: Domain analysis and reuse environment. *Ann. Softw. Eng.*, 5:125–141, January 1998.

17 K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

18 A. Kelshchev and V. Gribova. From an ontology-oriented approach conception to user interface development. In *ITHEA, International Journal ITA (Institue of Mathematics and Informatics, Bulgarian Academy of Sciences)*, volume 10. Institute of Information Theories and Applications FOI ITHEA, 2003.

19 J. Kollár and S. Chodarev. Extensible approach to dsl development. *Journal of Information, Control and Management Systems*, 8(3):207–215, 2010.

20 J. Kollár, S. Chodarev, E. Pietriková, Ľ. Wassermann, D. Hrnčič, and M. Mernik. Reverse language engineering: Program analysis and language inference. In *Informatics'2011 : proceedings of the Eleventh International Conference on Informatics*, pages 109–114. Košice, TU, November 16-18 2011.

21 T. Kosar, P. E. M. López, P. A. Barrientos, and M. Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, April 2008.

22 T. Kosar, N. Oliveira, M. Mernik, M. J. V. Pereira, M. Črepinšek, D. da Cruz, and P. R. Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7(2):247–264, May 2010.

23 G. Kösters, H. W. Six, and J. Voss. Combined analysis of user interface and domain requirements. In *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*, ICRE '96, pages 199–, Washington, DC, USA, 1996. IEEE Computer Society.

24 Michaela Kreutzová (Bačíková), Jaroslav Porubän, and Peter Václavík. First step for gui domain analysis: Formalization. *Journal of Computer Science and Control Systems*, 4(1):65–70, 2011.

25 Postech Software Engineering Laboratory. A review of asadal case tool.

26 X. Lei and R. Yong. Ontology generation from web tables: A 1+1+n approach. In *Proceedings of the 2010 International Forum on Information Technology and Applications - Volume 01*, IFITA '10, pages 234–239, Washington, DC, USA, 2010. IEEE Computer Society.

27 L. Lisboa, V. Garcia, E. de Almeida, and S. Meira. Toolday: a tool for domain analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 13:337–353, 2011.

28 B. Liu, H. Chen, and W. He. Deriving user interface from ontologies: A model-based approach. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '05, pages 254–259, Washington, DC, USA, 2005. IEEE Computer Society.

29 I. Luković, S. Ristić, A Popović, and P. Mogin. An approach to the platform independent specification of a business application. In *Central European Conference on Information and Intelligent Systems*, 2011.

30 M. Mernik, J. Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

31 M. Moon, K. Yeom, and H. Seok Chae. An approach to developing domain requirements as a core asset based on commonality and variability analysis in a product line. *IEEE Trans. Softw. Eng.*, 31:551–569, July 2005.

32 M. M. Moore and S. Rugaber. Domain analysis for transformational reuse. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 156–, Washington, DC, USA, 1997. IEEE Computer Society.

33 J.M. Neighbors. *Software construction using components*. PhD thesis, University of California, Irvine, 1980.

34 B. Omelayenko. Learning of ontologies for the web: the analysis of existent approaches. In *In Proceedings of the International Workshop on Web Dynamics*, 2001.

35 Aleksander P. Automatic ontology generation from web tabular structures. *AI Communications*, 19:2006, 2005.

36 H. Paulheim. Ontologies for user interface integration. In *Proceedings of the 8th International Semantic Web Conference*, ISWC '09, pages 973–981, Berlin, Heidelberg, 2009. Sp nger-Verlag.

37 J. Porubän and M. Bačíková. Definition of computer languages via user interfaces. In *FEEI 2010 : Electrical Engineering and Informatics : Proceeding of the FEEI of the TU of Koιsice. Available online: `http://hornad.fei.tuke.sk/~bacikova/publications/2010-08_feei10_kreutzova_CLANOK_final.pdf`*, pages 53–57. Technical University of Košice, September 2010.

38 J. Porubän, Forgáč M., M. Sabo, and M. Běhalek. Annotation based parser generator. *Computer Science and Information Systems : Special Issue on Advances in Languages, Related Technologies and Applications*, 2010.

39 J. R. G. Pulido, S. B. F. Flores, R. C. M. Ramirez, and R. A. Diaz. Eliciting ontology components from semantic specific-domain maps: Towards the next generation web. In *Proceedings of the 2009 Latin American Web Congress (la-web 2009)*, LA-WEB '09, pages 224–229, Washington, DC, USA, 2009. IEEE Computer Society.

40 T. T. Quan, S. Ch. Hui, A. Ch. M. Fong, and T. H. Cao. Automatic generation of ontology for scholarly semantic web. In *International Semantic Web Conference'04*, pages 726–740, 2004.

41 Christopher Ramming, Mm. Calton, Pu Examinateurs, Renaud Marlet, and Charles Consel. Domain-specific languages: Conception, implementation and application, phd. thesis.

42 J. Seidenberg and A. Rector. Web ontology segmentation: analysis, classification and use. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 13–22, New York, NY, USA, 2006. ACM.

43 J. Shim and H. Lee. Automatic ontology generation using extended search keywords. In *Proceedings of the 2008 4th International Conference on Next Generation Web Services Practices*, NWESP '08, pages 97–100, Washington, DC, USA, 2008. IEEE Computer Society.

44 S. Sie and J. Yeh. Automatic ontology generation using schema information. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence*, WI '06, pages 526–531, Washington, DC, USA, 2006. IEEE Computer Society.

45 M. Simos and J. Anthony. Weaving the model web: a multi-modeling approach to concepts and features in domain engineering. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 94–102, 1998.

46 N. Souchon and J. Vanderdonckt. A review of xml-compliant user interface description languages. pages 377–391. Springer-Verlag, 2003.

47 R. N. Taylor, W. Tracz, and L. Coglianese. Software development using domain-specific software architectures: Cdrl a011a curriculum module in the sei style. *SIGSOFT Softw. Eng. Notes*, 20(5):27–38, December 1995.

48 S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: From design to omplementation application to video device drivers generation. conception, implementation and application. *IEEE Transactions on Software Engineering*, 25(3):363–377, 1999.

49 Q. T. Tho, S. Ch. Hui, A. C. M. Fong, and T. H. Cao. Automatic fuzzy ontology generation for semantic web. *IEEE Trans. on Knowl. and Data Eng.*, 18(6):842–856, June 2006.

50 T. Thum, Ch. Kastner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Software Product Line Conference (SPLC), 2011 15th International*, pages 191–200. IEEE, August 2011.

51 Y. A. Tijerino, D. W. Embley, D. W. Lonsdale, Y. Ding, and G. Nagy. Towards ontology generation from tables. *World Wide Web*, 8(3):261–285, September 2005.

52 K. Tilly and Z. Porkoláb. Semantic user interfaces. *IJEIS*, 6(1):29–43, 2010.

53 T. Tudorache, N. F. Noy, S. M. Falconer, and M. A. Musen. A knowledge base driven user interface for collaborative ontology development. In *Proceedings of the 16th international conference on Intelligent user interfaces*, IUI '11, pages 411–414, New York, NY, USA, 2011. ACM.

54 A. Valerio, G. Succi, and M. Fenaroli. Domain analysis and framework-based software development. *SIGAPP Appl. Comput. Rev.*, 5:4–15, September 1997.

55 O. Vasilecas, D. Kalibatiene, and G. Guizzardi. Towards a formal method for the transformation of ontology axioms to application domain rules. *Information Technology and Control*, 38(4):271–282, 2009.

56 M. Wimmer. A meta-framework for generating ontologies from legacy schemas. In *Proceedings of the 2009 20th International Workshop on Database and Expert Systems Application*, DEXA '09, pages 474–479, Washington, DC, USA, 2009. IEEE Computer Society.

57 T. Wong, W. Lam, and E. Chen. Automatic domain ontology generation from web sites. *J. Integr. Des. Process Sci.*, 9(3):29–38, July 2005.

58 H. Yang and J. Callan. Ontology generation for large email collections. In *Proceedings of the 2008 international conference on Digital government research*, dg.o '08, pages 254–261. Digital Government Society of North America, 2008.

ABC with a UNIX Flavor

Bruno M. Azevedo and José João Almeida

Departmento de Informática, Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal
`pg19819@alunos.uminho.pt, jj@di.uminho.pt`

Abstract

ABC is a simple, yet powerful, textual musical notation. This paper presents ABC::DT, a rule-based domain-specific language (Perl embedded), designed to simplify the creation of ABC processing tools. Inspired by the Unix philosophy, those tools intend to be simple and compositional in a Unix filters' way. From ABC::DT's rules we obtain an ABC processing tool whose main algorithm follows a traditional compiler architecture, thus consisting of three stages: 1) ABC parser (based on `abcm2ps`' parser), 2) ABC semantic transformation (associated with ABC attributes), 3) output generation (either a user defined or system provided ABC generator).

1998 ACM Subject Classification H.5.5 Sound and Music Computing, D.3.4 Processors - Compilers

Keywords and phrases Music Processing, ABC Notation, Unix, Scripting, Compilers

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.203

1 Introduction

As computers were introduced to the world of music, a variety of file formats and textual notations emerged in order to describe music, such as, ABC [23], LilyPond [20] or MusicXML [17].

ABC is used as the base notation throughout all of this paper. Listing 1 illustrates an example of ABC notation and figure 1 its corresponding score.

Listing 1 Verbum caro factum est: Section 1; Part 1 - Soprano.

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 clef=treble-1 name="Soprano" sname="S."
G4 G2 | G4 F2 |A4 A2 | B4 z2|: B3 A GF| E2 D2 EF| G4 F2 | G6 !fine!:|
w: Ver- bum|ca- ro|fac- tum|est|Por - que *|to - dos *|hos sal -|veis
```

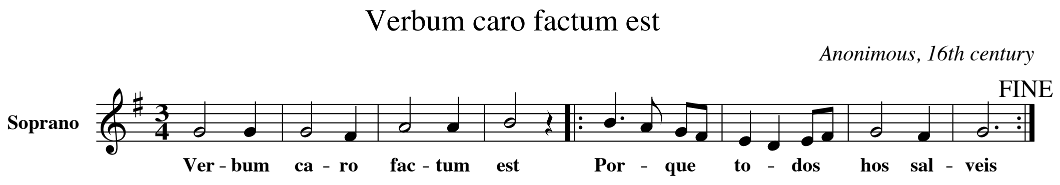

Verbum caro factum est
Anonimous, 16th century
FINE
Soprano
Ver - bum ca - ro fac - tum est Por - que to - dos hos sal - veis

Figure 1 Verbum caro factum est Score: Sections 1: Part 1 - Soprano.

There are many ABC processing tools and, among them, the most popular are the `abcm2ps` [18] typesetter and the MIDI creator `abc2midi` [1]. The first translates music written in ABC into customary sheet music scores in PostScript or SVG format. The latter converts an ABC file into a MIDI file.

UNIX Metaphor

The Unix philosophy [21] emphasizes the creation of simple, yet capable and efficient programs, which tackle only one problem at a time. Moreover, programs should handle text streams as a universal type. The latter allows programs to easily communicate with each other.

In order to facilitate the development of new Unix commands, Unix creators built a new language (C).

Unix is simple. It just takes a genius to understand its simplicity. (Dennis Ritchie)

When we move to the music world we also believe in building simple music commands, using a universal music stream type (ABC), creating a music command development language and exercising music command compositionality.

This paper describes a system for creating ABC processing tools with the following design goals:

- Generation of simple tools through a compact specification;
- ABC oriented;
- Being able to deal with real ABC music (more than a sequence of notes);
- Being able to associate transformations with specific ABC elements, allowing a surgical processing;
- Rich embedding mechanisms (using Perl for specific ABC transformations).

In short, we present a rule-based domain-specific language [16, 15] (DSL) - ABC::DT - for building simple, compositional (in a Unix filters' meaning) ABC processing tools.

This document is organized as follows: in section 2, we describe related music notations, tools and projects, and summarize the most relevant music representation approaches; in section 3, we discuss ABC::DT's rules and the algorithm of the generated ABC processing tool; finally, in section 4, we present some tools created with ABC::DT.

2 State of the Art

In this section we will describe the music notation standard ABC, present the most relevant ABCtools and projects and summarize the most popular music representation approaches.

2.1 ABC

Most music notation programs have a visual approach, in which the user drags and drops notes and symbols using the mouse and the resulting sheet is displayed on the screen. An alternative approach is writing music using a text-based notation. This is a non-visual mode that represents notes and other symbols using text characters, making it economic and sometimes intuitive to use and also making possible faster transcriptions. A specialized program then translates the notation into printable sheet music in some electronic format (e.g. PDF) and/or into a MIDI file.

Many text-based notations have been created [20, 17], and one of them was ABC, introduced by Chris Walshaw in 1991 as a means to share traditional folk music, such as Irish jigs.

ABC is a musical notation standard and not a software package. ABC was later expanded to provide multiple voices (polyphony), page layout details, and MIDI commands.

An ABC tune has a header with fields for title (T), composer (C), key signature (K), time signature or meter (M) and default note duration or length (L). The music is notated using the letters A (lá) to G (sol) to represent the notes. The notation has a simple and clean syntax, and is powerful enough to produce professional and complete music scores. Among other advantages, the following are the most important:

- powerful enough to describe most music scores available in paper;
- actively maintained and developed;
- the source files are plain text files;
- this format can be easily converted to other known formats;
- there are already tools for transforming and publishing ABC, such as, `abcm2ps` [18] and `abc2midi` [1];
- compact and clear notation;
- human readable;
- thousands of tunes available on the Internet;

ABC was adopted in this work in order to cope with real world problems that occurred in the project WikiScore [2].

2.2 Projects and Tools

In this subsection we discuss some the most relevant projects and tools being developed or used at the moment[1].

abcm2ps [18] A command line program which translates music written in ABC music notation into customary sheet music scores in PostScript or SVG format.
It is based on `abc2ps` 1.2.5 and was developed mainly to print Baroque organ scores that have independent voices played on multiple keyboards and a pedal-board. The program has since then been extended to support various other notation conventions in use for sheet music. Moreover, it is now one of the most complete ABC implementations.
It is developed in C language and the author, an organist and programmer called Jean-François Moine, releases "stable" and "development" versions of his program. As of this writing[2], the stable release is 6.6.22 and the development release is 7.5.2. Since release 7.2.1, `abcm2ps` tries to follow the ABC standard version 2.1.

abc2midi [1] A program that converts an ABC music notation file into a MIDI file.
It is part of the abcMIDI package, which includes other utility applications. The program was developed in C language by James Allwright in the early 1990s and has been supported by Seymour Shlien since 2003. The program contains many features, such as expansion of guitar chords, drum accompaniment, and support for micro tones which do not exist in other packages.

Music21 [8] A Python-based toolkit for computer-aided musicology.
Music21 is a set of tools for helping scholars and other active listeners answer questions about music quickly and simply.
Music21 builds on preexisting frameworks and technologies such as Humdrum, MusicXML, MuseData, MIDI , and Lilypond, but Music21 uses an object-oriented skeleton that makes

[1] A more extensive list of ABC software may be consulted in `http://abcnotation.com/software#linux`
[2] 20th May, 2013.

it easier to handle complex data. At the same time, Music21 tries to keep its code clear and make reusing existing code simple.

Applications of this toolkit include computational musicology, music informations, musical example extraction and generation, music notation editing and scripting, and a wide variety of approaches to composition, both algorithmic and directly specified.

It also has a large corpus of musical scores in many formats, including ABC and MusicXML.

abctool [14] A python script that manipulates music files in ABC format.

It's mostly useful for people working on the command line and/or editing ABC directly in an editor. It relies on external programs for certain tasks like converting into PostScript or transposing.

Its main features are reading from standard input or file, outputting to standard output (PostScript, PDF or MIDI), view (using `abcm2ps` and `gv`), transposition, translation of chord names to Danish/German, and removal of chords and fingerings.

It is open source, developed by Atte André Jensen and released under GPL.

Haskore [13] Haskore is a set of Haskell modules for creating, analyzing and manipulating music. Music can be made audible through various back-ends.

The formal approach used in this project is very elegant and powerful and is a very good studying resource. Nevertheless, when we want to process existing ABC music, we have many details that don't fit in Haskore model like slurs, dynamics, microtones. In order to process them, we have to forget those elements or introduce drastic changes to the model.

All the tools and projects presented were very relevant: `abctool` is simple command following Unix's philosophy; `abc2midi` and `abcm2ps` deal with processing real world ABCs, but for specific purpose; `Music21` has similar goals and has a very powerful and complex object oriented modules for music processing; `Haskore` is very flexible and elegant but can't deal with real world ABC details.

2.3 Internal Representation

The internal representation of musical information is an area of research that has been receiving a lot of contributions throughout time and there will never be a consensus about the structure it should have. One of the most influent matters in making such representations is its final purpose. There are different purposes like rendering of music, play back, printing, music analysis, composition, among others.

The scope of this work includes only music rendering and analysis, therefore the representation will have a well defined orientation, and a set of music properties will automatically be discarded. There are many models, data structures, paradigms, techniques, systems and theories proposed by many authors [6, 7, 5, 22, 24, 10] and none can be labeled as a "true" representation, as there will never be a closed definition of music and it is still difficult to represent all aspects of music.

As will be explained in section 3, this work presents a Perl module called ABC::DT, which can be viewed as a DSL embedded in Perl. It processes some input information and returns it. The input information is parsed and an internal representation is generated. That representation guides how the processing will be done. So, it is important to establish its structure, as it will determine how an ABC tune may be processed.

The most used representations will be shortly discussed next.

2.3.1 Structure

In the beginning, computer music systems represented music as a simple sequence of notes. It was a simple approach, thus making it difficult to encode structural relationships between notes, such as enveloping a group of notes in order to apply some kind of property.

It is widely accepted that music is best described at higher levels in terms of some sort of hierarchical structure [3]. This kind of structure has the benefit of isolating different components of the score, therefore allowing transformations, such as tempo or pitch, to be applied to each of them individually. It also represents a set of instructions for how to put the score back together, hence allowing to reassemble it as it was.

Musical events can spread behavior to other events through the binary relation *part-of*, which denotes relations like "measures part-of phrase." They can also inherit behavior and characteristics from other events through the *is-a* relation, which designates relations like "a dominant chord being a special kind of seventh chord" [12].

A single hierarchy scheme is not enough because music frequently contains multiple hierarchies, for instance, a sequence of notes can belong simultaneously to a phrase marking and a section (like a movement). So the need of a multilevel hierarchy appears. There are some other possible hierarchies: voices, sections (movement, section, measure), phrases, and chords, all of which are ways of grouping and structuring music.

A few representations have been proposed [9, 6] that support multiple hierarchies through named links relating musical events and through instances of hierarchies. And others where tags are assigned to events in order to designate grouping, such as, all notes under a slur.

2.3.2 Melodic and Harmonic Structures

In polyphonic music there are materials besides melody that are combined in a score: rhythm and harmony. Those three (melody, rhythm and harmony) determine the global quality of a score [4] and their combination is usually called a texture. When there's only one voice (melody) accompanied or not by chords, it is called monophony, but when there's two or more independent voices, it is called polyphony.

The study of independent melodies is relatively simple compared to the analysis of polyphony. Each voice moving through the horizontal dimension creates other effects by overlapping with notes in other voices. The necessity for representing these vertical structures arises so that the harmonic motion can be analyzed.

The variability of the score's texture originates an issue. A score may have different densities of notes per part and it is required that all events occurring at the same time are vertically aligned. So, Brinkman [6] suggests a solution that uses a linked representation of a sparse matrix. Each row of the latter references a part and each column the onset of the elapsed time, which would enable traversing the score in any direction required (vertical or horizontal). Thus, attaining a perception of the context of what's happening in a specific part, a feature that can't be achieved when dealing with representations with only one dimension. Moreover, it makes the task of score segmentation by part or time easier.

2.3.3 Summary

The internal representation's types of structures were discussed and it revealed that there are mainly two types that are most commonly approached by researchers: sequential and hierarchical. However, the decision of which structure type one should choose relies on the purpose the internal representation will have: rendering of music, printing, music analysis, composition, etc.

Regarding the horizontal and vertical dimensions of polyphonic music, a solution to enable the harmonic analysis of a score was suggested. A perfect representation would be one that was sufficiently general and complete to be useful in many different analytic tasks in many styles of music [12], like expressing common abstract musical patterns.

An assessment is taken after a, not so thorough, research on representations and their pros and cons: sequential and hierarchical structures are more suited to horizontal readings and tasks, such as re-rendering an ABC tune, since they preserve the original order of the elements on an ABC tune. Whereas, structures like sparse matrices grant both horizontal and vertical readings. Such structures provide a representation more suited to purposes requiring a way of accessing vertical events on a score. Yet, it does not maintain the original order of elements. For instance, in ABC, it is common to write a part alternately with other parts like (voice A, voice B, voice A, voice B). Meaning that a fragment of part A is written first, followed by that of part B, voice A and voice B again. When representing a score oriented to a vertical axis, the order of events is lost, thus invalidating tasks like re-rendering ABC tunes.

3 From ABC::DT to an ABC Processing Tool

A typical ABC processing tool follows a traditional compiler's structure:

1. Parse ABC input;
2. Transform the generated representation;
3. Generate the output;

In the first stage, the ABC parser generates an intermediate representation (IR) to be transformed in the following stage. This parser is independent of the intended transformation and is constant. In the second stage, the IR is transformed according to the ABC::DT rules. Each rule is composed by the pair *actuator* ⇒ *transformation*, where the actuator describes the IR's part to be transformed. Finally, in the third stage, an output of the transformed IR is generated.

Figure 2 illustrates the ABC processing tool architecture.

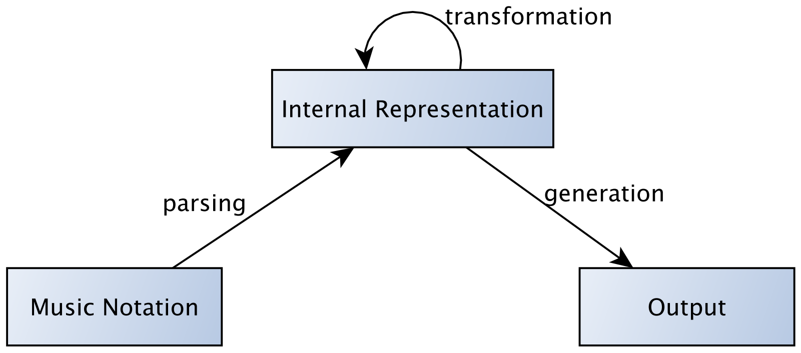


Figure 2 ABC processing tool architecture.

In order to generate a specific tool, we only need to build the stages - 2) and 3) - that depend on the rules.

3.1 Parse ABC Input

As previously stated in the introduction, we want to be able to deal with real ABC music. The ABC parser has to be robust, i.e., to be able to expect cases that it doesn't recognize.

The main options for building the parser were: to build it from scratch; to reuse an existing parser from robust programs like `abcm2ps` or `abc2midi` and adapt it to the requirements; or to use directly one of the aforementioned programs' parsers.

Since building a robust parser is very time consuming, the first solution was discarded. The second option would raise problems when adapting our parser to newer versions. So, `abcm2ps`' parser was the natural choice.

3.1.1 `abcm2ps` Parser's Features

`abcm2ps` is one of the most widely used programs for working with ABC, not just as a standalone software but as part of many applications. This fact implies that it's not a piece of software that was casually made. It was designed to process ABC in the best way possible, therefore its quality is acknowledged.

It is actively maintained and well documented which facilitates the analysis of the structures it generates. Moreover, its author, Jeff Moine, was and still is a preponderant influence for the evolution of the ABC notation and standard.

The IR generated by its parser follows the sequential structure type, in other words, each element captured by the parser is simply appended to an ordered list. An element is any component existing in ABC, from the header information - like the key or initial meter - to a note, bar or a tuplet. The processor that will go through the structure has to keep record of the context of each element. The context comprises components like the voice, the meter, the length, the key, among others.

Given that `abcm2ps` was designed to print ABC, its IR is not suited for music analysis or composition purposes. Therefore, it lacks all the benefits inherent to an hierarchical representation, such as, inheriting behavior and characteristics between musical events.

Still, it can be easily organized as a set of monophonic voices. This set might, for instance, be used as a starting point to describe relationships between vertical musical entities on a polyphonic score.

As the aim of this work is to build a toolkit based on scripts, the sequential structure reveals itself as an appropriate structure that meets our requirements.

The sequence of elements that the structure provides can be easily mapped to an array or a hash. These data types are part of the common, yet powerful, data types of a scripting language like Perl, which is exactly what we want.

3.1.2 From `abcm2ps` Parser's IR to Perl

`abcm2ps`'s parser is implemented in C, so the structure that it generates (a list of C data structures) has to be adapted to Perl. This adaptation is done by a Perl serialization[3] process of the original C structure. Hence, we've created a program - called *C2Perl* - that parses an ABC file, transforms the generated C structure and prints the serialized Perl output.

In short, *Parse* ABC *Input* stage is comprised of a Perl serialization of `abcm2ps`'s parser generated structure followed by a Perl evaluation of the serialized structure into a Perl *hash*.

[3] Serialization is the process of translating data structures into a format that can be stored and resurrected later in the same or another computer environment.

This way, we obtain a Perl structure that maps the original C structure.

Figure 3 depicts the internal workflow of the *Parse* ABC *Input* stage. *C2Perl*'s workflow is represented by the group node '*C2Perl*'.

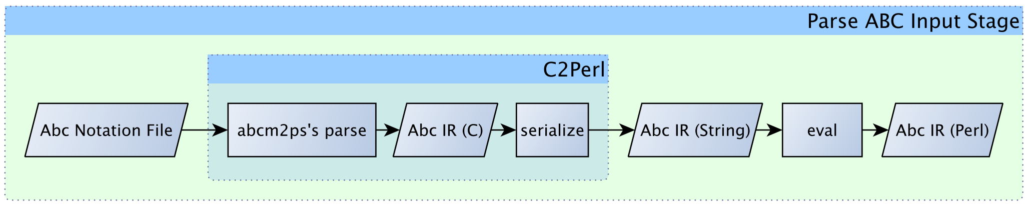


Figure 3 *Parse* ABC *Input* stage.

We are merely mapping the original C structure to a Perl one, thus keeping the original order and meaning. However, it could be possible to reorganize the structure in order to serve other purposes. For example, organize it as being oriented to the part, meaning that we could access directly to a specific part. Or organize it by elapsed time, meaning that it would be possible to retrieve all events that occurred in a specific moment in time.

3.2 Transform the Generated Representation

This stage's process - *abc_processor* - makes an IR traversal applying the ABC::DT rules to each element.

The generic processing strategy is to provide a set of transformations for very specific points (defined by actuators) and through that obtain the general tools. Any point not covered by the rule's actuators is kept unchanged, following the default transformation which is the identity function.

This kind of strategy is effective in building tools that do simple transformations - we only need to provide what is to be changed.

3.2.1 ABC::DT Rules

ABC::DT rules - `handler` - are defined by a correspondence between an actuator and a tranformation. An actuator selects a specific element - like a note - or a set of elements - like all elements that are defined in a particular context/state. Each actuator is translated into an expression that matches different element attributes, in order to accurately select it.

The actuators enable the existence of different levels of detail that guide the search for the required element. Therefore, the actuator has a natural notation, in which, more generic elements are written before more specific ones. The elements within an actuator are separated by the characters '`::`'.

For example, '*in_line::K:*' selects all key elements (K) with state 'in_line', that is, a key which is defined after the header and is surrounded by square brackets - `[K:G]`. Another example, '*note::!f!*' selects all note elements which have the decoration `!f!` associated - `!f!G`.

Due to the existence of different levels of detail, when there is more than one actuator that matches an element, the most specific is the one applied to the element.

In ABC::DT rules, the user can define a special actuator called *-default* to describe how to transform each uncovered element. Optionally, a *-end* actuator can be defined, which enables a general post processing of the final ABC, hence, making possible to attain different output formats.

3.2.2 Processor Algorithm

abc_processor is guided by the IR's structure, meaning that each element is processed sequentially. It admits a table of rules, called `handler` - a dispatch table[4] - in which an actuator is associated with a tranformation. During the tune's processing, when a element matches an actuator, the corresponding transformation is applied. An actuator selects a specific element or a group of elements. A transformation is specified by the user and it defines how each element should be processed according to its internal values.

This implementation's main features are:

Dispatch Table ABC::DT rules are defined by a correspondence between the actuators and tranformations, called `handler`.

Higher-Order Processing The transformations are user specified functions.

Systematic In order to build a tool, a user must define what and how is to be transformed.

Specify only the necessary If no actuator applies, the identity function is used.

Rich Actuators The set of actuators is comprised of well structured elements in order to provide a precise processing.

Processing strategies There is a table of strategies. Each strategy defines how a transformation's output is to be merged with others.

During the traversal, *abc_processor* calculates the current element's state, including the voice id, the time elapsed per voice. That state grants a more complete control of what can be processed and provides a richer semantic processing.

The processor's algorithm was inspired in the one used in XML::DT [11], a processing module of XML documents.

3.3 Generate the Output

In this stage, the transformed representation is outputted and it may be of the same type as the input, which enables the composition of other tools.

The identity function, which is called *toabc*, prints the contents of an element just as they were in the ABC source tune. This function's implementation was based on the one used by `tclabc` [19] which also uses `abcm2ps`' parser.

4 ABC::DT by Example

In this section we will present examples of tools created using ABC::DT, thus demonstrating how easily a (simple) tool or some occasional processing can be made.

4.1 All But One

When there is a multi-voice score, like a four part choir, it is important to, for instance, the Soprano to hear all the other parts except hers. That way, she can study her part knowing what the rest is going to sound.

[4] A dispatch table is a table of pointers to functions or methods.

The *All-but-one* tool generates an ABC score whose goal is to help musicians in individual rehearsal of multi-voice music for studying purposes.

In ABC, it is possible to add commands to control audio properties. `abc2midi` recognizes MIDI directives – `%%MIDI`, followed by different parameters. For *All-but-one*, we need to add a MIDI directive to reduce the volume of the voice. To be more precise, it has to generate a change-volume MIDI directive – `%%MIDI control 7 NewVolume`, where NewVolume is a number between (0-127) – after the voice definition for it to be silenced.

This command line program, may receive command line options:

-v, –voice A string is expected identifying either the voice's id - a number generated by the parser - or name that will be attenuated.

-m, –min-volume A number is expected defining the volume's value for the voice to be attenuated.

The ABC::DT specification is quite simple. There is only one rule: selecting the voice to be attenuated, and add a change-volume command, as illustrated in listing 2.

Listing 2 Handler.

```
my %handler = (
  "V:$requested_voice" => sub {
    toabc() . "%%MIDI control 7 $min_volume\n";
  }
);
```

The variable `$requested_voice` is defined by the command line option. So, if it is "Tenor," the actuator is `V:Tenor`, and it will search for the element *voice* whose name is "Tenor."

The output of *All-but-one* is an ABC tune so it can be chained with other ABC tools.

Listing 3 shows how this tool could be used. It reads the tune 100.abc (listing 4) and the output is shown in listing 5.

Listing 3 ABC All But One.

```
abc_all_but_one -v=Tenor -m=25 100.abc
```

Listing 4 File `100.abc`.

```

X:100
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
G4 G2|G4 F2|A4 A2|B4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
V:2 name="Contralto" clef=treble
D4 D2|E4 D2|E4 F2|G4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
V:3 name="Tenor" clef=treble-8
G3 A B2|c4 A2|c4 c2|d4 z2|:
w: Ver - bum|ca- ro|fac- tum|est|
V:4 name="Baixo" clef=bass
G,4 G,2|C,4 D,2|A,4 A,2|G,4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
```

Listing 5 File `100_all_but_tenor.abc`.

```
X:100
T:Tuti
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" clef=treble
G4 G2|G4 F2|A4 A2|B4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
V:2 name="Contralto" clef=treble
D4 D2|E4 D2|E4 F2|G4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
V:3 name="Tenor" clef=treble-8
%%MIDI control 7 25
G3 A B2|c4 A2|c4 c2|d4 z2|:
w: Ver - bum|ca- ro|fac- tum|est|
V:4 name="Baixo" clef=bass
G,4 G,2|C,4 D,2|A,4 A,2|G,4 z2|:
w: Ver- bum|ca- ro|fac- tum|est|
```

Note the MIDI command `%%MIDI control 7 25` after the voice definition `V:3 name="Tenor" sname="T." clef=treble-8`. This way, the voice "Tenor" is going to be attenuated when `abc2midi` reproduces the score.

4.2 ABC Paste

This tool, as the Unix Paste, merges the voices of tunes parallel to each other in the time perspective. In other words, each voice starts at the beginning of the resulting tune.

Some decisions were made regarding what should be done with some information present in a tune. This ensured that the resulting tune was consistent with each individual tune:

1. The context at each point in the tune is recorded. The context comprises the current voice, the key, the meter, the length, the tempo and the number of measures for each voice.
2. Any context change like the key or the meter is written only if it differs from the current.
3. The resulting tune's header is the one present in the first tune which has an actual tune written, in other words, at least one note.
4. In the resulting tune, any voice that has fewer measures than the longest one is appended with measure rests.

The tool's algorithm is divided in three stages: 1) retrieving the header for the resulting tune, 2) pasting the tunes and 3) appending any necessary rests.

1. As mentioned before, the resulting tune's header is the one from the first tune with at least on note written. This follows a simple algorithm where each tune is searched in the order they are passed in. As soon as it finds a tune with a note written it stops, following to the next step. The `handler` to be passed to the processor needs only three entries, each corresponding to a tune's state that the `abcm2ps`' parser generates.
2. Pasting is the most complicated part, yet in the end it was not that difficult to implement. The algorithm consists on running the processor for each tune and concatenating each result. The handler has some entries like the one in listing 6, where everytime the element corresponding to the measure bar is visited, a counter for the current voice's written measures is incremented. The identity function *toabc* is called so that the actual bar can be outputted.

Listing 6 Counting measure bars.

```
my %handler = (
   'bar' => sub {
     $tune_info{$c_voice_id}{measures}++;
     toabc();
   },
   ...
 );
```

   Other entries update the current voice variable when a *voice* is found or the current key when a *key* is found. Through the context variables, which are constantly updated, it is possible to compare the current context and the new. This enables the possibility of not printing the context declaration if it is the same as the current, thus making the resulting tune cleaner without useless duplications.
3. Final step happens after step 2) and it consists on verifying if there is any voice with fewer measures than the voice with the biggest number of measures. If there is such a voice then a multiple measure rest with the difference is appended to that voice. This is possible because, in step 2), the number of measures for each voice was being recorded.

In the end the output generated is printed to the output. Since it is still an ABC tune it can be the input of other tools like this one or ABC Cat that will be described next. Listing 7 shows how this tool could be used. It reads tunes 101.abc (listing 1) and 103.abc (listing 8) and the output is shown in listing 9 with its respective score (figure 7, area A).

Listing 7 ABC Paste by example.

```
abc_paste 101.abc 103.abc
```

Listing 8 Verbum caro factum est: Section 1; Part 3 - Tenor.

```
X:103
T:Verbum caro factum est
C:Anon, 16th century
M:3/4
L:1/8
K:G
V:3 clef=treble-8 name="Tenor" sname="T."
G3 A B2 | c4 A2 | c4 c2 | d4 z2 |:\
w: Ver - bum | ca- ro | fac- tum | est |
d2 B4 | c2 B4 | c2 A4 | G6 :|
w: Por- que | to- dos  | hos sal-|veis
```

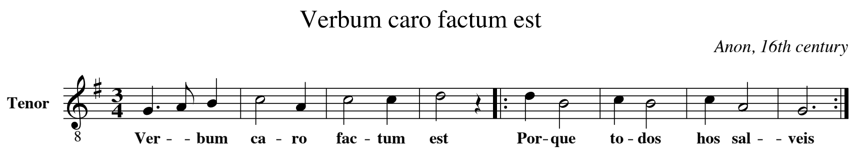

Verbum caro factum est
Anon, 16th century
Tenor
Ver - - bum ca - ro fac - tum est Por-que to - dos hos sal - - veis

Figure 4 Verbum caro factum est: Section 1; part 3 - Tenor.

Listing 9 Verbum caro factum est: Section 1; Part 1 & 3.

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" sname="S." clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|: \
w: Ver- bum | ca- ro | fac- tum | est |
B3 A GF| E2 D2 EF| G4 F2| G6!fine!:|
w: Por - que *| to - dos * | hos sal-|veis
V:3 name="Tenor" sname="T." clef=treble-8
G3 A B2| c4 A2| c4 c2| d4 z2|: \
w: Ver - bum | ca- ro | fac- tum | est |
d2 B4| c2 B4| c2 A4| G6:|
w: Por- que | to- dos  | hos sal-|veis
```

4.3 ABC Cat

This tool is based on Unix's cat, as it consists on the concatenation of each tune one after the other in the time perspective. In other words, any voice present in the second tune is always printed after any voice present or not in the first, and so on.

Some design goals were established:

1. The context at each point in the tune is recorded. The context comprises the current voice, the key, the meter, the length, the tempo. The number of measures for each voice is recorded separately for each tune.
2. Any context change like the key or the meter is written only if it differs from the current.
3. Each tune's header information regarding the tune's context is always written except if it is the same as the current one.
4. For each tune, before printing it, a verification for missing voices is made in the current tune and all prior to that. This way, measure rests can be appended in order to have a consistent resulting tune.
5. Any voice that has fewer measures than the longest one will be appended with measure rests.

The tool comprises only one step. Yet it is more complex than ABC Paste's. Its algorithm consists in traversing all tunes, running the processor for each tune and verifying if there are any measure rests to append to a voice. This is done by comparing voice's measures within the current tune and previous ones. The `handler` is very similar to the one used in ABC Paste.

In the end the output generated is printed to the output. Since it is still an ABC tune it can be the input of other tools.

Listing 10 shows how this tool could be used. It reads tunes 201.abc (listing 11) and 303.abc (listing 12) and the output is shown in listing 13 with its respective score (figure 7, area B).

Listing 10 ABC Cat by example.

```
abc_cat 201.abc 303.abc
```

Listing 11 Verbum caro factum est: Section 2; Part 1 - Soprano.

```
X:201
T:Solo Fem
C:Anon, 16th century
M:3/4
L:1/8
K:C
V:1 clef=treble name="Soprano" sname="S."
 [L:1/8] [M:3/4][K:G]
B4c2 | B2 A2 > G2 | G4 F2 | G4 G2 |
w: 1.~Y la | Vir-gen * | le de-| zi-a:
```

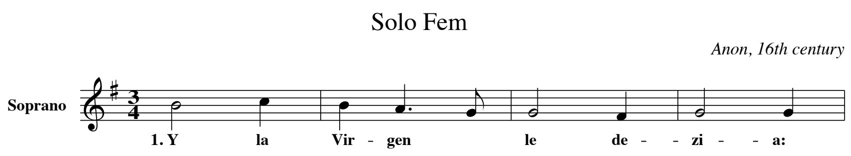


Figure 5 Verbum caro factum est: Section 2; Part 1 - Soprano.

Listing 12 Verbum caro factum est: Section 3; Part 3 - Tenor.

```
X:303
T:Solo Tenor
C:Anon, 16th century
M:3/4
L:1/8
K:G
V:3 clef=treble-8 name="Tenor" sname="T."
 [M:3/4] d4 e2| d2c2 > B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

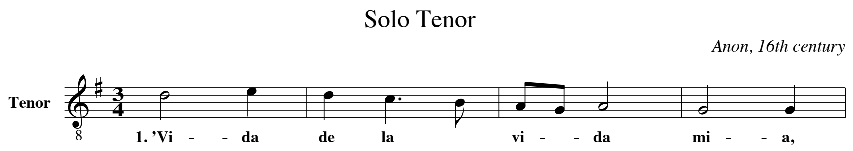


Figure 6 Verbum caro factum est: Section 3; Part 3 - Tenor.

Listing 13 Verbum caro factum est: Section 2: Part 1 & Section 3: Part 3.

```
X:201
T:Solo Fem
C:Anon, 16th century
M:3/4
L:1/8
K:C
V:1 name="Soprano" sname="S." clef=treble
 [K:G]
 B4c2| B2 A2> G2| G4 F2| G4 G2|
w: 1.~Y la | Vir-gen * | le de-| zi-a:
 [V:1] Z4 |
 [V:3] Z4 |
V:3 name="Tenor" sname="T." clef=treble-8
 d4 e2| d2c2> B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

4.4 Real Example

A real application for these tools could be their composition. Using the score *Verbum caro factum est* whose sections and parts are divided in separate files, it is possible to assemble the whole score by composing ABC Paste with ABC Cat. The score will be composed by the examples shown previously, so it will be comprised of three sections and only two parts (Soprano and Tenor). Listing 14 shows how the tools are composed and listing 15 shows the ABC for the composed score.

Listing 14 ABC Cat and Paste by example.

```
abc_cat (
   abc_paste ( 101.abc 103.abc )
   abc_cat   ( 201.abc 303.abc )
)
```

Listing 15 Verbum caro factum est: Sections 1, 2 & 3; Parts 1 & 3.

```
X:101
T:Verbum caro factum est
C:Anonimous, 16th century
M:3/4
L:1/8
K:G
V:1 name="Soprano" sname="S." clef=treble
G4 G2| G4 F2| A4 A2| B4 z2|: \
w: Ver- bum | ca- ro | fac- tum | est |
B3 A GF| E2 D2 EF| G4 F2| G6!fine!:|
w: Por - que *| to - dos * | hos sal-|veis
V:3 name="Tenor" sname="T." clef=treble-8
G3 A B2| c4 A2| c4 c2| d4 z2|: \
w: Ver - bum | ca- ro | fac- tum | est |
d2 B4| c2 B4| c2 A4| G6:|
w: Por- que | to- dos  | hos sal-|veis
V:1
B4c2| B2 A2> G2| G4 F2| G4 G2| \
w: 1.~Y la | Vir-gen * | le de-| zi-a:
[V:1] Z4|
[V:3] Z4|
V:3
d4 e2| d2c2> B2|AGA4| G4 G2|
w: 1.~'Vi-da | de la * | vi - da | mi-a,
```

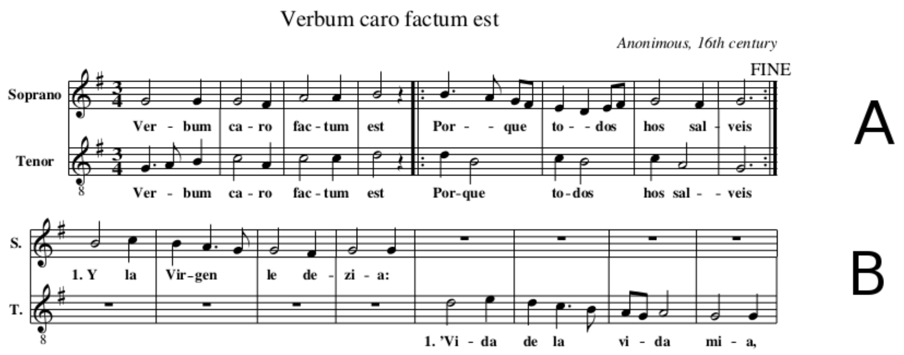

Verbum caro factum est
Anonimous, 16th century
FINE
Soprano
Ver - bum ca - ro fac - tum est Por - - que to - - dos hos sal - veis
Tenor
Ver - bum ca - ro fac - tum est Por-que to-dos hos sal - - veis
A
S.
1. Y la Vir-gen le de - zi - a:
T.
1. 'Vi - da de la vi - da mi - a,
B

Figure 7 Verbum caro factum est Score: Sections 1, 2 & 3; Parts 1 & 3.

5 Conclusions

Inspired in a solution that revealed successful - the creation of the language C to help developing Unix - a DSL, called ABC::DT, was created in this work as well.

Reusing `abcm2ps`'s parser was very important to help guarantee this work's quality, coverage and developing time. The generated IR is source oriented which allows obtaining valid ABC and robust tools - it knows how to deal with unknown elements.

The representation used must be complete enough to enable the application of many different analytic tasks. However, that fact doesn't invalidate an approach that starts by generating a sequential structure and from it generating something more suited to more complex uses.

Using Perl as the language embedded into ABC::DT provides a rich environment to allow easy processing of text. Furthermore, through the use of data structures, like hashes, the user has bigger expressive power to specify transformations.

We believe that the rule based processor makes it possible to write very compact tools.

One of our main goals is to build an ABC operating system. Moreover, presently, there is a lack of music notation general processing tools, particularly for ABC. Thus, the existence of DSL's like ABC::DT helps to the simplification of crafting new ABC processing tools.

References

1 James Allwright and Seymour Shlien. abc2midi. `http://abc.sourceforge.net/abcMIDI/`. Tool.

2 J.J. Almeida, N.R. Carvalho, and J.N. Oliveira. Wiki::score - a collaborative environment for music transcription and publishing. 2012. `http://wiki-score.org/`.

3 M. Balaban. *A Music Workstation Based on Multiple Hierarchical Views of Music.* State University of New York at Albany, Department of Computer Science, 1987.

4 B. Benward and M. Saker. *Music: In Theory and Practice.* McGraw-Hill, 2003.

5 Jeff Bilmes. A model for musical rhythm. In *Proceedings of the International Computer Music Conference*, pages 207–207. International Computer Music Association, 1992.

6 A Brinkman. A Data Structure for Computer Analysis of Musical Scores. *Proceedings of the ICMC*, 1984.

7 William Buxton, William Reeves, Ronald Baecker, and Leslie Mezei. The Use of Hierarchy and Instance in a Data Structure for Computer Music. *Computer Music Journal*, 1978.

8 Michael Scott Cuthbert and Ben Houge. Music21. `http://web.mit.edu/music21/`. Toolkit.

9 Roger B. Dannenberg. A structure for efficient update, incremental redisplay and undo in graphical editors. *Software: Practice and Experience*, 1990.

10 Roger B Dannenberg. A Brief Survey of Music Representation Issues, Techniques, and Systems. *Computer Music Journal*, 1993.

11 José João Dias de Almeida. *Dicionários dinâmicos multi-fonte.* Tese de doutoramento, Universidade do Minho, 2003.

12 Henkjan Honing. Issues on the representation of time and structure in music. *Contemporary Music Review*, 1993.

13 Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation–an algebra of music–. *Journal of Functional Programming*, 1996.

14 Atte André Jensen. abctool. `http://atte.dk/abctool/`. Tool.

15 Tomaž Kosar, Pablo A Barrientos, Marjan Mernik, et al. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 2008.

16 Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Varanda João Maria Pereira, Matej Črepinšek, Cruz Daniela Da, and Rangel Pedro Henriques. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 2010.

17 Recordare LLC. Musicxml. `http://www.makemusic.com/musicxml`. Musical Notation.

18 Jean-François Moine. abcm2ps. `http://moinejf.free.fr/`. Tool.

19 Jean-François Moine. tclabc. `http://moinejf.free.fr/`. Tool.

20 Han-Wen Nienhuys and Jan Nieuwenhuizen. Lilypond. `http://lilypond.org/`. Musical Notation.

21 E.S. Raymond. *The art of Unix programming.* Addison-Wesley Professional, 2004.

22 A Smaill, G Wiggins, and M Harris. Hierarchical music representation for composition and analysis. *Computers and the Humanities*, 1993.

23 Chris Walshaw. Abc notation. `http://abcnotation.com/`. Musical Notation.

24 Geraint Wiggins, Mitch Harris, and Alan Smaill. Representing music for analysis and composition. In M Balaban, K Ebcio Vglu, O Laske, C Lischka, and L Soriso, editors, *Proceedings of the Second Workshop on AI and Music.* Dept. of Artificial Intelligence, Edinburgh, Association for the Advancement of Artificial Intelligence, 1989.

Specifying Adaptations through a DSL with an Application to Mobile Robot Navigation

André C. Santos[1,3], João M. P. Cardoso[2], Pedro C. Diniz[3], and Diogo R. Ferreira[1]

1 **IST – Technical University of Lisbon, Portugal**
{acoelhosantos, diogo.ferreira}@ist.utl.pt
2 **Faculty of Engineering, University of Porto, Portugal**
jmpc@acm.org
3 **INESC-ID, Lisbon, Portugal**
pedro@esda.inesc-id.pt

Abstract

Developing applications for resource-constrained embedded systems is a challenging task specially when applications must adapt to changes in their operating conditions or environment. To ensure an appropriate response at all times, it is highly desirable to develop applications that can dynamically adapt their behavior at run-time. In this paper we introduce an architecture that allows the specification of adaptable behavior through an external, high-level and platform-independent domain-specific language (DSL). The DSL is used here to define adaptation rules that change the run-time behavior of the application depending on various operational factors, such as time constraints. We illustrate the use of the DSL in an application to mobile robot navigation using smartphones, where experimental results highlight the benefits of specifying the adaptable behavior in a flexible and external way to the main application logic.

1998 ACM Subject Classification D.2.8 – Software Engineering – Software Architectures – data abstraction, domain-specific architectures, languages; D.3.2 – Programming Languages – Language Constructs and Features – frameworks; D.3.2 – Programming Languages – Language Classifications – specialized application languages

Keywords and phrases Domain-specific language, run-time adaptations, adaptive behavior, embedded systems, mobile robot navigation.

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.219

1 Introduction

The continued miniaturization of computing devices has contributed to making embedded systems pervasive in a wide range of diverse contexts and thus with a wide variety of computational requirements (see e.g., [2]). Regardless of the device, be it mobile phones, vehicle equipments, medical instruments, or smart home components, all of these systems embody very stringent requirements in terms of reliability, maintainability, availability, safety, security, efficiency, energy consumption, among others. Overall, the diversity of embedded systems and requirements pose tremendous challenges to the development and maintainability of their software applications. In particular, this software must operate within acceptable performance parameters in resource-constrained environments while being subject to changing operating conditions (e.g., temporary unavailability of sensors, decreasing battery level, real-time requirements, memory limitations, intermittent connectivity).

Run-time adaptability is seen as a viable strategy to cope with these challenges (e.g., [14]). For example, the software implementation could leverage the use of different but equivalent processing algorithms, changing algorithm parameters, switching sensors, or simply changing the frequency of some computations (see, e.g., [3, 19]). However, implementing dynamic behavior in embedded applications involves a considerable amount of effort, as the inclusion of such behavior in the application is complicated and error-prone due to the high degree of intertwining between application and adaptation code [13]. This additional programming effort usually requires a mixture of conditional coding and low-level operations, which translates into reduced code readability and more difficult maintenance. Furthermore, such efforts typically scale poorly when multiple adaptations are used, making continuous development harder [13]. These problems occur regardless of programming language, device or target platform, and they are further exacerbated by the existing plethora of languages, devices and platforms. In short, developing an embedded application with run-time adaptability is by force of circumstance a complex and time-consuming endeavor (e.g., [14]).

To reduce the development burden, there have been some attempts to support adaptability at several levels and through different mechanisms, for example through context-oriented programming (e.g., [7]). However, no current solution has provided the necessary infrastructure to achieve a flexible and domain-tailored approach. Considering the existing background and related work, the main contribution of the present work is an approach for the development of adaptable software applications for embedded systems based on a domain-specific language (DSL). Our approach can be applied to other fields besides embedded systems, however we emphasize on these types of systems since due to their characteristics, they are often more highly constrained than others, and thus in need for adaptive solutions.

The DSL enables the high-level specification of adaptation policies and strategies, using a flexible and simplified way of defining the rules that produce the necessary run-time reconfigurations, in an external way to the main application logic. The usefulness of run-time adaptability in embedded systems, as well as the advantages of having a dedicated approach to specify and manage the strategies for adaptation, are illustrated through a set of experimental results in a case study application. This work builds upon our preliminary DSL assessment in [18], and the case study is based on a previously developed prototype system for mobile robot navigation [17]. The case study allows us to demonstrate the feasibility of our DSL-based approach to flexibly specify adaptable behavior and easily verify its consistency, when compared to a general-purpose language such as Java.

The remainder of this paper is organized as follows. In Section 2 we introduce an adaptation-aware application architecture. Section 3 describes the DSL. Section 4 demonstrates the use of the DSL in an application to mobile robot navigation. Section 5 discusses relevant related work, and finally Section 6 concludes the paper.

2 Architectural Decoupling for Adaptations

Adaptation is a process, which modifies the behavior of a system in order to improve the interaction with the remaining components or the outside environment. Therefore, an application is adaptable when it is possible to adjust the execution of its main logic, thus making the application behavior dynamic. Our work focuses on the adaptation logic as an independent adaptation policy entity, that defines a procedure, composed by multiple strategies, which are plans of action that achieve a certain goal through a set of adaptation rules. This separation of application and adaptation concerns allows for the reuse of adaptation mechanisms, a higher-level of abstraction, potential for scalability and extensibility, and a simplified approach to integrate adaptations with software applications.

Figure 1 presents a diagram depicting the main entities and relationships involved in a model where the adaptation logic is external to the application and takes the form of one or more adaptation policies that can target specific reconfiguration concerns (e.g., a policy mainly targeted at reducing the energy consumption of the application; or a policy mainly targeted at increasing the performance of the application). Additionally, an application is influenced by (i) a set of user requirements, i.e., functional and non-functional constraints that the application must comply with in order to perform as intended; (ii) an environmental context, i.e., properties related to the operating conditions (e.g., sound level, light intensity); and (iii) the system where it executes, i.e., the execution platform and the infrastructure (e.g., battery level, network resources).

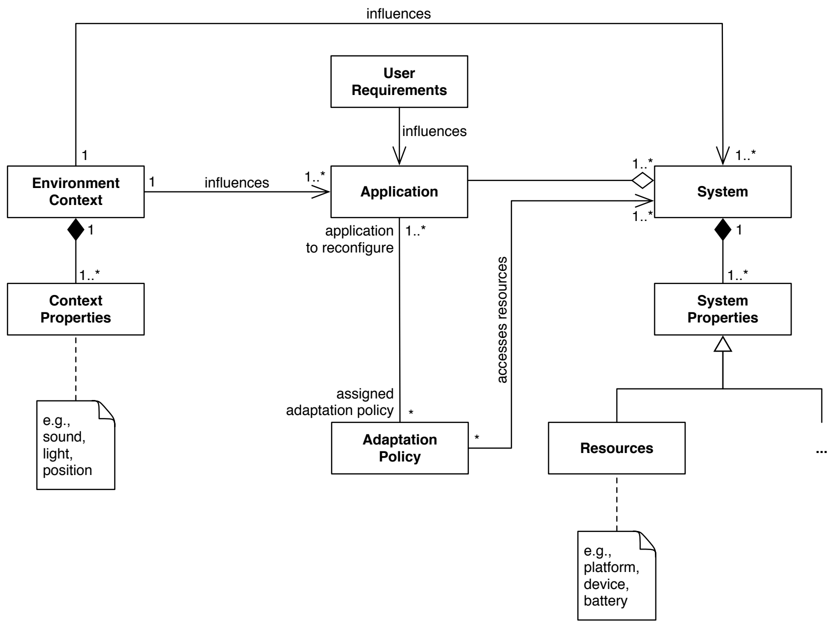


Figure 1 General entity diagram for an application model with an independent adaptation entity.

Moreover, in general, and regardless of software architecture, coding style, etc., an application comprises a series of computational steps, algorithmic components, and input/output parameters. As such, many embedded applications include components that can be configured via different parameters that influence the computational impact of the system as a whole, in terms of several observable metrics such as accuracy, execution time, energy, power, CPU load, quality-of-service, etc. This configurable interface allows for the selection of a number of system configurations thus enabling dynamic and adaptive application behavior.

In order to specify the dynamic behavior of embedded applications, we implement the adaptation logic through a DSL aiming at abstracting the adaptation concerns from the main application logic. DSLs offer substantial gains in expressiveness and ease of use compared with general-purpose programming languages (GPLs) in their domain of application, since they provide a notation close to an application domain, and are based only on the concepts and features of that domain [5, 12]. Given the existence of numerous programming languages, platforms and devices, a DSL-based approach that would allow for a single specification to be deployed in multiple environments would be very useful.

The adaptive behavior specified with the DSL can then be coupled with software applications in embedded systems through numerous mechanisms, such as through joint compilation, interpretation at run-time, mapped to another independent software component, deployed to another processor in a multicore embedded system, amongst other options. To this end, the proposed DSL is a specification mechanism that allows: (i) domain specificity by providing abstractions for commonly used adaptation actions; (ii) flexibility due to the adaptation independence from the application code and logic; (iii) portability and interoperability, targeting multiple embedded platforms and supporting several programming languages; (iv) verifiability and conflict detection, as domain abstractions allow for an easy understandability of the specified behavior; (v) productivity and comprehension improvement; and also (vi) an easier way to specify and deploy, for the same application, different strategies for different environments/services and/or target devices (e.g., in software product lines).

Although the use of a DSL is an efficient solution for the problem of abstracting and externalizing the adaptable behavior of embedded systems and applications, we also studied and considered other approaches to address the same problem, namely a DSL embedded in a GPL, and the use of a domain-specific library within a GPL. These alternatives represent interesting solutions, but they fall short as an independent solution, since either extending an existing GPL or using a library would be more restricting in terms of interoperability among different platforms and reusability between different systems.

3 A DSL for the Specification of Adaptations

The proposed DSL focuses on adaptation concerns, exposing high-level constructs for looping and setting periodic tasks, conflict avoidance, condition testing, time and memory evaluations, among others. These constructs allow for the flexible specification of adaptation policies that can range from simple parameter changes to complex code reconfigurations.

In the proposed DSL, an adaptation policy is specified as a set of *strategies* comprising four sections: *declarations*, *operations*, *rules*, and *code*. Figure 2 presents a model of an adaptation policy with its main structural components.

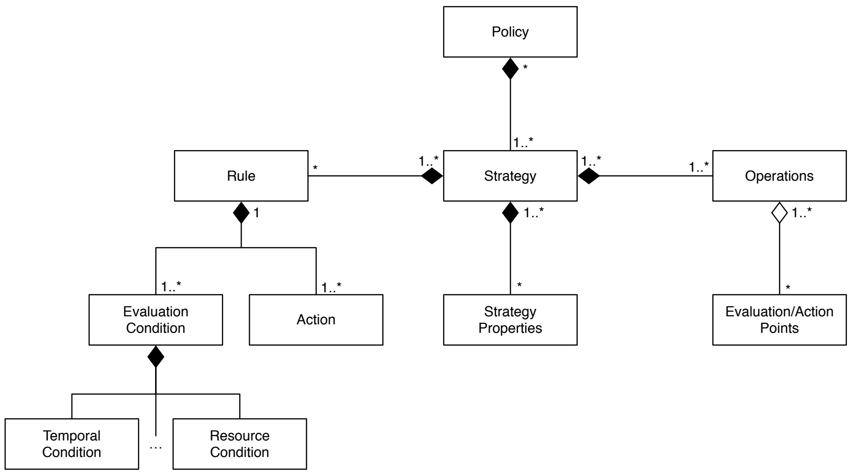


Figure 2 Overview model for an adaptation policy entity.

Declarations are reserved for information that is required for the specification of the adaptation process (e.g., variables to be used, algorithm parameters, imported functions). *Operations* specify mainly system interfacing with information on where the adaptation rules will be triggered (e.g., connection points). The *rules* section specifies the adaptation actions that apply reconfigurations to the application. Finally, in the *code* section, functions and other components can be defined in another programming language (e.g., C) to promote extensibility. The DSL also provides abstractions to access some application- and system-related properties. This access to certain characteristics is provided by the monitoring tasks that in a later stage are incorporated into the application by the implementation toolchain.

An example of an adaptation policy specified with the DSL, associated with the inference of human activity context, is shown in Listing 1. In this example application, the context is calculated by an inference process imported to the DSL (line 2). Operationally, locations for rule evaluations are defined (lines 4–7). The *rules* section (lines 9–18) defines two rules that express periodic adjustments to the inference when the battery of the device reaches a low level or when the computation time takes longer than the defined rate. In order to acquire data on the energy level of the device, an additional function in Java is defined in the *code* section (lines 20–22), whose details are omitted for simplicity.

Listing 1 DSL specification for an adaptable activity inference system.

```
1  strategy activityAdaptationStrategy{
2     imported function [String context] fftKnnInf(int FftSamples=2048);
3  
4     operations{
5        r1 evaluation point "location_1";
6        r2 evaluation point "location_2";
7     }
8  
9     rules{
10       r1: every(60sec){
11          if(code.java.getEnergyLvl() < 30){fftKnnInf.FftSamples=512;}
12          else{fftKnnInf.FftSamples=2048;}
13       }
14       r2: every(10sec){
15          if(fftKnnInf.rate < 1Hz){fftKnnInf.FftSamples--;}
16          else{fftKnnInf.FftSamples++;}
17       }
18    }
19 
20    code.java{
21       double getEnergyLvl(){ (...) }
22    }
23 }
```

3.1 Language Components

A policy is the adaptation “program” that defines strategies of adaptation, further composed of multiple properties and other components. A strategy is a high-level entity that embodies the mechanisms of adaptation that are tailored to a specific purpose (e.g., energy-aware adaptations, execution time compliance adaptations). Furthermore, the strategy entity can also be defined to receive configuration parameters in order to be adaptable to different situations or conditions. This allows for the strategy elements to be reused across environments with different characteristics.

Declarations Within a strategy, the *declarations* section allows for the specification of variables, functions, and other components to be used in all other sections. Here, variable declaration has similar semantics to other programming languages with the additional inclusion of valid value ranges that the variable may assume. Functions from the source application can be imported and are linked to the respective implementations. This section may also define default values for the input parameters of the imported functions.

Operations The *operations* section describes important operational blocks in the computational process. The main objective of this section is to provide information on the execution of the application through execution blocks, giving an understanding on the execution flow and on interfacing connection points. The *operations* section is built with a main execution block that can be associated with execution properties. Other alternative execution blocks can also be defined. Moreover, sub-block structures can be defined to frame specific steps or components of the application's workflow. Such sub-block structures are used to concentrate operation steps that may be activated or deactivated, allowing a more dynamic and powerful adaptation structure. Operational connection points define references to locations where the adaptations will be triggered and therefore executed in the application's source code. Connection points only define the target location for rule executions; other adaptation properties, such as execution periodicity are defined in the rules section. Points can be associated with function calls or with specific locations in the source code, identified by special-purpose annotations (e.g., "`//@ evaluation point location_1`" for Java).

Rules The *rules* section specifies multiple adaptation actions, responsible for performing the necessary changes that control the behavior of the target application. An individual rule is composed of an identifier, a triggering condition (e.g., periodically or by events) and the adaptation action code. The execution of a rule is atomic in the sense that its operations either all occur, or nothing occurs, denoting an atomic transaction. Rule management is conducted in this section and thus adding, removing or modifying existing rules can be accomplished without scattered changes. The existence of multiple rule blocks assigned to different conditions or events could cause conflicts, as incompatible actions could be invoked if multiple rules where activated simultaneously. However, with a centralized location for the adaptation rules, verification and validation can be more easily accomplished. The resolution of conflicts is performed through prioritization, where rule blocks are prioritized by their order of specification (default conflict solver) or through explicit prioritization using an `evaluate` control command (specific conflict solver), where boolean logic and prioritization functions (e.g., `first`) can be applied. Additionally, predicates can be used for finer grain control of the flow of rule execution, providing a simpler yet powerful mechanism to protect against incompatibilities arising from rules which for example try to access common resources.

Code The *code* section allows the developer to extend the DSL by adding functionality that is not present (e.g., platform-specific code), or to extend the application without making changes directly to it. Variables and functions defined in this section have a global scope and thus can be used in all other sections. The *code* section must indicate the programming language in which the enclosed programming code will be specified. The use of this section comes with the cost of reducing the DSL's interoperability, as language- and platform-specific code may be introduced here.

3.2 Policy Correctness

In an adaptation policy, it is likely that reconfigurations, such as a parameter changes, may lead to incompatibilities, execution errors and integrity conflicts. Addressing these conflicting situations is of paramount importance for policy correctness, and thus verification and validation are required processes conducted on several levels for evaluating different aspects (e.g., adaptation rule conflicts), and different targets (e.g., all or only specific DSL sections). In this paper, we focus on analyzing the problems and conflicting situations for the *rules* section, which is a core component in the specification of the adaptable behavior. In particular, rules and their actions are potentially in conflict if they: (i) share at least a subset of triggering conditions, causing more than one rule to be triggered to execute at the same time; (ii) manipulate a subset of the same parameters, which may cause incompatible behavior in the execution of actions; (iii) override or overlap themselves, causing one rule to become unreachable or redundant; or (iv) are incompatible due to requirements or objectives, since even with different triggering conditions, or different actions, there can be additional specific functional requirements by the stakeholders.

In order to better perceive and assess the set of adaptation rules defined, we propose a verification process based on automata theory [10]. This approach allows to model the *rules* section into directed non-deterministic finite automata where conflicting situations can be identified, both statically and dynamically. Adaptation rules can be viewed as an automata with a set of adaptation states, an alphabet of different triggering conditions, and a transition function that maps the transformation from one adaptation state to another, according to the provided input condition. Through automata operations other properties and characteristics can be extrapolated, for example, (i) the product operation provides the combination of all possible adaptation states and transitions; (ii) minimization allows the removal of useless and unreachable states; or (iii) intersection to identify common states. Furthermore, with an automata-based model, adding, removing and changing rules, can be easily perceived and thus analyzed for conflicts.

3.3 Interfacing and Implementation

In this work, we developed a toolchain that incorporates the independent adaptations into the target source code of the application to be adapted. First, the developer must analyze and evaluate the application source for possible adaptations. Second, depending on the application there may be minor modifications of the source code to be done to explicitly identify functions, inputs, and outputs. With the application better prepared to be adapted, it is possible to specify the adaptations using the DSL. The adaptation code specified with the DSL must then be verified and validated to assess potential errors and conflicts. With a valid DSL adaptation specification, the DSL code is translated into the target source code language (e.g., DSL $\rightarrow$ Java). The compilation and code generation allow the weaving of the adaptations into the application's original source code, and so the adaptations are incorporated at compile-time. With the adaptations incorporated, the complete application can be compiled using standard compilers (e.g., javac). The adaptable application can now be deployed to the execution environment.

4 Application to Mobile Robot Navigation

This section presents a case study for adaptation specification based on our own previous work [17], where we developed a navigation system comprising a Lego NXT Mindstorms robot together with a Nokia N80/N95 smartphone (see Figure 3). The mobile robot was intended to explore the environment while simultaneously inferring its location. In this system, the mobile robot is controlled by the smartphone, where navigation algorithms are executed, generating controls that are transmitted back to the robot.

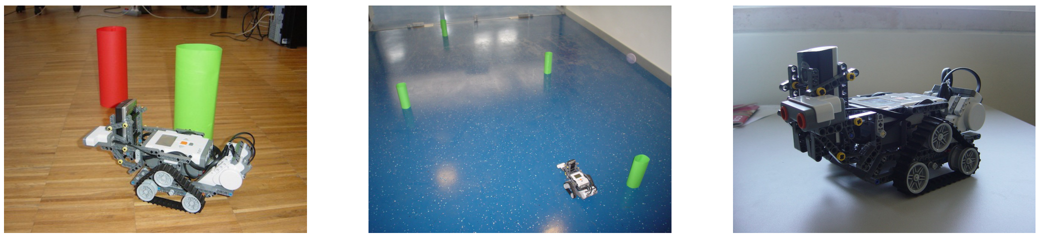


Figure 3 Mobile robot and smartphone system used in the case study application.

The concept of the application is that from continuously captured images obtained from the camera of the smartphone, special landmarks and features can be detected and used to update an internal model that uses such information to both navigate and locate itself in the environment. The application is composed of multiple algorithms, which expose several characteristics and input parameters (e.g., number of samples for computing the location) whose configuration impacts both the output and the processing requirements of the navigation (e.g., execution time, memory consumption). Managing these characteristics allow the use of adaptations for optimizations and for guaranteed continuous execution. As localization is inherently uncertain, it has been addressed with probabilistic methods, namely *particle filters* [8], which is the method used for adaptation in this case study. Additionally, for experimental testing, two setups were used: Setup 1 – a Java ME Platform SDK 3.0 mobile device emulator; and Setup 2 – a Nokia N95 smartphone.

4.1 Particle Filter Algorithm and Adaptation Analysis

The particle filter algorithm is used to track the evolution of the robot's pose (i.e., position and orientation) by building a sample-based representation, which approximately estimates the state of the robot's pose. The set of samples used for estimation are known as particles, and represent at each timestamp a hypothesis of what the true state of the robot's pose might be (an estimation based on simulation). The evaluation of the multiple hypothesis, given from the particles, allows for an overall global estimation of the correct robot's location. Structurally, the algorithm is composed of three main phases: *prediction*, *update* and *resample*; which are executed and looped over time, as the robot moves within its environment. Figure 4 represents a simulation of the estimation model for localization based on the particle filter algorithm, depicting the mobile robot, the particles, and the best particle estimation.

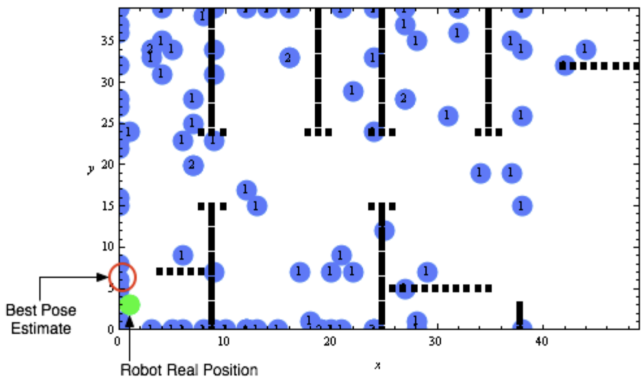


Figure 4 Mobile robot and particle positions and weights after several iterations of the algorithm, depicting also the position of the best particle, which is an estimate of the robot's real position.

The implementation used in this work is based on the approach presented in [16] and is applied for global localization, i.e., identification of the robot's position in an *a priori* known map. In this case study, the environment map is viewed as a 2-dimensional occupancy grid, and accuracy measurements are performed with the euclidean distance between the real robot position and the best overall estimate computed.

Through analysis and experimental testing conducted on the algorithm, for higher accuracy, the number of particles should be as high as possible, limited, however, to the size of the environment and the amount of computational resources available (to improve efficiency). The size of the environment influences the computational complexity of the algorithm, as larger maps require more particles and more movements to produce reasonable results, whilst also inheriting a higher degree of uncertainty in the results produced. Also, the more the robot moves in the environment, the more time will be available for estimation, therefore producing higher certainty in the location estimation, due to continuous refinements.

4.2 Adjusting the Number of Particles

One of the most relevant problems detected with the implementation performed in [17] was the difficulty to define the finite number of particles. When choosing the number of particles to use, it is very important to take into consideration the available computational resources, the time constraint to comply and the navigation requirements. A low number of particles may not be enough to provide a good pose estimate, while a high number may provide a better estimation, but at a much higher computational cost, which may not be feasible. Using a fixed number of particles will neither be effective in terms of accuracy, neither in optimizing the computation of the algorithm in the presence of varying conditions.

Figure 5 shows how different map sizes with different particle numbers influence the euclidean distance of the estimation to the real robot position (i.e., accuracy), execution time, memory used, and power consumed.

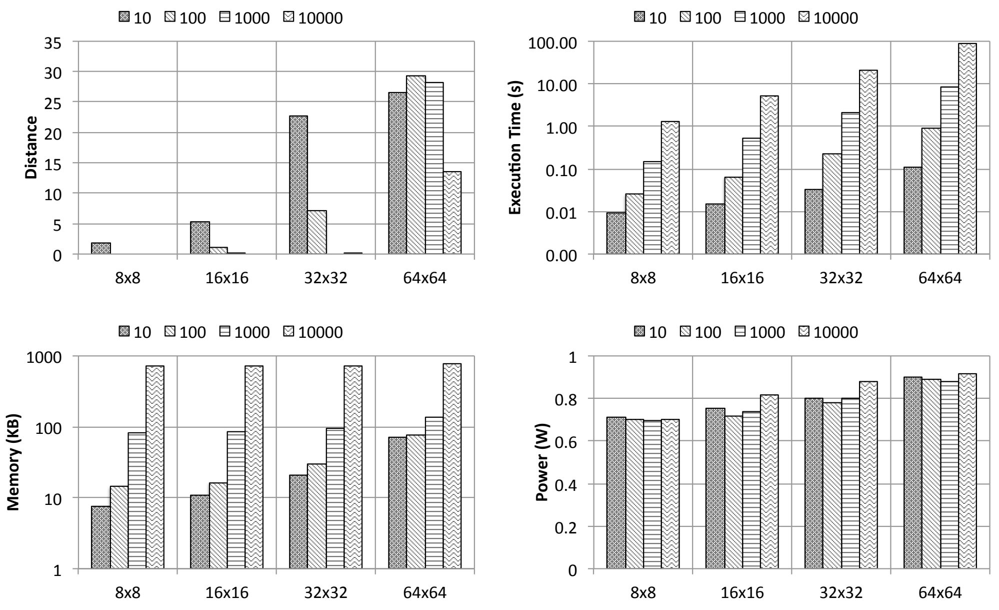


Figure 5 Measurements for distance, execution time, memory (using setup 1), and power (using setup 2) according to different map sizes and number of particles (10, 100, 1000, 10000).

Possible adaptation policies could consider the number of particles as a function of the environment map size, or as a function of the available execution time or free memory for computation. For the same number of particles, as the map size increases, the euclidean distance, execution time, memory and power all increase. For the same map size, increasing the number of particles improves accuracy (i.e., decreases distance), but increases execution time and memory, and to a lesser extent power consumption.

4.2.1 Adjusting to the Map Size

As the localization is accomplished through a map of the environment, in order to save memory and reduce the execution complexity, the entire map might not be completely loaded or only be provided on demand. With each new map, comes the possibility to reconfigure the number of particles, for example, as a function of the map width and height. Experiments conducted using this adaptation policy suggest that adapting the number of particles according to the map size, i.e., $(width \times height)$, $(width \times height)/2$, and $(width \times height)/3$, represent more efficient solutions both in accuracy and in the execution time and memory, than with a fixed number of particles, i.e., 10, 50, 100, 500, 1000, and 5000.

A DSL specification for this adaptation scenario is presented in Listing 2. The specification imports two functions, one encapsulating the particle filter algorithm – `particleFilter` – and another for retrieving the current map size – `getMapSize` (lines 1 and 2, respectively). There is one rule defined – `r_map` – to adjust the number of particles used in the algorithm, according to the map width and height (lines 9–11). Before starting the execution of the particle filter algorithm, the map size is used to recalculate and assign the number of particles to be used, as specified in the operations section (line 5).

Listing 2 DSL specification for the number of particles adaptation considering the map size.

```
1  import function particleFilter(int particles=100);
2  import function [int w, int h] getMapSize();
3
4  operations{
5     r_map evaluate before particleFilter;
6  }
7
8  rules{
9     r_map: every(particleFilter){
10       particleFilter.particles = getMapSize().w x getMapSize().h;
11    }
12 }
```

4.2.2 Adjusting to Computational Constraints

Additionally, new behavior can be added in order to further adapt in accordance to requirements specific to the device energy, execution time or memory conditions. Even using an adaptable number of particles defined as shown in Section 4.2.1, which contributed to better efficiency in the tradeoff between accuracy and computational requirements, the change and variation over time in the device's computational conditions (e.g., device energy dropping below a 20% threshold level) would cause additional difficulties on execution.

Herein, we extend the adaptable behavior presented before with additional reconfigurations that further adjust the number of particles used when the device's energy level is below 20%, when it is consuming a large amount of memory, and when in violation of an execution time constraint. To demonstrate the adaptable behavior, we designed a scenario of navigation, where the mobile robot explores a territory composed of eight areas (two areas with size 8×8, two with size 16×16, two with size 32×32, and two with size 64×64). Using this adaptation specification, the original algorithm is now equipped with reconfigurations that improve its accuracy in situations where both the computational conditions and maps change.

Listing 3 DSL specification for the adaptation of the number of particles according to map size, device energy level, time, and memory limitations.

```
import function particleFilter(int particles=100);
import function [int w, int h] getMapSize();
import function [int level] getEnergyLevel();

operations{
   r_energy evaluation before particleFilter;
   r_map evaluation before particleFilter;
   r_time evaluation point "resample";
   r_memory evaluation point "resample";
}

rules{
   evaluate: order(r_map, r_energy, r_time, t_memory);

   r_map: every(particleFilter){
      particleFilter.particles = getMapSize().w x getMapSize().h;
   }
   r_energy: every(5sec){
      if(getEnergyLevel() < 20){ <p_energy_down == 0>
         particleFilter.particles -= particleFilter.particles / 2;
         <p_energy_down = 1>
      }else{ <p_energy_down == 1>
         particleFilter.particles += particleFilter.particles / 2;
         <p_energy_down = 0>
      }
   }
   r_time: every(resample){
      if(particleFilter.elapsed_time >= 20){
         particleFilter.particles -= particleFilter.particles / 4;
         <p_time_down = 1>
      }else{ <p_memory_down == 0 && p_energy_down == 0>
         particleFilter.particles += particleFilter.particles / 4;
         <p_time_down = 0>
      }
   }
   r_memory: every(resample){
      if(particleFilter.memory_consumed >= 3000){
         particleFilter.particles -= particleFilter.particles / 4;
         <p_memory_down = 1>
      }else{ <p_time_down == 0 && p_energy_down == 0>
         particleFilter.particles += particleFilter.particles / 4;
         <p_memory_down = 0>
      }
   }
}
```

From experimental results comparing the adaptable specification for the particle number in contrast with three different fixed values (i.e., 50, 1000, 5000), the average and cumulative distance is lower than in all other fixed particle number tested, meaning that the overall accuracy of the estimation was higher. Regarding execution time and memory, the adaptable version is comparable to the use of 1000 fixed number of particles (on average the adaptable specification used 860 particles), however this is higher than for fixed 50 particles and much lower than for 5000 fixed particles. Also, the energy consumed in the scenario was lower for the adaptable number of particles (74mAh in contrast to 77mAh for fixed 1000 particles).

The advantages of this adaptable behavior and the possibility of further configuration to be more tailored to other operating conditions, stimulate the desire to specify it through our DSL approach. A possible specification for such behavior is presented in Listing 3, that shows that with the additional rules (when in comparison to Listing 2) it is possible to further refine the number of particles used, taking into account how much energy (lines 18–26), time (lines 27–35), and memory (lines 36–44) are available.

4.2.3 Dealing with Conflicts

Considering Listing 3 with four different rules for adaptation, the need for an evaluation order and priority becomes imperative. From the analysis of the rule automata, some transitions were detected as potential conflicting situations. These problematic situations were detected automatically due to the similarity between code actions, i.e., the same algorithm parameter was manipulated and the operation performed is the opposite (addition and subtraction on `particleFilter.particles`). To ensure the correct adaptive behavior, and according to the information perceived from the automata, additional restriction predicates needed to be defined. For example, considering the case when for low energy conditions the number of particles is reduced, it should not be increased if the time constraint was satisfied. The automata-based detection of such a problematic situation is presented in Figure 6.

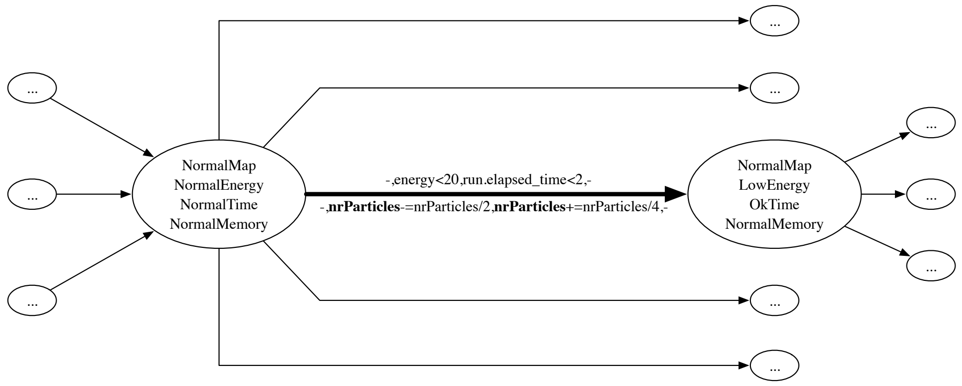

...
NormalMap
NormalEnergy
NormalTime
NormalMemory
-,energy<20,run.elapsed_time<2,-
-,**nrParticles**-=nrParticles/2,**nrParticles**+=nrParticles/4,-
NormalMap
LowEnergy
OkTime
NormalMemory

Figure 6 Detection of a conflicting situation through the cartesian product of the individual automata representing the rules. For simplicity, these automata are only an excerpt.

Figure 6 depicts a situation where the conflicting manipulation of the `particleFilter particles` parameter with contradictory operations was detected (transition in bold) in the transition to an adaptable state due to energy becoming low (decreasing particles) and the execution time being satisfied (increasing particles).

For such task, the specification in Listing 3 defines predicates for the execution of the adaptation rules. The predicates define additional evaluation conditions for the adaptation rules. This evaluation prioritization guarantees, in this case, that the `r_map` has no dependency towards other rules, `r_energy` rules depends only on itself, `r_time` and `r_memory` when in violation can be executed, however, when not in violation, they will only increase the number of particles, if and only if, the other rules have not been executed. Concretely, for example, considering rule `r_time`, when in violation of the time constraint, it sets the predicate `p_time_down` to 1 (an assignment for true → execution occurred) meaning the action for number of particles decrease occurred. If it is not in violation of the time constraint, then an increase of the number of particles could take place, however, in order to be compatible with the other rules, the action only executes if the predicates `p_memory_down` and `p_energy_down` are assigned 0 (meaning that they are set to false and were not executed).

4.3 Discussion

Besides defining the adaptable behavior with the DSL, for comparison we implemented the equivalent behavior inside the application logic, since it is the most common procedure to incorporate adaptations. Data on the comparison between the adaptable behavior specified in a DSL version and in a Java (GPL) version are provided in Figure 7, where case 0 is the default application, case 1 adds the adaptations due to the map size (Section 4.2.1), and case 2 adds the adaptations due to computational constraints (Section 4.2.2). The transition cases 0 → 1 and 1 → 2, focus on the changes from one case to the other.

DSL	1	2	1 → 2	
LoC	13	41	28	215.38%
Words	31	95	64	206.45%
Imports	2	3	1	50.00%
Operation instructions	2	5	3	150.00%
Rule blocks	1	4	3	300.00%
Transition points			3	
Similarity			15%	

Java	0	1	2	0 → 1		1 → 2	
LoC	5229	5236	5325	7	0.13%	89	1.70%
Methods	305	305	306	0	0.00%	1	0.33%
Method LoC (avg)	12.53	12.55	12.65	0.02	0.15%	0.10	0.81%
Attributes	131	132	135	1	0.76%	3	2.27%
McCabe Cyclomatic Complexity (avg)	2.45	2.45	2.48	0	0.00%	0.03	1.31%
Altered Methods				2		3	
Transition points				3		2	
Similarity				56%		56%	

Figure 7 Comparison between the particle filter adaptations specified in the DSL and in Java. LoC of the DSL may slightly vary from the specifications presented due to code formatting.

From the analysis of Figure 7, it is possible to verify how the specification of adaptable behavior reflects itself both in the DSL and in the GPL. The main observations that can be drawn from this comparison are:

- Average textual similarity for the Java versions was conducted solely on the modified methods. The unaltered code was not considered as its large size (due to the rest of the application) would overwhelm the similarity results.
- Transition points for the DSL consist in an added function, and rule connection points and their corresponding actions. In Java, they consist mostly on method changes and new methods; and most importantly on different files (two to three separate files).
- Regarding lines of code, in absolute values the DSL increase from case 1 → 2 is three times less than in the GPL. Of course, due to the size of the Java code, i.e., many classes and packages in comparison to one DSL specification file, in relative percentage terms

the DSL increase in lines of code is much higher.

- In the DSL all the modifications necessary to be made in order to transform the specification from case 1 → 2 were performed solely in one specification file. In contrast, in the Java version, three different classes had to be modified. Also, the Java code introduced for adaptations is intertwined with the application logic.
- Due to the evaluation and action locations of the adaptation rules, in the DSL the modifications were confined to the rule block section, while in the Java version, the adaptable behavior rules where placed at two different code locations.
- The use of predicates allows the correct prioritization and also the conflict avoidance between the different rule actions. In Java, the predicates required added control variables, branches and overall more confusing coding. The predicates were also embedded within some instructions, becoming intrusive to already defined statements of code.
- The rule relative to the adjustment of particle number in consequence of the map size was defined in the Java code in a different location than the other rules. In the DSL all rule behavior is defined in one location.

It is important to mention the minimal additional effort that was required to perform minor modifications to the original application, in order to prepare the application for the weaving of the generated adaptable behavior code specified with the DSL.

5 Related Work

Adaptation in software applications has commonly been accomplished through the use of conditional expressions, parameterization, and exceptions [4]. In today's dynamic computational environments and requirements, autonomic computing and true adaptive behavior cannot simply be accomplished through such methods, as they are error-prone and introduce complexity by intertwining adaptation and application behaviors, scaling poorly and thus rendering software evolution and maintenance hard. To this end, several approaches have been proposed to support software adaptations, such as (i) frameworks and architectures, (ii) context-oriented programming, and (iii) dynamic aspect-oriented programming. Other approaches such as feature-oriented programming and change-oriented software engineering are also of relevance. Our approach differs as we focus on a completely independent domain language, which does not extend nor is tailored specifically to another host language, platform or environment, offering more flexibility, structure and comprehension.

Framework and architectural models These offer dynamic adaptation infrastructures, however with no wide adoption, possibly due to limited adaptation support or due to effectively low adaptation facilities in practice [14]. Some examples of such architectural approaches include MADAM [4] and Rainbow [6]. Furthermore, these approaches require software to be developed according to new and complex component-based architectures which are not ideal solutions to already developed and deployed applications.

Context-Oriented Programming (COP) These concepts have commonly been implemented as extensions to several languages [1] (e.g., Subjective-C [7]). However, each language extension comes with its own approach to the COP paradigm and implementations commonly suffer from execution overhead [1]. The objective of COP languages is to modify the behavior of a program by associating code definitions with context-related layers that are activated or deactivated according to the current context. The behavior of objects and methods thus depends on the context in which they execute. Autonomic behavior can be accomplished by executing code variations in reaction to changes in context states. Although adaptations are performed in applications, the adaptation specification distinguishes from our approach as it depends, and is tailored, solely to context states. Also, the context-based adaptations are normally embedded within the application code itself.

Dynamic Aspect-Oriented Programming (AOP) Aims at improving the separation of concerns and thus can be used to encapsulate the adaptations that are required to implement an autonomic system [9]. With this technique, an adaptation can be created using aspects, and woven statically or dynamically, providing an extremely powerful tool to allow the application to be modified [15] (e.g., AspectJ [11]). The AOP concerns are similar to our implementation approach, as aspects alter the behavior of the application code by inserting additional concerns, which can be adaptation-related, at various points in a program (similar to our evaluation points). In fact, an aspect-oriented approach can be used in our work to implement the necessary modifications at the application code level. However, our approach differs from traditional AOP as we focus on specifying adaptation both at the code level and at the design level, with constructs tailored specifically to the adaptation domain in order to define relations between adaptations, the periodic evaluation for adaptations triggering, etc.

6 Conclusions

In this paper we proposed a DSL-based approach to specify adaptable behavior in embedded applications in a flexible way and externally to the main application logic. Using this DSL-based approach, adaptation strategies can be specified and modified without tampering with the application code. Furthermore, different strategies defined in the DSL can be shared and deployed over different platforms and programming languages, promoting fast prototyping. We illustrated the application of the proposed approach in the case study of a mobile robot with a navigation application running on a smartphone. With the DSL, it was possible to easily and flexibly specify the adaptable behavior of that application. The experimental results highlight the benefits of specifying the adaptable behavior through the DSL-based approach, when compared to implementing the same behavior by re-programming directly the application. We also showed how adaptation strategies are specified, and evaluated the impact of modifying and extending an existing strategy with new rules. In short, we demonstrated that not only is the DSL code more succinct, but changes and improvements are also easier to implement.

Acknowledgements The work presented was supported by *Fundação para a Ciência e a Tecnologia* (FCT) under grant no. SFRH/BD/47409/2008.

References

1 Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A Comparison of Context-Oriented Programming Languages. In *Proc. of the Int'l Workshop on Context-Oriented Programming (COP'09–ECOOP'09)*, pages 6:1–6:6. ACM, 2009.

2 Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A Survey on Context-Aware Systems. *Int'l Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007.

3 Davide Figo, Pedro C. Diniz, Diogo R. Ferreira, and João M. P. Cardoso. Preprocessing Techniques for Context Recognition from Accelerometer Data. *Personal Ubiquitous Computing*, 14(7):645–662, 2010.

4 J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

5 M. Fowler. *Domain-Specific Languages.* Addison-Wesley. Pearson Education, 2010.

6 D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

7 Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective–C: Bringing Context to Mobile Platform Programming. In *Proc. of the 3rd Int'l Conf. on Software Language Engineering (SLE'10)*, volume 6563 of *LNCS*, pages 246–265. Springer, 2010.

8 N.J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation. *Proc. of the IEEE Radar and Signal Processing*, 140(2):107–113, 1993.

9 Philip Greenwood and Lynne Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. *Proc. of the 2004 Dynamic Aspects Workshop (DAW'04), RIACS*, pages 76–88, 2003.

10 John Hopcroft, Rajeev Motwani, and Jeffrey Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

11 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *LNCS*, pages 327–354. Springer, 2001.

12 Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37:316–344, December 2005.

13 M. Mikalsen, J. Floch, N. Paspallis, G.A. Papadopoulos, and P.A. Ruiz. Putting Context in Context: The Role and Design of Context Management in a Mobility and Adaptation Enabling Middleware. In *Proc. of the 7th Int'l Conf. on Mobile Data Management (MDM'06)*, pages 76–83, 2006.

14 Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th Int'l Conf. on Software Engineering (ICSE'08)*, pages 899–910. ACM, 2008.

15 Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In *Proc. of the 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147. ACM, 2002.

16 Ioannis Rekleitis. *Cooperative Localization and Multi-Robot Exploration.* PhD thesis, School of Computer Science, McGill University, Montréal, 2003.

17 André C. Santos. Autonomous Mobile Robot Navigation using Smartphones. Master's thesis, Instituto Superior Técnico – Technical University of Lisbon, 2008.

18 André C. Santos, Pedro C. Diniz, João M. P. Cardoso, and Diogo R. Ferreira. A Domain-Specific Language for the Specification of Adaptable Context Inference. In *Proc. of the IEEE/IFIP Int'l Conf. on Embedded and Ubiquitous Computing (EUC'11)*, pages 268–273. IEEE Computer Society, 2011.

19 Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. Improving Energy Efficiency of Location Sensing on Smartphones. In *Proc. of the 8th Int'l Conf. on Mobile Systems, Applications, and Services (MobiSys'10)*, pages 315–330. ACM, 2010.

Part VI

Natural Language Processing

Dictionary Alignment by Rewrite-based Entry Translation

Alberto Simões[1] **and Xavier Gómez Guinovart**[2]

1 **Centro de Estudos Humanísticos, Universidade do Minho**
Campus de Gualtar, Braga, Portugal
ambs@ilch.uminho.pt

2 **Galician Language Technology and Applications (TALG Group)**
Universidade de Vigo, Galiza, Spain
xgg@uvigo.es

Abstract

In this document we describe the process of aligning two standard monolingual dictionaries: a Portuguese language dictionary and a Galician synonym dictionary. The main goal of the project is to provide an online dictionary that can show, in parallel, definitions and synonyms in Portuguese and Galician for a specific word, written in Portuguese or Galician.

These two languages are very close to each other, and that is the main reason we expect this idea to be viable. The main drawback is the lack of a good and free translation dictionary between these two languages, namely, a dictionary that can cover lexicons with more than one hundred thousand different words.

To solve this issue we defined a translation function, based on substitutions, that is able to achieve an F_1 score of 0.88 on a manually verified dictionary of nine thousand words. Using this same translation function to align a Portuguese–Galician dictionary we obtained almost 50% of the dictionary lexicon (more than eighty thousand words) alignment.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases translation, rewrite system, dictionary, dictionary alignment

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.237

1 Introduction

Dicionário-Aberto[1] [10] resulted from the transcription, validation, and annotation of a printed dictionary for the Portuguese language, compiled by Cândido de Figueiredo, and published in 1913. It was transcribed as a Gutenberg Project book, but the main goal for this task was the use of this dictionary to bootstrap an XML-encoded dictionary that could be enriched and expanded by the community, and that can be used in Natural Language Processing (NLP) tasks. The document was subject to different steps on semantic annotation and orthography modernization [8], and is currently being used for the extraction of different NLP resources [11, 9]. It is also available in a web site for online querying, both as a standard form, and as a RESTless server.

In the context of another project [5], a synonym dictionary for the Galician language [7] (check also Guinovart and Simões, this proceedings) was converted from Microsoft Word files to a semantic-rich XML file. This dictionary was also corrected, widened in lexical extension, and modernized, taking into account the current norms for the Galician language. At the

[1] Available at http://dicionario-aberto.net/

2^{nd} Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 237–247
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

moment the result of this work is not available to download because it is not finished yet, but the dictionary contents will soon be available in a web site for online querying.

Given the proximity of the two languages we decided that it would be interesting to show Galician definitions together with the Portuguese definitions. This would be useful for language researchers, but it can also be used to enrich a common thesaurus.

The main problem to bring this project to life is the alignment task: how to make entries from both dictionaries correspond to each other.

This task could be easy to execute if there was a bilingual dictionary. But there are few bilingual dictionaries between Portuguese and Galician, and the ones available are too small to allow the alignment of dictionaries with more than a hundred thousand entries.

The main contribution of this article is the test of the following hypothesis:

> *Given the proximity between the two languages, would it be possible to transform a Portuguese word in the dictionary into a Galician word just by applying a set of rewrite rules?*

The following section describes the translation function, what rewrite rules are used, and the order in which they should be applied. Section 3 describes two evaluation processes: the first one using a small Portuguese–Galician bilingual dictionary, that was hand-curated; the second one, using a bigger dictionary obtained by dictionary triangulation. In Section 4 the dictionary alignment process is performed, and the results discussed. Finally, we conclude with some final remarks.

2 Translation Function

The translation function that, given a set of valid Galician words (L_{gl}) and a Portuguese word (w_{pt}), returns a single[2] Galician word, will be denoted by $\mathcal{T}(L_{gl}, w_{pt})$, and is defined as a set of substitutions that rewrite a Portuguese word into a Galician translation.

Table 1 summarizes the performed substitutions. First, the Portuguese word is tried as a Galician word without any modification. If it is does not exist in the target lexicon, the substitutions are performed. The substitutions are ordered from more general substitutions to more specific ones (this was done manually, both from the authors knowledge of the two languages, and querying the dictionary to confirm the number of cases for each substitution). In some cases, less general rules needed to be performed first than more general ones, because of their interdependence. For example, the substitution from *-ção* > *ción* depends on the existence of the *ç* character that might be substituted by the *z* character, if the more general substitution *ç* > *z* is applied first. If they get applied in the wrong order the second substitution will not take place (as no *ç* character will be found), decreasing the number of correctly translated words.

Notice that some substitutions have two possible targets. In these cases, both possible words are maintained, and consequent substitutions will be applied to all words. That is why there are some substitutions that include the string being substituted as the substitution result: *im-* > *im-, inm-*. This rule will force that all Portuguese words with the prefix *im-* will be rewritten into two possible translations: the original one, and another one where *im-* was substituted by *inm-*.

[2] As explained below, the function generates, internally, a set of possible translations, but only one is returned.

At the end of the substitution process each possible translation is checked against the Galician lexicon (L_{gl}), and the first one that exists is returned (this one, w_{gl}, is the translation of w_{pt} using the translation function). This means that, internally, the translation function is over-generating words (both correct words and non-existent words), given that they can be filtered before returning, using the target lexicon.

To exemplify the rewrite rules, starting with the word *impassível*, we can derive:

$$\begin{array}{lll} \textit{impassível} & >_A & \textit{impasível} \\ & >_M & \textit{impasíbel, impasible} \\ & >_{AI} & \textit{impasíbel, impasible, inmpasíbel, inmpasible} \end{array}$$

The words from this list of generated words are then searched in the Galician lexicon and the first one that exists is returned as correct: *impasíbel*.

3 Evaluation

To evaluate the substitutions we performed two different runs, using two different dictionaries. The first one uses a small translation dictionary from Galician to Portuguese that was hand-curated. The second experiment was performed on a Galician–Portuguese dictionary obtained by triangulation using different pivot languages. The following sections explain the used metrics, detail the origin of these dictionaries, and present and discuss the obtained results.

3.1 Evaluation Metrics

The two evaluations were performed using hypothesis testing. Table 2 presents the Type I and Type II error matrix. Cell counts are computed as follows:

(TP) True Positives – a Portuguese word is correctly transformed by the translation function into one of the possible corresponding translations;

(FP) False Positives – the proposed translation for the Portuguese word is not the correct one, but is listed in the Galician lexicon (it is present in the dictionary as a translation for some other word);

(TN) True Negative – the proposed translation for the Portuguese word is not listed in the Galician lexicon, but is a correct translation. This can never happen because if the translation is correct, then it exists in the gold standard, and therefore, it will necessarily exist in the Galician lexicon (as it is computed from the gold standard). Thus, it is impossible to have such a word: $TN = 0$.

(FN) False Negative – whenever the proposed translation word does not exist in the Galician lexicon, and is not a correct translation. This happens every time the translation is not in the Galician lexicon (as it is computed from the translation pairs).

To evaluate the proposed substitutions we computed the usual metrics: accuracy, precision, recall and F_1 measure, using the standard formulae.

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{1}$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2}$$

Table 1 List of the translation function substitutions, by application order.

Identifier	Substitution	Examples
ID		*mesmo > mesmo, normativa > normativa*
A	`ss > s`	*passo > paso*
B	`j > x`	*sujeito > suxeito, injectar > inxectar*
C	`-ção > -ción,-zón`	*adivinhação > adiviñación, coração > corazón*
D	`ç > z`	*laço > lazo, carroça > carroza*
E	`nh > ñ`	*unha > uña*
F	`-dizer > -dicir`	*contradizer > contradicir, desdizer > desdicir*
G	`z ([eiéíêî]) > c`	*bronze > bronce*
H	`lh > ll`	*alho > allo*
I	`vr > br`	*livro > libro*
J	`-agem > -axe`	*arbitragem > arbitraxe*
K	`g ([eiéíêî]) > x`	*faringe > farinxe, agência > axencia*
L	`-ável > -ábel,-able`	*amável > amable, amábel*
M	`-ível > -íbel,-ible`	*possível > posible, posíbel*
N	`-velmente > belmente,-blemente`	*previsivelmente > previsibelmente, previsiblemente*
O	`-eio > -eo`	*alheio > alleo*
P	`-ância > -ancia`	*abundância > abundancia, alternância > alternancia*
Q	`-ência > -encia`	*abstinência > abstinencia, agência > axencia*
R	`-aria > -ería,-aría`	*livraria > librería, libraría; tesouraria > tesourería, tesouraría*
S	`-ário > -ario`	*operário > operario, vestiário > vestiario*
T	`-óri[oa] > -ori[oa]`	*absolutório > absolutorio, aleatória > aleatoria*
U	`-são > -sión,-són`	*ilusão > ilusión, brasão > brasón*
V	`-rão > -rón,-rán`	*padrão > padrón, alcorão > alcorán*
W	`-mão > -món,-mán`	*limão > limón, caimão > caimán*
X	`-ião > ión,-ián`	*ancião > anción, anfitrião > anfitrión*
Y	`-ício > -icio`	*edifício > edificio*
Z	`-óide > -oide`	*asteróide > asteroide*
AA	`-ídio > -idio`	*presídio > presidio*
AB	`-ânico > -ánico`	*mecânico > mecánico*
AC	`-édia > -edia`	*comédia > comedia*
AD	`-cimento > -cemento`	*reconhecimento > recoñecemento* (always as suffix, not as a word)
AE	`-m > -n`	*além > alén*
AF	`-crever > -cribir`	*escrever > escribir, inscrever > inscribir*
AG	`-u > -u,-o`	*mau > mao, museu > museo, ateu > ateo*
AH	`-var > -bar`	*reprovar > reprobar* (when *-var* is kept, full word matches the PT word)
AI	`im- > im-,inm-`	*imortalidade > inmortalidade, improvável > improbábel*
AJ	`qua- > cua-,ca-`	*quanticamente > cuanticamente, quadro > cadro*
AK	`qua > cua`	*adequado > adecuado*
AL	`-xão > -xón,-xión`	*inflexão > inflexión, paixão > paixón*
AM	`rv > rv,rb`	*preservação > preservación, estorvar > estorbar*
AN	`-iver > -ivir`	*conviver > convivir, sobreviver > sobrevivir*

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{3}$$

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} \tag{4}$$

For better understanding of these measures, the evaluation tables presented in the next sections include two additional columns: one with the number of correct translations; and the other one with the number of additional correct translations generated by the application of that substitution.

3.2 Evaluation 1: Gold Standard

The rules were defined with a gold standard dictionary that was used to evaluate the substitutions relevancy, and the better sequence to use. For that purpose we downloaded a Portuguese–Galician translation dictionary from the Apertium project [4]. All multi-word sequences were removed, and a spell checker was used in the Portuguese portion of the dictionary to detect words written in the Brazil orthography (that were manually rewritten to the European Portuguese orthography), words that were written according to the Orthographic Agreement of 1990 (the dictionary to align uses orthography before 1990), and some other wrong words were also fixed.

After this cleaning process, the dictionary counts 9 224 pairs. Note that each pair maps a Portuguese word to a set of possible Galician translations. Table 3 presents the results. Each line refers to a different run, adding a new rule to the rule set. The first line, labeled as ID, corresponds to the first run, without any substitution. Looking into the accuracy for that line, one can see that 58% of the Portuguese words in the dictionary do not need translation, as they are shared across languages. In the second line the substitution A is activated ($ss > s$), leading to more 163 correct translations. For the third run, and before substitution B is ran, the system performs the substitution A. This means that each row includes the previous substitutions, and this explains the relevance of the delta column, which shows the number of accepted translations that each substitution generates[3]

There are some rules with a small delta, like rule Z. Nevertheless, the suffix *-oide* is specific of technical terms. The small dictionary used for this specific evaluation does not cover technical terms (with few exceptions), and we expect the rule to be more productive with a bigger dictionary.

At the end of the experiment we were able to keep the precision above 99.5% (higher than the obtained without any substitution) and a recall of 79.5% (compared with the 58.6% obtained without substitutions) resulting in a slight good F_1 measure.

[3] In fact, not exactly, as words might need more than one substitution to be correct.

Table 2 Hypothesis Type I and type II error matrix.

$\mathcal{T}(L_{gl}, w_{pt}) = w_{gl}$	Correct	Incorrect
w_{gl} is a Galician word	TP	FP
w_{gl} is not a Galician word	TN	FN

Table 3 Given 9 226 pairs mapping Portuguese words to a set of possible Galician words, the table presents precision, recall and F_1 measure; accuracy, total of correct words, and delta of correct words from last run. Note that substitutions are cumulative (meaning that when substitution B is performed, substitution A was performed before).

Subst. Id.	Precision	Recall	F_1	Accuracy	Correct	Δ
ID	0.9954	0.5859	0.7376	0.5843	5390	5390
A	0.9952	0.6038	0.7516	0.6020	5553	163
B	0.9951	0.6158	0.7608	0.6139	5663	110
C	0.9952	0.6567	0.7912	0.6546	6038	375
D	0.9951	0.6687	0.7999	0.6665	6148	110
E	0.9952	0.6782	0.8066	0.6760	6235	87
F	0.9952	0.6786	0.8070	0.6764	6239	4
G	0.9953	0.6838	0.8107	0.6816	6287	48
H	0.9953	0.6927	0.8169	0.6905	6369	82
I	0.9953	0.6934	0.8174	0.6911	6375	6
J	0.9953	0.6964	0.8195	0.6942	6403	28
K	0.9955	0.7210	0.8363	0.7187	6629	226
L	0.9955	0.7256	0.8394	0.7232	6671	42
M	0.9955	0.7284	0.8413	0.7260	6697	26
N	0.9957	0.7482	0.8544	0.7458	6879	182
O	0.9957	0.7496	0.8553	0.7472	6892	13
P	0.9957	0.7515	0.8565	0.7490	6909	17
Q	0.9957	0.7588	0.8612	0.7563	6976	67
R	0.9957	0.7602	0.8621	0.7577	6989	13
S	0.9958	0.7680	0.8672	0.7655	7061	72
T	0.9958	0.7703	0.8686	0.7678	7082	21
U	0.9958	0.7772	0.8731	0.7747	7146	64
V	0.9958	0.7780	0.8735	0.7755	7153	7
W	0.9958	0.7783	0.8737	0.7758	7156	3
X	0.9958	0.7796	0.8746	0.7771	7168	12
Y	0.9958	0.7806	0.8752	0.7781	7177	9
Z	0.9958	0.7807	0.8753	0.7782	7178	1
AA	0.9958	0.7813	0.8756	0.7787	7183	5
AB	0.9958	0.7818	0.8759	0.7793	7188	5
AC	0.9958	0.7822	0.8762	0.7797	7192	4
AD	0.9959	0.7836	0.8770	0.7810	7204	12
AE	0.9959	0.7855	0.8783	0.7830	7222	18
AF	0.9959	0.7863	0.8787	0.7837	7229	7
AG	0.9957	0.7876	0.8795	0.7849	7240	11
AH	0.9957	0.7882	0.8799	0.7856	7246	6
AI	0.9958	0.7903	0.8812	0.7876	7265	19
AJ	0.9956	0.7928	0.8827	0.7900	7287	22
AK	0.9956	0.7940	0.8834	0.7912	7298	11
AL	0.9956	0.7947	0.8839	0.7920	7305	7
AM	0.9956	0.7951	0.8842	0.7924	7309	4
AN	0.9956	0.7955	0.8844	0.7927	7312	3

3.3 Evaluation 2: Triangulated Dictionary

The dictionary used in the previous section is not a large dictionary. When trying to evaluate the translation algorithm in a bigger bilingual dictionary we hit a wall: the scarcity of free Portuguese–Galician dictionaries.

To solve this issue we performed triangulation with different dictionaries:

- Using the Portuguese–Spanish (12 340 pairs) and the Spanish–Galician (7 581 pairs) bilingual dictionaries from the Apertium translation software, resulting in a Portuguese–Galician bilingual dictionary with 5 045 pairs;
- Using the Portuguese–Spanish (12 340 pairs) and the Spanish–English (24 912 pairs) bilingual dictionaries from the Apertium translation software, and an English–Galician (17 626 pairs) bilingual dictionary from the CLUVI project [6], resulting in a Portuguese–Galician bilingual dictionary with 6 644 pairs;
- Using the Portuguese–English (14 600 pairs) from a merchandising application offered years ago by a beverages make, and the English–Galician (17 626 pairs) bilingual dictionary from CLUVI project, resulting in a Portuguese–Galician bilingual dictionary with 8 589 pairs.

These three dictionaries obtained, and the original Portuguese–Galician dictionary used in the previous section, were added together, resulting in a 14 492 pairs bilingual dictionary (5 268 more pairs than the original dictionary).

Before presenting the results, a brief explanation of how the triangulation process was performed, and how the dictionaries were merged together is in order:

- **Triangulation:** Each one of the dictionaries used in any of the triangulation processes contains lists of pairs, mapping words from the source language to a list of words in the target language. Therefore, the process needs two source dictionaries $\mathcal{D}_1 : L_S \mapsto \mathcal{P}(L_I)$ and $\mathcal{D}_2 : L_I \mapsto \mathcal{P}(L_T)$. For each word in the source language S we feed each possible translation (language I) to the second dictionary, obtaining a set of possible translations in our target language (language T): $\mathcal{D}_1 \circ \mathcal{D}_2 : L_S \mapsto \mathcal{P}(L_T)$. Note that this composition is defined as the composition of $\mathcal{D}_2$ for each word w_I that results from applying $\mathcal{D}_1$ to a specific source words w_S.
- **Addition:** The addition of two dictionaries $\mathcal{D}_1 : L_S \mapsto \mathcal{P}(L_T)$ and $\mathcal{D}_2 : L_S \mapsto \mathcal{P}(L_T)$ results in a dictionary $\mathcal{D}_{1+2} : L_S \mapsto \mathcal{P}(L_T)$ where, for each word w_S from the source language, we compute the union the the possible translations from each dictionary.

Using the same substitution process as described earlier, we obtain the results presented in table 4. With this bigger dictionary, with possibly more errors, we get some more words that maintain orthography between languages, but also more words where substitutions produce valid words. The precision drops from the previous 99% to 96.6% (still above 95%), and the recall from the nearly 80% from the previous evaluation to 68.9%. The F_1 measure keeps above 0.80.

4 Dictionary Alignment

As explained before, our main goal is the alignment of entries from *Dicionário-Aberto* (DA) with the revised edition of the *Diccionario de Sinónimos da Lingua Galega* (DSLG).

One problem with this process is that DA uses an old Portuguese orthography, but some work has already been initiated to modernize its language. Although the Portuguese orthography is changing again (with the late adoption of an Orthography Agreement from 1990 [3]), the process of modernization is being performed to the orthography used before 1990. The main reason is that it is easy to migrate it to the current orthography [1], but the inverse

Table 4 Given 14 492 pairs mapping Portuguese words to a set of possible Galician words, the table presents precision, recall and F_1 measure; accuracy, total of correct words, and delta of correct words from last run. Note that substitutions are cumulative (meaning that when substitution B is performed, substitution A was performed before).

Subst. Id.	Precision	Recall	F_1	Accuracy	Correct	Δ
ID	0.9668	0.5022	0.6611	0.4937	7155	7155
A	0.9664	0.5176	0.6741	0.5084	7368	213
B	0.9663	0.5275	0.6824	0.5179	7506	138
C	0.9668	0.5646	0.7129	0.5538	8026	520
D	0.9661	0.5746	0.7206	0.5633	8163	137
E	0.9658	0.5831	0.7272	0.5713	8279	116
F	0.9658	0.5834	0.7274	0.5716	8283	4
G	0.9656	0.5875	0.7305	0.5754	8339	56
H	0.9648	0.5953	0.7363	0.5827	8444	105
I	0.9648	0.5958	0.7367	0.5831	8451	7
J	0.9649	0.5986	0.7388	0.5858	8490	39
K	0.9654	0.6204	0.7554	0.6069	8795	305
L	0.9656	0.6274	0.7606	0.6136	8893	98
M	0.9656	0.6311	0.7633	0.6172	8944	51
N	0.9662	0.6439	0.7728	0.6297	9126	182
O	0.9661	0.6451	0.7736	0.6308	9142	16
P	0.9662	0.6470	0.7750	0.6327	9169	27
Q	0.9663	0.6542	0.7802	0.6396	9269	100
R	0.9663	0.6556	0.7812	0.6410	9289	20
S	0.9662	0.6631	0.7865	0.6481	9392	103
T	0.9661	0.6657	0.7882	0.6505	9427	35
U	0.9662	0.6719	0.7926	0.6565	9514	87
V	0.9661	0.6730	0.7934	0.6575	9529	15
W	0.9662	0.6735	0.7937	0.6579	9535	6
X	0.9660	0.6746	0.7944	0.6590	9550	15
Y	0.9659	0.6757	0.7951	0.6600	9564	14
Z	0.9659	0.6759	0.7952	0.6601	9566	2
AA	0.9659	0.6762	0.7955	0.6604	9571	5
AB	0.9659	0.6768	0.7959	0.6610	9579	8
AC	0.9659	0.6771	0.7961	0.6613	9584	5
AD	0.9660	0.6781	0.7968	0.6623	9598	14
AE	0.9660	0.6797	0.7979	0.6638	9620	22
AF	0.9660	0.6804	0.7984	0.6644	9629	9
AG	0.9659	0.6814	0.7991	0.6654	9643	14
AH	0.9660	0.6819	0.7994	0.6659	9650	7
AI	0.9661	0.6841	0.8010	0.6681	9682	32
AJ	0.9660	0.6863	0.8025	0.6701	9711	29
AK	0.9660	0.6873	0.8032	0.6711	9726	15
AL	0.9661	0.6881	0.8037	0.6718	9736	10
AM	0.9660	0.6884	0.8039	0.6721	9740	4
AN	0.9660	0.6887	0.8041	0.6724	9744	4

Table 5 Substitution used, the number of words from the Portuguese dictionary with translation (and corresponding percentage), the number of words from the Galician dictionary used as translations (and the corresponding percentage).

	Portuguese Words		Galician Words	
Substitution	Count	Percentage	Count	Percentage
ID	12711	15.3502%	12711	33.7475%
A	13082	15.7982%	13065	34.6874%
B	13447	16.2390%	13421	35.6326%
C	14348	17.3270%	14321	38.0220%
D	14764	17.8294%	14728	39.1026%
E	15174	18.3245%	15138	40.1912%
F	15179	18.3306%	15143	40.2044%
G	15311	18.4900%	15263	40.5230%
H	15856	19.1481%	15807	41.9673%
I	15874	19.1699%	15820	42.0019%
J	15953	19.2653%	15899	42.2116%
K	16365	19.7628%	16306	43.2922%
L	16571	20.0116%	16512	43.8391%
M	16683	20.1468%	16624	44.1365%
N	16716	20.1867%	16657	44.2241%
O	16752	20.2302%	16693	44.3197%
P	16797	20.2845%	16738	44.4391%
Q	16969	20.4922%	16910	44.8958%
R	17003	20.5333%	16944	44.9861%
S	17150	20.7108%	17091	45.3763%
T	17237	20.8159%	17178	45.6073%
U	17359	20.9632%	17300	45.9312%
V	17420	21.0369%	17361	46.0932%
W	17436	21.0562%	17377	46.1357%
X	17469	21.0960%	17410	46.2233%
Y	17505	21.1395%	17445	46.3162%
Z	17505	21.1395%	17445	46.3162%
AA	17511	21.1468%	17451	46.3321%
AB	17521	21.1588%	17461	46.3587%
AC	17524	21.1625%	17464	46.3667%
AD	17564	21.2108%	17504	46.4729%
AE	17586	21.2373%	17526	46.5313%
AF	17596	21.2494%	17536	46.5578%
AG	17647	21.3110%	17564	46.6322%
AH	17669	21.3376%	17584	46.6853%
AI	17712	21.3895%	17627	46.7994%
AJ	17740	21.4233%	17648	46.8552%
AK	17765	21.4535%	17673	46.9215%
AL	17784	21.4764%	17693	46.9746%
AM	17813	21.5115%	17718	47.0410%
AN	17817	21.5163%	17722	47.0516%
DIC	20084	24.2540%	19989	53.0705%

process is not injective. Also, the rules used to translate Portuguese words into Galician words take advantage of the orthography before 1990: they would be harder to write for modern Portuguese as it is more ambiguous.

DA has more than 128 000 entries, but as we are also maintaining words in the old orthography, the number of real different words is lower. Also, as the modernization process is not 100% accurate, and to remove some extra error from this process, we used the *Vocabulário Ortográfico do Português*[4] [2] (VOP) to filter what words to align. The removal of duplicate entries (like *pharmácia* and *farmácia*, where only the latter should be used) results in about 110 000 different entries. The VOP lexicon includes more than 155 000 different words. The intersection of these two lexicons includes 82 807 entries. These are the entries we are trying to align at this moment. Regarding the DSLG lexicon, it has 24 571 entries (41 923 meanings or groups of synonyms), totalling 37 665 unique words (entries or synonyms).

Table 5 presents the results of the alignment process. Although the difference between pure string matching (identity function) and the use of substitutions is not huge if we look into percentages, the truth is that the use of substitutions was able to align about five thousand words, and almost half of the Galician dictionary was used as a translation. The final line of the table (identified as `DIC`) is the result of using the substitutions and, for those words that after being translated do not exist in the target lexicon, using the dictionary used in the second evaluation we performed (section 3.3), resulting in a few more than two thousand words recognized.

The big difference between these two dictionaries, and the fact of their being dictionaries (and therefore including a lot of unfrequent words) explain the low percentage of success. Nevertheless, further research should be done in order to understand how the substitutions set can be made better for bigger result sets.

5 Final Remarks

In this paper we present an approach to translate Portuguese words in a dictionary into Galician words using a set of string substitutions. Although the approach is unable to translate all words (and that was never our goal), it can be used to translate a reasonable amount of Portuguese words with a decent precision value.

Nevertheless, we deliberately ignored a relevant problem: false friends. These are words that have the same or similar writing in Portuguese and Galician, but have different meanings. There are mainly two different situations:

- two words that share a subset of the meanings. For instance, *talho* (PT) and *tallo* (GL) share the majority of their senses, but there are some of them that are specific to Portuguese (for example, the place where meat is sold);
- two words that have complete different meanings. An example would be the word *presunto* (written in the same way in the two languages) that means *ham* in Portuguese (a noun), but means *alleged* in Galician (an adjective);

In the first case the alignment between the two entries should be kept. But the second case is completely wrong, and should probably be removed from the alignments. In order to do that, a list of false friends would be needed, or some kind of heuristic to detect the semantic distance between the dictionary entries. In any case, this research direction should be followed in the near future in order to guarantee a high quality level in the dictionary alignment results.

[4] Portuguese Orthographic Vocabulary

This work should be extended in two different directions: first, researching the results obtained, to understand how a larger percentage of alignments can be achieved (and evaluating the alignment quality); second, analysing how incorporating the Galician dictionary into Dicionário-Aberto can result in a better user experience. For instance, Dicionário-Aberto includes a navigation ontology (relations between concepts are extracted and presented to the user as a navigation feature). It might be possible to use the alignment between the two dictionaries to obtain better concept relations, and therefore a more complete navigation ontology.

Acknowledgments This work was partially supported by Grant TIN2012-38584-C06-04, supported by the Ministry of Economy and Competitiveness of the Spanish Government on "*Adquisición de escenarios de conocimiento a través de la lectura de textos: Desarrollo y aplicación de recursos para el procesamiento lingüístico del gallego (SKATeR-UVIGO)*"; and by the Xunta de Galicia through the "*Rede de Lexicografía (Relex)*" (Grant CN 2012/290) and the "*Rede de Tecnoloxías e análise dos datos lingüísticos*" (Grant CN 2012/179).

References

1 José João Almeida, André Santos, and Alberto Simões. Bigorna – a toolkit for orthography migration challenges. In *Seventh International Conference on Language Resources and Evaluation (LREC2010)*, Valletta, Malta, may 2010.

2 Margarita Correia (coord.). Vocabulário Ortográfico do Português, 2010. Lisbon: ILTEC/-Portal da Língua Portuguesa.

3 Diário da República. Acordo ortográfico da língua portuguesa, 1990. Technical Report 193, série I-A, 23 de Agosto, 1991. `http://www.portaldalinguaportuguesa.org/index.php?action=acordo&version=1990`.

4 Mikel Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, pages 1–18, July 2011.

5 Xavier Gómez Guinovart, Xosé María Gómez Clemente, Andrea González Pereira, and Verónica Taboada Lorenzo. Galnet: WordNet 3.0 do galego. *Linguamática*, 3(1):61–67, 2011.

6 Xavier Gómez Guinovart, Alberto Álvarez Lugrís, and Eva Díaz Rodríguez. *Dicionario moderno inglés-galego*. 2.0 Editora, Ames, 2012.

7 Camiño Noia Campos, Xosé María Gómez Clemente, and Pedro Benavente Jareño. *Diccionario de sinónimos da lingua galega*. Galaxia, Vigo, 1997.

8 Alberto Simões and José João Almeida. Processing XML: a rewriting system approach. In Alberto Simões, Daniela da Cruz, and José Carlos Ramalho, editors, *XATA 2010 — 8ª Conferência Nacional em XML, Aplicações e Tecnologias Aplicadas*, pages 27–38, Vila do Conde, Maio 2010.

9 Alberto Simões, José João Almeida, and Rita Farinha. Processing and extracting data from Dicionário Aberto. In Nicoletta Calzolari et al., editor, *Seventh International Conference on Language Resources and Evaluation (LREC2010)*, pages 2600–2605, Valletta, Malta, may 2010. European Language Resources Association (ELRA).

10 Alberto Simões and Rita Farinha. Dicionário Aberto: Um novo recurso para PLN. *Vice-Versa*, 16:159–171, December 2011.

11 Alberto Simões, Álvaro Iriarte Sanromán, and José João Almeida. Dicionário-aberto – a source of resources for the portuguese language processing. *Computational Processing of the Portuguese Language, Lecture Notes for Artificial Intelligence*, 7243:121–127, April 2012.

Combining Language Independent Part-of-Speech Tagging Tools

György Orosz[1], László János Laki[1], Attila Novák[1], and Borbála Siklósi[2]

1 **MTA-PPKE Language Technology Research Group –**
Pázmány Péter Catholic University, Faculty of Information Technology
50/a Práter street, Budapest, Hungary
`{oroszgy, laki.laszlo, novak.attila}@itk.ppke.hu`

2 **Pázmány Péter Catholic University, Faculty of Information Technology**
50/a Práter street, Budapest, Hungary
`siklosi.borbala@itk.ppke.hu`

Abstract

Part-of-speech tagging is a fundamental task of natural language processing. For languages with a very rich agglutinating morphology, generic PoS tagging algorithms do not yield very high accuracy due to data sparseness issues. Though integrating a morphological analyzer can efficiently solve this problem, this is a resource-intensive solution. In this paper we show a method of combining language independent statistical solutions – including a statistical machine translation tool – of PoS-tagging to effectively boost tagging accuracy. Our experiments show that, using the same training set, our combination of language independent tools yield an accuracy that approaches that of a language dependent system with an integrated morphological analyzer.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases part-of-speech tagging, combination, agglutinative languages, machine learning, machine translation

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.249

1 Introduction

Part-of-speech tagging is one of the basic and most studied tasks of computational linguistics. There are several freely available language independent solutions which are usually based on statistical methods. The robust and accurate operation of these tools is crucial, since they are usually one of the first components of any linguistic processing chain. Thus errors propagating from this level affect the result of systems performing more complex language processing tasks.

In our present work, we describe a method of combining two independent tools: the HMM-based PurePos [14] and the HuLaPos PoS tagger [12] based on the Moses decoder [11]. Deeper investigation of incorrectly classified words has reflected that the overlap between the errors made by each of these systems is very small. Inspired by this observation, we experimented with possibilities of combining the knowledge of these systems. We prove that using the combination of the two language independent systems yields a better result than using a simple majority voting of three tools by extending the investigation to a third system as well. Our results also show that for Hungarian, the tagging accuracy of the presented language independent method approaches that of the augmented version of PurePos that employs a language dependent morphological analyzer.

2 Tools

In this article, we explore ways of combining the following tools:

1. **PurePos** is an open source hybrid system for full morphological disambiguation: it is capable of not just selecting the most probable tag for a token, but assigning a lemma as well. It is based on hidden Markov models, but it can use an integrated morphological analyzer module as well to tag unseen words and to assign lemmas. The tool is based on algorithms described by Brants [3] and Halácsy [9], but what distinguishes it from them is the complete integration of a morphological analyzer. Its extremely short training time is due to the usage of a simple smoothed trigram model while some tweaks in the implementation result in a high precision. It is implemented in Java, thus it can be easily extended and is portable. Integration of a morphology results in a further boost in its PoS tagging accuracy and also makes lemmatization possible.
2. **HuLaPos**: is a morphological annotation tool based on an SMT decoder(HuLaPos [12]). It is possible to create a near state-of-the-art system that we created with minimal preprocessing and – compared to corpus sizes needed for SMT – a relatively small training set. Another advantage of the SMT translation process applied for PoS tagging is that it is able to consider the context of a word in both directions. Its only weakness is that it is not capable of tagging out-of-vocabulary (OOV) words not represented in the training corpus. However with smoothing based on the distribution of rare words, the error rate of tagging of OOV words was decreased.
3. **OpenNLP**: In our experiments, we also used the **maximum entropy** and **perceptron learning** algorithms implemented in the OpenNLP toolkit[1] [2]. These are very popular annotation methods, since the feature sets used for training can be easily adapted for new tasks. However the main drawback of these algorithms is that their training time is extremely high compared to HMM based models. They were employed in our tests with their default feature set.

We have seen in the case of the PurePos system that morphological knowledge is very useful, especially in the case of agglutinating languages (such as Hungarian), but a morphological analyzer is often not available and it is very time consuming to build one and it requires the involvement of expert linguists.

3 Motivation for Tagger Combination

We investigated the tagging performance (table 1) and errors (figure 1) of the above described four systems and found, that though the accuracy of PurePos is higher relative to the others, its errors overlap only slightly with those of the HuLaPos system. There are also cases where words mistagged by PurePos and HuLaPos are correctly tagged by one of the two modules of OpenNLP. However, the error sets of the maxent and perceptron learning algorithms are very similar to each other.

We performed our experiments on a modified version of the Hungarian Szeged Corpus [6], in which PoS annotation was automatically converted to morphosyntactic tags used by the Hungarian HuMor morphological analyzer [13, 15]. It was done for the purpose of comparing our results with an available state-of-the-art hybrid disambiguator. The taggers described above use the same rich tagset that is available in the training corpus and provided by the

[1] Henceforward PE denotes the preceptron learning method while ME denotes the maximum entropy learning method of the toolkit.

Table 1 Tagging precision of the baseline systems.

PoS tagger	Precision
PurePos	97.85%
HuLaPos	97.57%
OpenNLP perceptron	93.86%
OpenNLP maxent	93.03%

analyzer. In the case of the training data this amounts to more than a thousand different tags. 10% of the corpus was separated for testing and another 10% of the corpus is used for development and tuning purposes. Each set contains about 7100 sentences, while the rest, about 57000 sentences, were used for training of the systems.

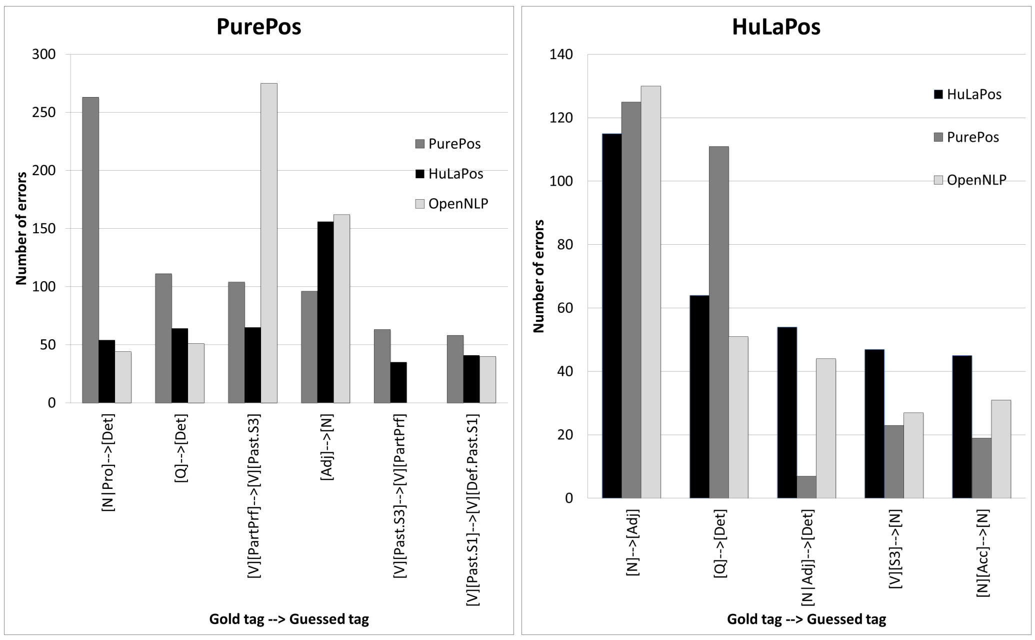


Figure 1 Comparing the most frequent errors of PurePos and HuLaPos.

Performing a deep error analysis, we collected the most frequent error types that together represent 30% of all errors of each tool. From figure 1. one can conclude, that there are some error types, that are specific to PurePos. These are the following: mistagging of demonstrative pronouns as definite articles (`az[N|Pro]` 'that' vs. `az[Det]` 'the') and the numeral `egy[Q]` 'one' as the indefinite article `egy[Det]` 'a'). There is a significant difference between the performance of tagging past participles (`[V][PartPrf]`): these are mistagged as simple past verb forms (`[V][Past.S3]`) by PurePos more often than by HuLaPos.

On the other hand, the SMT system very often assigns wrong tags to word forms not seen in the training set while it performs better for words that were seen. Besides, some typical errors of HuLaPos are the following: mistagging adjectives (`[Adj]`) as nouns (`[N]`) and assigning a nominative noun tag (`[N]`) to verbs (`[V][S3]`) and accusative nouns (`[N][ACC]`).

While there are some cases where the perceptron learning method could guess the right label while both HuLaPos and PurePos missed it (such as the verbal tag `[V][Past.S3]`),

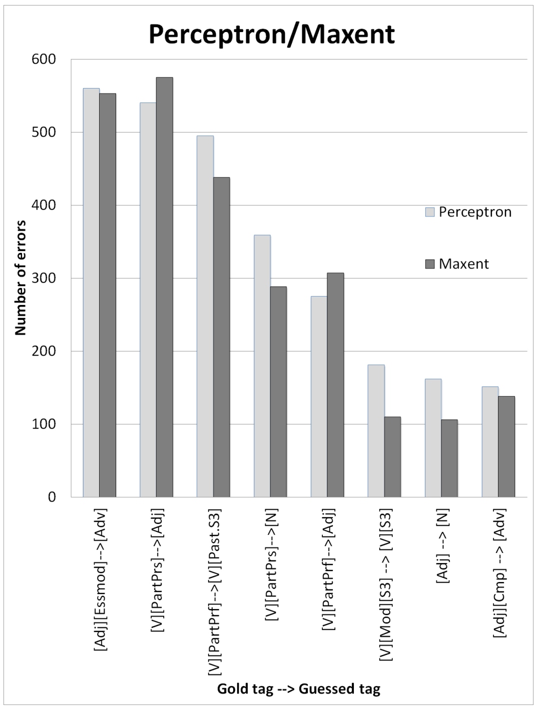


Figure 2 Comparing the ME and PE methods error rate.

its overall tagging precision is significantly lower. Moreover, comparing the ME and PE methods (see figure 2) we can conclude that their errors are mostly overlapping. This is due to the fact that both their feature sets and the decoding algorithm are the same, only the training algorithm differs.

Table 2 The maximal knowledge of tagger combinations.

Name	Precision
Max2: PurePos + HuLaPos	98.84%
Max3: PurePos + HuLaPos + PE	99.21%
Max4: PurePos + HuLaPos + PE + ME	99.27%

Hypothetical maximum Relying on the above error analysis we can calculate what performance a hypothetical combination algorithm could reach that always succeeds in selecting the best of all tags proposed by the taggers to be combined. This oracle tagger was simulated by presuming that one could always decide which tagger to trust. Thus the combination of two (Max2), three (Max4) or four (Max4) tools may perform significantly better than each of the individual ones, if one manages to combine them right. The ideal combined tagger that could aggregate the knowledge of the PurePos, HuLaPos and the PE systems, could lower the error rate of the best one with almost 66% percent. One that combines the best two: PurePos and HuLaPos only, could also achieve an almost of 46% error rate reduction. While mining the knowledge of all four systems could in theory result in the best tagger, since the two methods derived from OpenNLP highly correlate, we skip the poorly performing maxent tagger in this investigation.

In the rest of this paper, we focus only on the combination of the two or three best performing systems.

4 Tagger Combinations

The task of combining several classification systems is traditionally composed of two subtasks. First, one has to select the appropriate features that may be used, then one must select an appropriate combining algorithm. (In data mining, this procedure is commonly referred to as stacking learners.) Although part-of-speech tagging is a classification task [16], where the tagger assigns the most probable tag to each token in the sentence, it is not done in a token by token manner, rather sentence by sentence. Consequently, the individual tagging events are not independent. Most of the statistical tagging algorithms heavily rely on this fact: finding the most probable tag sequence for the sentence instead of individually disambiguating the morphological class of each token [14, 3, 9, 16]. Considering this, one could create a sentence- or a token-based combination system. A sentence-based solution would select the proper tagger for each sentence, while a token-based one does the same for each token. The former is a feasible method of combining MT (machine translation) systems, but for taggers, we opted for token-based combination.

4.1 Related Works

Creating and applying part-of-speech taggers has a long history starting from rule-based systems to applying machine learning methods. One of the first attempts of combining such methods was done by Brill and Wu [4]. They propose an instance based learning system for tagging a token that employs contextual clues such as the surrounding words and their suggested tags. Hajič et al. use a series of tagger with a rule-based approach [8] to combine disambiguators in order to improve overall tagging accuracy. A comprehensive study was presented by Halteren et al. [10] in which a detailed overview about previous combination attempts is given mainly using machine learning techniques. They also present several combination methods and systematically compare and evaluate them. For the optimal usage of the training corpus cross-validation is used to train the second-level classifier. All of these works conclude that the "combination of several different learning systems enables to raise the performance ceiling".

4.2 Voting

As a baseline system, we implemented a standard token–based unweighted voting scheme. As we saw in section 4, the errors of the three best-performing systems differ significantly, but the error distributions of the two methods in OpenNLP seem to be correlated. That is why we only took the three of the best performing systems for this stacking, namely: PurePos, HuLaPos and PE. The tags are calculated as follows:

1. tag the sentence with all systems,
2. for each token choose the tag that has the most votes ,
3. if there is no such, take the one proposed by PurePos.

Applying this scheme to the development set, it (Comb3) increases the overall accuracy to 98.18%, that is a 15.35% error reduction rate. Applying the same simple voting scheme to all of the four available taggers (Comb4) results in inferior performance compared to Comb3. The reason for this is that typical errors of the ME and PE systems co-occur and, and they can together win the vote with a wrong suggestion.

Compared to PurePos, this growth is significant (see table 3), but this combination yields just 24.26% of the hypothetical maximum error reduction rate.

Table 3 Comparing the accuracy of the simple voting scheme.

Tagger name	Precision
Comb3	98.18%
Comb4	98.10%
PurePos	97.85%
Max4	99.21%

4.3 Stacking

In our further combination experiments, we took into account the possibilities of stacking only the two best performing systems. As previously, we used the token based combination approach. A commonly used method in the area of stacking is to use a metalearner that is built upon various algorithms that solve the same task using significantly different methods. It learns which classifier to trust in various contexts, thus discovers the best way to combine their output. The models learnt by the actual systems applied on the task to be solved are usually called level-0 models, while the one that is learnt by the combiner is called level-1 model. This examination implies the following questions:

1. Using a fixed size corpus, what is the best way to use all the knowledge in the data?
2. What sort of combining algorithms perform the best?
3. Given a combiner, what is the most informative feature combination?

In data mining tasks, it is usual to use level-0 attributes for the level-1 classifier [17], but in our case that is hard to apply, since each tagger we use has a different kind of feature sets. We applied features that are commonly used in taggers, and added some more, that are specific for our task:

- the word to be tagged and words that precede or follow it
- guessed tags from PurePos and HuLaPos for the actual word, the previous word, the next word, the second previous word, the second word on the right,
- at most ten long suffixes of the word
- whether the word contains a hyphen,
- whether the word contains a dot,
- whether the word starts with an uppercase letter.

In the case of nominal attributes[2], an additional `<none>` value is needed to which nominal attributes of words never seen in the level-1 training data are mapped. We use the same solution for the output tag feature, because in the case of an agglutinating language like Hungarian, all elements of the morphological tag set (over 1000 different tags in our case) cannot be expected to be present in the training corpus. While the predicted level-1 class label generally could in theory be either the correct tag or the name of the preferred system, we chose the latter approach, since, due to the huge number of possible PoS tags, the training data for a system using the former approach would be extremely sparse.

David Wolpert, the inventor of stacking, proposed to use a “relatively global, smooth” level-1 learner, thus we investigated the following classifiers, which in addition to being simple, were shown to be able to handle nominal attributes: Naïve Bayes [5] , IB1 and IBk [1]. Since our features are clearly not independent, Naïve Bayes is not expected to perform very well, thus we used it as another baseline. The instance based methods fit to our model rather

[2] In data mining terminology nominal attributes are ones, that have fixed set of possible values.

well since, in the case of nominal attributes (and that is indeed what we have), the distance function is the square root of the number of attributes that are the same, thus providing a simple but efficient classification rule[3].

Training with Cross-Validation

For the best utilization of the corpus, we applied training with cross-validation. We split the training set into 5 equal sized parts and trained level-0 taggers (PurePos, HuLaPos) five times using 4/5 of the corpus, and the rest was annotated by both taggers in each round. The union of these annotated parts was used for training the level-1 classifier. Thus the full training data was available for level-1 training, yet separating the two phases of the training process. In addition, this workflow made it possible that level-0 taggers also be trained on the full training data in the end.

Table 4 Tagging precision of the combined tagger methods.

Combination	Precision
IBk, k=1	98.32%
IB1	98.30%
Naïve Bayes	98.26%

Evaluating the combination methods on the development set (table 4), we can state that the best results were obtained unsing instance based learning (IB), more specifically the algorithm called IBk in the WEKA framework[4] [7], with the k parameter set to 1. The aggregation of the two best performing classifiers results in a significantly better accuracy than the simple voting scheme of 3 or 4 systems, with the IB learning algorithms beating all the others as expected.

5 Evaluation

Table 5 Evaluating the tagger combinations on the test set.

Name	Precision
PurePos	97.89%
Morphologically augmented PurePos	**98.57%**
Simple voting of 3	98.19%
Combination with Naïve Bayes	98.28%
Combination with IB1	98.36%
Combination with IBk, k=1	**98.39%**
Maximal knowledge of Purepos and HuLaPos	98.86%

Choosing the best performing IBk combination, we compared (5. table) its performance on the held out test set with that of the baseline systems and the hypothetical best combination. While the simple voting scheme yielded only 30.93% and Naïve Bayes 40.21%, IBk reached

[3] Other machine learning algorithms – such as C4.5 – were also considered to be involved, but unfortunately they were not able to handle the large amount of data and the huge number of discrete features that were provided during the experiments.
[4] Weka is a collection of machine learning algorithms for data mining tasks.

51.55% of the hypothetical maximum improvement. The best performing IBk system produces only 14.18% more errors than the morphologically augmented PurePos system, which includes a language specific symbolic component. Our PoS combination results are in accordance with the observations made by Halteren et al. [10] that stacking performance is significantly better than voting. Because there is only a little overlap between the errors made by the taggers, we could achieve a significant error rate reduction by combining only two taggers.

We are not aware of any previous work exploiting the strengths of an SMT system, thus on of the achievements of this work is on discovering the possible supplementary usage of HuLaPos in such a task. Beside of this the feature set proposed by Brill [4] was also extended to be able to perform well with agglutinative languages such as Hungarian.

6 Conclusion

In this paper, we presented a combination of two PoS taggers that is able to decrease the error rate of the better tagger by 23.70%. One of this tools was a machine translation system, that were discovered to greatly complement the HMM one. While the combination uses a machine learning algorithm, we presented a way of using the whole training data for training the level-1 and level-0 models at the same time. The performance of the combined system approximates that of a morphologically augmented one, which heavily relies on language dependent linguistic knowledge. The presented method could be used as a language independent high precision PoS tagging tool. Since our results are promising we are planning to extend the investigation of our PoS combining technique to full morphological disambiguation[5] and other languages as well.

References

1 David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, 1991.

2 Jason Baldridge, Thomas Morton, and Gann Bierner. The OpenNLP maximum entropy package, 2002.

3 Thorsten Brants. TnT - A Statistical Part-of-Speech Tagger. In *Proceedings of the sixth conference on Applied natural language processing*, number i, pages 224–231. Universitsität des Saarlandes, Computational Linguistics, Association for Computational Linguistics, 2000.

4 Eric Brill and Jun Wu. Classifier combination for improved lexical disambiguation. In *Proceedings of the 17th international conference on Computational linguistics-Volume 1*, pages 191–195. Association for Computational Linguistics, 1998.

5 William B. Cavnar and John M. Trenkle. N-Gram-Based Text Categorization. *Ann Arbor MI*, 48113(2):161–175, 1994.

6 Dóra Csendes, János Csirik, and Tibor Gyimóthy. The Szeged Corpus: A POS tagged and syntactically annotated Hungarian natural language corpus. In *Proceedings of the 5th International Workshop on Linguistically Interpreted Corpora LINC 2004 at The 20th International Conference on Computational Linguistics COLING 2004*, pages 19–23, 2004.

7 Eibe Frank, Mark Hall, Geoffrey Holmes, Richard Kirkby, Bernhard Pfahringer, Ian H Witten, and Len Trigg. Weka – a machine learning workbench for data mining. *Data Mining and Knowledge Discovery Handbook*, pages 1269–1277, 2010.

[5] There is also a need of high precision disambiguation between lemmas especially in the case of agglutinating languages.

8 Jan Hajič, Pavel Krbec, Pavel Květoň, Karel Oliva, and Vladimír Petkevič. Serial combination of rules and statistics: A case study in czech tagging. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 268–275. Association for Computational Linguistics, 2001.

9 Péter Halácsy, András Kornai, and Csaba Oravecz. HunPos: an open source trigram tagger. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, pages 209–212, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

10 Hans Van Halteren, Jakub Zavrel, and Walter Daelemans. Improving Accuracy in Word Class Tagging through the Combination of Machine Learning Systems. *Computational Linguistics*, 27(2):199–229, 2001.

11 Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In Annie Zaenen and Antal Van Den Bosch, editors, *Computational Linguistics*, volume 45 of *ACL '07*, pages 177–180. Association for Computational Linguistics, Association for Computational Linguistics, 2007.

12 László János Laki. Investigating the Possibilities of Using SMT for Text Annotation. In *SLATE 2012 - Symposium on Languages, Applications and Technologies*, pages 267–283, Braga, Portugal, 2012. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

13 Attila Novák. Milyen a jó humor? In *Magyar Számítógépes Nyelvészeti Konferencia 2003*, pages 138–145., Szeged, 2003.

14 György Orosz and Attila Novák. PurePos – an open source morphological disambiguator. In Bernadette Sharp and Michael Zock, editors, *Proceedings of the 9th International Workshop on Natural Language Processing and Cognitive Science*, pages 53–63, Wroclaw, 2012.

15 Gábor Prószéky and Attila Novák. Computational Morphologies for Small Uralic Languages. In *Inquiries into Words, Constraints and Contexts.*, pages 150–157, Stanford, California, 2005.

16 Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Proceedings of the conference on empirical methods in natural language processing*, volume 1, pages 133–142, 1996.

17 Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* 3rd edition, 2011.

Comparing Different Machine Learning Approaches for Disfluency Structure Detection in a Corpus of University Lectures*

Henrique Medeiros[1], Fernando Batista[1], Helena Moniz[2], Isabel Trancoso[3], and Luis Nunes[4]

1 **Laboratório de Sistemas de Língua Falada - INESC-ID, Lisboa, Portugal**
ISCTE - Instituto Universitário de Lisboa, Lisboa, Portugal
`hrbmedeiros@hotmail.com, Fernando.Batista@iscte.pt`

2 **Laboratório de Sistemas de Língua Falada - INESC-ID, Lisboa, Portugal**
FLUL/CLUL, Universidade de Lisboa, Lisboa, Portugal
`helena.moniz@inesc-id.pt`

3 **Laboratório de Sistemas de Língua Falada - INESC-ID, Lisboa, Portugal**
Instituto Superior Técnico (IST), Lisboa, Portugal
`isabel.trancoso@inesc-id.pt`

4 **ISCTE - Instituto Universitário de Lisboa, Lisboa, Portugal**
Instituto de Telecomunicações, Lisboa, Portugal
`luis.nunes@iscte.pt`

Abstract

This paper presents a number of experiments focusing on assessing the performance of different machine learning methods on the identification of disfluencies and their distinct structural regions over speech data. Several machine learning methods have been applied, namely Naive Bayes, Logistic Regression, Classification and Regression Trees (CARTs), J48 and Multilayer Perceptron. Our experiments show that CARTs outperform the other methods on the identification of the distinct structural disfluent regions. Reported experiments are based on audio segmentation and prosodic features, calculated from a corpus of university lectures in European Portuguese, containing about 32h of speech and about 7.7% of disfluencies. The set of features automatically extracted from the forced alignment corpus proved to be discriminant of the regions contained in the production of a disfluency. This work shows that using fully automatic prosodic features, disfluency structural regions can be reliably identified using CARTs, where the best results achieved correspond to 81.5% precision, 27.6% recall, and 41.2% F-measure. The best results concern the detection of the interregnum, followed by the detection of the interruption point.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases Machine learning, speech processing, prosodic features, automatic detection of disfluencies

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.259

* This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia – under Ph.D grant SFRH/BD/44671/2008, parcially supported by projects CMU-PT/HuMach/0039/2008 and PEst-OE/EEI/LA0021/2011, and also by DCTI - ISCTE – Instituto Universitário de Lisboa.

1 Introduction

Disfluencies are a linguistic mechanism used for on-line editing a message. Disfluencies encompass several distinct types, namely, filled pauses, prolongations, repetitions, deletions, substitutions, fragments, editing expressions, insertions or complex sequences (more than one category uttered) [27]. Those events have been studied from different perspectives, in Psycholinguistics, in Linguistics, in Text-to-speech, and in Automatic Speech Recognition (ASR). The latter will be the focus of our study, since it is well-known that disfluencies are a challenging structure for ASR systems, mainly due to the fact that they are not well recognized and the adjacent words are also influenced and may be erroneously identified.

Automatic speech recognition systems have recently earned their place in the information society, and are now being applied for well-known tasks, like automatic subtitling, speech translation, speech summarization, and production of multimedia content. Speech is a rich source of information from which a vast number of structural phenomena can be extracted, apart from a text stream. Enriching the ASR output with structural phenomena is crucial for improving the human readability, for further automatic processing tasks, and also opens new horizons to a vast range of applications. Disfluencies characterize spontaneous and prepared speech and play a special role as a structural phenomena in speech [12, 4, 6]. Considering them becomes indispensable in the development of a robust and natural ASR systems, because: i) they may trigger readability issues caused by an interruption of the normal flow of an intended message, ii) they provide crucial clues for characterizing the speaker, the speaking styles and iii) also in combination with segmentation tasks, they provide better sentence-like units detection.

This paper analyses the performance of different machine learning methods on the prediction of disfluent sequences and their distinct regions in a corpus of university lectures in European Portuguese. This paper complements the analysis performed in the scope of the work described in [17], where, for the first time, results for disfluency detection on Portuguese university lectures were presented. The specific domain is very challenging, mainly due to the fact that it comprehends quite informal lectures, contrasting with other data already collected of more formal seminars [10].

The chosen algorithms represent state-of-the-art machine learning techniques and are widely used by the scientific community for similar problems. The choice of methods was limited to a subset of methods available in the Weka suite, but other methods currently not available could also be explored, including CRFs (Conditional Random Fields), a promising method for sequence modeling. CARTs, in particular, have been widely adopted for related tasks in the literature [33, 30, 1, 32, 13, 22]. The purpose of this study is to assess the performance of the different methods, and reveal their strengths and weaknesses on the task of identifying the regions of a disfluency.

This paper is organized as follows: Section 2 overviews the literature concerning the detection of disfluencies and corresponding methods. Section 3 describes the *corpus* used in our experiments as well as the multilayer information available. Section 4 describes the adopted features. Section 5 describes the performance metrics that have been used for the evaluation. Section 6 presents experiments for either detecting elements that belong to a disfluent sequence, or distinguishing between those elements. Section 7 points out the major conclusions and presents issues still open for future work.

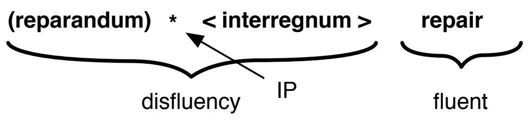


Figure 1 Different regions related to a disfluent sequence.

2 Related Work

Disfluent sequences have a structure composed of several possible regions: a region to be auto-corrected, the *reparandum*; a moment where the speaker interrupts his/her production, known as the *interruption point* (IP); an optional *editing phase* or *interregnum*, filled with expressions such as "uh" or "you know"; and a *repair* region, where speech fluency is recovered [11, 27, 22]. Figure 1 illustrates such structure. Determining such structural elements is not a trivial task [22, 34], but it is known that speakers signal different cues in those regions [9] and several studies have found combinations of cues that can be used to identify disfluencies and repairs with reasonable success [22, 7]. According to [29, 22, 7], based on the analysis of several disfluent types, those cues may relate to segment duration, intonation characteristics, word completion, voice quality alternations, vowel quality and co-articulation patterns [29]. According to [13, 38] fragments can be problematic for recognition if not considered and fairly identified. In a different perspective they are also referred to as important cues to disfluent regions identifiable throughout prosodic features [38]. Even thought fragments are common in human speech, [3] shows that they can present different significant characteristics across languages. Filled pauses are also problematic since they can be confused and recognized as functional words, usually resulting in fragment-like structures that decrease the ASR performance [5, 28]. The potential benefit of modeling disfluencies in a speech recognizer in Spanish has been studied by [26], following a data driven approach.

For European Portuguese, only a recent and a reduced number of studies on characterizing disfluencies have been found in the literature. [36] analyze the acoustic characteristics of filled pauses *vs.* segmental prolongations in a corpus of Portuguese broadcast news, using prosodic and spectral features to discriminate between both categories. Slight pitch descendent patterns and temporal characteristics are pointed out as the best cues for detecting these two categories. [21, 20] use the same university lectures corpus subset also used in the present study and concluded that the best features to identify if a disfluency should be rated as either a fluent or a disfluent are: prosodic phrasing, contour shape, and presence/absence of silent pauses. Recently, [19] analyze the prosodic behavior of the different regions of a disfluency sequence, pointing out to prosodic contrast strategy (pitch and energy increases) between the reparandum and the repair. The authors evidenced that although prosodic contrast marking between those regions is a cross speaker and cross category strategy, there are degrees in doing so, meaning, filled pauses exhibit the highest f_0 increase and repetitions the highest energy one. Regarding temporal patterns, [18] show that the disfluency is the longest event, the silent pause between the disfluency and the following word is longer in average than the previous one, and that the first word of a repair equals the silent pause before a disfluency, being the shortest events.

Different methods have been proposed for similar tasks in the literature, either generative or discriminative. The scientific community often assumes the CARTs produce good results, therefore being the preferred choice. In contrast to single model usage multi-method classifications as well as multi-knowledge sources usually result in better predictions [13, 1, 15, 31, 37].

Table 1 Properties of the Lectra training subset.

Corpus subset →	train+dev	test
Time (h)	28:00	3:24
Number of sentences	8291	861
Number of disfluencies	8390	950
Number of words (including filled pauses and fragments)	216435	24516
Number of elements inside a disfluency	16360	2043
Percentage of elements inside disfluencies	7.6%	8.3%

3 Data

This work is based on Lectra, a speech corpus of university lectures in European Portuguese, originally created for multimedia content production and to support hearing-impaired students [35]. The corpus contains records from seven 1-semester courses, where most of the classes are 60-90 minutes long, and consist mostly of spontaneous speech. It has been recently extended, now containing about 32h of manual orthographic transcripts [25]. Experiments here described use approximately 28h of the corpus to train models, and the remaining portion for testing. Table 1 presents overall statistics about the data.

Besides the manual transcripts, we also have available force-aligned transcripts, automatically produced by the in-house ASR Audimus [23]. The ASR used in this study was trained for the Broadcast News domain, therefore unsuitable for the university lectures domain. The scarcity of text materials in our language to train language models for this domain has motivated the decision of using the ASR in a forced alignment mode, in order not to bias the study with the poor results obtained with an out-of-domain recognizer. The corpus is available as self-contained XML files [2] that includes not only all the information provided by the speech recognition, but also the manually annotated information like punctuation marks, disfluencies, inspirations, etc. Each XML file also includes information related to pitch, energy, duration that comes from the speech signal and that has been assigned to different units of analysis, such as words, syllables and phones.

4 Feature Set

An XML parser was specially created with the purpose of extracting and calculating features from the XML files described in the previous section. The following features were extracted either for the current word (cw) or for the following word (fw): $conf_{cw}$, $conf_{fw}$ (ASR confidence scores), dur_{cw}, dur_{fw} (word durations), $phones_{cw}$, $phones_{fw}$ (number of phones), syl_{cw}, syl_{fw} (number of syllables), $pslope_{cw}$, $pslope_{fw}$ (pitch slopes), $eslope_{cw}$, $eslope_{fw}$ (energy slopes), [$pmax_{cw}$, $pmin_{cw}$, $pmed_{cw}$, $emed_{cw}$ (pitch maximum, minimum, and median; energy median)], $emax_{cw}$, $emin_{cw}$ (energy maximum and minimum), $bsil_{cw}$, $bsil_{fw}$ (silences before the word). The following features involving two consecutive words were calculated: $equals_{pw,cw}$, $equals_{cw,fw}$ (binary features indicating equal words), $sil.cmp_{cw,fw}$ (silence comparison), $dur.cmp_{cw,fw}$ (duration comparison), $pslopes_{cw,fw}$ (shape of the pitch slopes), $eslopes_{cw,fw}$ (shape of the energy slopes), $pdif_{pw,cw}$, $pdif_{cw,fw}$, $edif_{pw,cw}$, $edif_{cw,fw}$ (pitch and energy differences), $dur.ratio_{cw,fw}$ (words duration ratio), $bsil.ratio_{cw,fw}$ (ratio of silence before each word), $pmed.ratio_{cw,fw}$, $emed.ratio_{cw,fw}$ (ratios of pitch and energy medians). Features expressed in brackets were used only in preliminary tests, but their contribution was not substantial and therefore, for simplification, they were not used in subsequent experiments. It is important to notice that some of the information contained in the features that were not used in subsequent experiments is already encoded by the remaining features, such as slopes, shapes, and differences.

Pitch slopes were calculated based on semitones rather than raw frequency values. Slopes in general were calculated using linear regression. Silence and duration comparisons assume 3 possible values, expanding to 3 binary features: $>$ (greater than), $=$ (equal), or $<$ (less than). The pitch and energy shapes expand to 9 binary features, assuming one of the following values $\{RR, R-, RF, -R, --, -F, FR, F-, FF\}$, where $F = Fall$, $- = stationary$, $R = Rise$, and the i^{th} letter corresponds to the word i. The ratios assume values between 0 and 1, indicating whether the second value is greater than the first. All the above features are based on audio segmentation and prosodic features, except for the feature that compares two consecutive words at the lexical level. In future experiments, we plan to replace it by an acoustic-based feature that compares two segments of speech on the acoustic level.

Apart from the previous automatic features, experiments use two additional features that indicate the presence of fragments (FRG) and filled pauses (FP). We are currently using the manual classifications of those categories, but we also aim at verifying the impact of our set of features in the automatic identification of those categories. It is important to notice that while the automatic identification of fragments is still an active research area [13, 38], the automatic identification of filled pauses in spontaneous speech has been applied with an acceptable performance [24, 8].

5 Evaluation Metrics

The following widely used performance evaluation metrics will be applied along the paper: Precision, Recall, F-measure, Slot Error Rate (SER) [16]. All these metrics are based on slots, which correspond to the elements that we aim at classifying. For example, for the task of classifying words as being part of a disfluency, a slot corresponds to a word marked as being part of a disfluency. Most of the results presented in the scope of this paper include all the standard metrics. However, F-measure is a way of having a single value for measuring the precision and the recall simultaneously and, as reported by [16], “this measure implicitly discounts the overall error rate, making the systems look like they are much better than they really are”. For that reason, the preferred performance metric for performance evaluation will be the SER, which also corresponds to the NIST error rate used in their RT (Rich Transcription) evaluation campaigns. Notice, however, that SER is an error metric that assume values greater than 100 whenever the number of errors are greater than the number of slots in the reference.

The Receiver Operating Characteristic (ROC) is another performance metric, based on performance curves, that can also be used for more adequate analysis [14]. It consists of plotting the false alarm rate on the horizontal axis, while the correct detection rate is plotted on vertical. Most experiments reported in this paper also include a ROC value that corresponds to the area under the ROC curve.

6 Experiments and Results

Experiments here described were conducted using Weka[1], a collection of open source machine learning algorithms and a collection of tools for data pre-processing and visualization. Different classification algorithms were tested, namely: Naive Bayes, Logistic Regression, Multilayer Perceptron, CARTs and J48. For each one of the tested algorithms, the default parameters where were used.

The remainder of this section presents two complementary studies concerning the automatic detection of disfluencies and the identification of their structural elements, where the focus lies on comparing the results achieved with different methods. The first study involves a binary classification and aims at automatically identifying which words belong to a disfluent sequence. The second study comprises a multiclass classification that aims at distinguishing between five different regions related with disfluencies: IP, interregnum, any other position in a disfluency, repair, any other position outside a disfluency. Concerning the multiclass classification, details relative to distinct disfluent zone classification performance will be presented.

6.1 Detecting Elements belonging to Disfluent Sequences

This first set of experiments aims at automatically identifying words that belong to a disfluency. Table 2 summarizes the overall performance results, in terms of time taken and correctly classified instances, for binary predicting whether a word (including filled pauses and fragments) belongs to a disfluent sequence or not. Each column represents results for a distinct algorithm, namely: baseline achieved by simply selecting the most common prediction (ZeroR), Naive Bayes (NB), Logistic Regression (LR), Classification and Regression Tree (CART), MultiLayer Perceptron (MLP) and J48. The percentage of Correctly Classified Instances takes into account all the elements that are being classified, and not only slots (*vide* Section 5). The baseline achieved using ZeroR (91.7%) corresponds to marking all words as being outside of a disfluency, which is consistent with the percentage of elements in the test corpus belong to disfluencies (*vide* Table 1). The value referred as Kappa indicates whether a classifier is doing better than chance. The last two lines of the table reveal that both Logistic Regression and CARTs are the most promising approaches. The time taken to build the model is considerable less for Logistic Regression, when compared with the other methods. In fact, Logistic Regression is approximately 85 times faster when compared to CART, and the other performance results presented in the table are quite similar.

The detailed performance results for each method based on slots are also presented in Table 3, where each slot corresponds to elements marked as being part of a disfluency. The first 3 columns report the actual counts for *Correct*, *Inserted* (not marked in the reference),

[1] Weka version 3-6-8. http://www.cs.waikato.ac.nz/ml/weka

Table 2 High level performance analysis for predicting words that belong to disfluencies.

	ZeroR	NB	LR	CART	MLP	J48
Time taken to build the model (seconds)	0.1	551.9	40.5	3412.4	8473.2	3818.5
Time taken to test the model (seconds)	3.2	5.6	3.1	2.0	9.2	1.9
Correctly classified instances (%)	91.7	89.8	94.4	94.4	93.9	94.4
Kappa	0.0	0.362	0.503	0.502	0.489	0.505

Table 3 Detailed performance analysis on predicting words that belong to disfluencies.

Method	Cor	Ins	Del	Precision	Recall	F	SER	ROC
Naive Bayes	891	1339	1152	40.0	43.6	41.7	121.9	0.771
Logistic Regression	765	95	1278	89.0	37.4	52.7	67.2	0.797
CART	754	73	1289	91.2	36.9	52.5	66.7	0.726
MultiLayer Perceptron	799	244	1244	76.6	39.1	51.8	72.8	0.779
J48	778	115	1265	87.1	38.1	53.0	67.5	0.733

Table 4 High level performance analysis for a multiclass prediction.

	ZeroR	NB	LR	CART	MLP	J48
Time taken to build the model (secs)	0.1	574.7	1391.1	6148.8	10209.7	4602.1
Time taken to test the model (secs)	3.4	7.4	3.9	1.8	12.3	1.9
Correctly classified instances (%)	88.7	76.5	91.4	91.5	91.4	91.4
Kappa	0.0	0.223	0.416	0.420	0.414	0.414

and *Deleted* (marked in the reference but not correctly classified) slots. Values presented for *Precision*, *Recall*, *F-measure* and SER (*vide* Section 5) represent percentages. Because CARTs are not probabilistic classifiers, the ROC value can not be fairly computed, and for that reason it was not presented. Results reveal that CART and Logistic Regression present the best performance values, where CARTs achieved a better precision and Logistic Regression achieved a better recall. It is interesting to notice that while the F-measure is better for the Logistic Regression, the SER is the best for CART, which might be a more meaningful measure.

6.2 Distinguishing between all the Structural Elements

This set of experiments aims at identifying the structural elements that compose or are related to a disfluency. Table 4 summarizes the overall performance results, in terms of time taken and correctly classified instances. The time taken to build the model is considerable less for Naive Bayes, but the performance is above the baseline achieved using ZeroR. Performing Logistic Regression is also less time consuming than the other three methods, but such difference is now less notorious than before. The values presented in the last two rows suggest that all approaches (except Naive Bayes) achieve similar performances, and that CARTs achieve the best results by a small difference.

Table 5 presents a more detailed analysis of the performance of each one of the approaches, revealing that CART should be the best choice for this type of problem. The table also includes the number of substitutions (Sub), which correspond to the number of mistakes

Table 5 Detailed performance analysis for a multiclass prediction.

Method	Cor	Ins	Del	Sub	Precision	Recall	F-measure	SER
Naive Bayes	980	3983	1317	466	18.1	35.5	23.9	208.7
Logistic Regression	763	118	1883	117	76.5	27.6	40.6	76.7
CART	762	71	1899	102	81.5	27.6	41.2	75.0
MultiLayer Perceptron	753	99	1891	119	77.5	27.3	40.3	76.3
J48	749	96	1891	123	77.4	27.1	40.2	76.4

Table 6 Zone discrimination CART results.

	Cor	Ins	Del	Sub	Prec.	Recall	F	SER
IP	271	82	449	0	76.8	37.6	50.5	73.8
interregnum	366	12	1	0	96.8	99.7	98.3	3.5
other word inside disfluency	19	33	937	0	36.5	2.0	3.8	101.5
repair	106	46	614	0	69.7	14.7	24.3	91.7
outside disfluency	*21682*	*23581*	*43435*	*0*	*47.9*	*33.3*	*39.3*	*102.9*
Overall performance	762	71	1899	102	81.5	27.6	41.2	75.0

Table 7 Cart confusion matrix.

Classified as →	IP	interregnum	in-disf	repair	outside disf
IP	271	0	19	5	425
interregnum	0	366	0	0	1
other word inside disfluency	58	0	19	14	865
repair	0	3	3	106	608
outside disfluency	24	9	11	27	*21682*

between the different possible slots. The best precision is by far achieved using a CART and Logistic Regression achieved the second best performance, and all metrics reflect this difference coherently.

6.2.1 Detailed CART Results

Taking into account that the best results previously presented concern CARTs, this section presents detailed performance results obtained with this approach. The best results for automatically identifying each one of the structural elements that are related with disfluencies are detailed in Table 6. The table reveals that, from all the structural elements related with a disfluency, the interregnum is by far the easiest to detect. That is an expected result because that information about filled pauses and fragments is being provided as a feature. All the presented results reveal a good precision when compared to recall except for interregnum. Good results considering both the F-measure and SER are also achieved for the detection of the IP. That is also not surprising, because the interruption point is often followed by filled pauses and sometimes preceded by fragments, for which our feature set includes information. The IP region is often referred as containing good clues for detecting disfluencies because the surrounding regions present characteristic contrasts in terms of feature values. Detecting the repair zone can also be performed at a considerably high precision, contrasting with the corresponding recall. A more deep word context analysis is needed to improve the recall performance on this classification. The worst classification refers to words that are marked as being part of a disfluent sequence, but not being neither the IP nor the interregnum, which correspond to words that most of the times are similar to fluent words. The line concerning the elements *outside a disfluency* refers to elements that were not considered one of the five possible structural elements of a disfluency, and correspond to non-slots.

The previous analysis can be complemented by also taking into consideration the corresponding confusion matrix, which is presented in Table 7. The matrix reveals that most of the elements are classified as being “outside of a disfluency”, the most common situation in the corpus.

7 Conclusions

Different machine learning methods have been tested on the prediction of disfluent sequences and their distinct regions in a corpus of university lectures in European Portuguese. In terms of computational effort, Logistic Regression is the best choice, being much faster than the other classification approaches for binary predictions. Our experiments on the automatic identification of disfluent sequences suggest that similar results can be achieved using either CARTs or Logistic Regression. While CARTs tend to favor a better precision, Logistic Regression result in a better recall. Our experiments that distinguish between structural elements in a disfluent sequence suggest that CARTs are consistently better than the other tested approaches.

This paper complements the first studies that have been performed on detecting disfluencies and disfluency related regions for Portuguese university lectures [17]. For the future, we are planning a similar work for distinguishing between disfluency locations and punctuation marks.

References

1 Don Baron, Elizabeth Shriberg, and Andreas Stolcke. Automatic punctuation and disfluency detection in multi-party meetings using prosodic and lexical cues. In *in Proc. of the International Conference on Spoken Language Processing*, pages 949–952, 2002.

2 Fernando Batista, Helena Moniz, Isabel Trancoso, Nuno Mamede, and Ana Mata. Extending automatic transcripts in a unified data representation towards a prosodic-based metadata annotation and evaluation. *Journal of Speech Sciences*, 2(2):115–138, November 2012.

3 Cheng-Tao Chu, Yun-Hsuan Sung, Yuan Zhao, and Daniel Jurafsky. Detection of word fragments in mandarin telephone conversation. In *INTERSPEECH 2006*. ISCA, 2006.

4 H. Clark. *Using language*. Cambridge: Cambridge University Press, 1996.

5 Martin Corley and Oliver W. Stewart. Hesitation disfluencies in spontaneous speech: The meaning of um. *Language and Linguistics Compass*, 2(4):589–602, 2008.

6 R. Dufour, V. Jousse, Y. Estève, F. Béchet, and G. Linarès. Spontaneous speech characterization and detection in large audio database. In *13-th International Conference on Speech and Computer (SPECOM 2009)*, St Petersburg (Russia), 21-25 june 2009.

7 E. Shriberg. Phonetic consequences of speech disfluency. In *International Congress of Phonetic Sciences*, pages 612–622, 1999.

8 Masataka Goto, Katunobu Itou, and Satoru Hayamizu. A real-time filled pause detection system for spontaneous speech recognition. In *In Proceedings of Eurospeech '99*, pages 227–230, 1999.

9 D. Hindle. Deterministic parsing of syntactic non-fluencies. In *ACL*, pages 123–128, 1983.

10 L. Lamel, G. Adda, E. Bilinski, and J. Gauvain. Transcribing lectures and seminars. In *Interspeech 2005*, Lisbon, Portugal, September 2005.

11 W. Levelt. Monitoring and self-repair in speech. *Cognition*, (14):41–104, 1983.

12 W. Levelt. *Speaking*. MIT Press, Cambridge, Massachusetts, 1989.

13 Yang Liu. Word fragment identification using acoustic-prosodic features in conversational speech. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: Proceedings of the HLT-NAACL 2003 student research workshop - Volume 3*, NAACLstudent '03, pages 37–42, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

14 Yang Liu and Elizabeth Shriberg. Comparing evaluation metrics for sentence boundary detection. In *Proc. of the IEEE ICASSP*, Honolulu, Hawaii, 2007.

15 Yang Liu, Elizabeth Shriberg, Andreas Stolcke, Dustin Hillard, Mari Ostendorf, and Mary Harper. Enriching speech recognition with automatic detection of sentence boundaries and disfluencies. *IEEE Transactions on Audio, Speech and Language Processing*, 14(5):1526–1540, 2006.

16 J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel. Performance measures for information extraction. In *Proc. of the DARPA Broadcast News Workshop*, Herndon, VA, Feb. 1999.

17 Henrique Medeiros, Helena Moniz, Fernando Batista, Isabel Trancoso, and Luis Nunes. Disfluency detection based on prosodic features for university lectures. Interspeech 2013 (submitted).

18 Helena Moniz, Fernando Batista, Ana Isabel Mata, and Isabel Trancoso. Analysis of disfluencies in a corpus of university lectures. In *ExLing 2012*, August 2012.

19 Helena Moniz, Fernando Batista, Isabel Trancoso, and Ana Isabel Mata da Silva. Prosodic context-based analysis of disfluencies. In *In Interspeech 2012*, 2012.

20 Helena Moniz, Fernando Batista, Isabel Trancoso, and Ana Isabel Mata. *Toward Autonomous, Adaptive, and Context-Aware Multimodal Interfaces: Theoretical and Practical Issues*, volume 6456 of *Lecture Notes in Computer Science*, chapter Analysis of interrogatives in different domains, pages 136–148. Springer Berlin / Heidelberg, Caserta, Italy, 1st edition edition, January 2011.

21 Helena Moniz, Isabel Trancoso, and Ana Isabel Mata. Classification of disfluent phenomena as fluent communicative devices in specific prosodic contexts. In *Interspeech 2009*, Brighton, England, 2009.

22 C. Nakatani and J. Hirschberg. A corpus-based study of repair cues in spontaneous speech. *Journal of the Acoustical Society of America (JASA)*, (95):1603–1616, 1994.

23 J. Neto, H. Meinedo, M. Viveiros, R. Cassaca, C. Martins, and D. Caseiro. Broadcast news subtitling system in portuguese. In *ICASSP 2008*, pages 1561–1564, 2008.

24 Douglas O'Shaughnessy. Recognition of hesitations in spontaneous speech. In *Proceedings of the 1992 IEEE international conference on Acoustics, speech and signal processing - Volume 1*, ICASSP'92, pages 521–524, Washington, DC, USA, 1992. IEEE Computer Society.

25 Thomas Pellegrini, Helena Moniz, Fernando Batista, Isabel Trancoso, and Ramon Astudillo. Extension of the lectra corpus: classroom lecture transcriptions in european portuguese. In *SPEECH AND CORPORA*, 2012.

26 Luis Javier Rodriguez Fuentes and M.I. Torres. Spontaneous speech events in two speech databases of human-computer and human-human dialogues in spanish. *Language and Speech*, 49(3):333–366, September 2006.

27 E. Shriberg. *Preliminaries to a Theory of Speech Disfluencies.* PhD thesis, University of California, 1994.

28 Elizabeth Shriberg. Disfluencies in switchboard, 1996.

29 Elizabeth Shriberg. To "errrr" is human: Ecology and acoustics of speech disfluencies. *Journal of the International Phonetic Association*, 31:153–169, 2001.

30 Elizabeth Shriberg, Rebecca Bates, and Andreas Stolcke. A prosody-only decision-tree model for disfluency detection. In *Proc. EUROSPEECH*, pages 2383–2386, 1997.

31 Matthew Snover and Bonnie Dorr. A lexically-driven algorithm for disfluency detection. In *in Proc. of HLT/NAACL*, pages 157–160, 2004.

32 Andreas Stolcke, Noah Coccaro, Rebecca Bates, Paul Taylor, Carol Van Ess-Dykema, Klaus Ries, Elizabeth Shriberg, Daniel Jurafsky, Rachel Martin, and Marie Meteer. Dialogue act modeling for automatic tagging and recognition of conversational speech. *Comput. Linguist.*, 26(3):339–373, September 2000.

33 Andreas Stolcke, Elizabeth Shriberg, Rebecca Bates, Mari Ostendorf, Dilek Hakkani, Madelaine Plauche, Gokhan Tur, and Yu Lu. Automatic detection of sentence boundaries and disfluencies based on recognized words, 1998.

34 Frederik Stouten, Jacques Duchateau, Jean-Pierre Martens, and Patrick Wambacq. Coping with disfluencies in spontaneous speech recognition: Acoustic detection and linguistic context manipulation. *Speech Communication*, 48(11):1590–1606, 2006.

35 Isabel Trancoso, Rui Martins, Helena Moniz, Ana Isabel Mata, and Céu Viana. The lectra corpus - classroom lecture transcriptions in european portuguese. In *LREC*, 2008.

36 A. Veiga, S. Candeias, C. Lopes, and F. Perdigão. Characterization of hesitations using acoustic models. In *International Congress of Phonetic Sciences - ICPhS XVII*, volume -, pages 2054–2057, August 2011.

37 Andreas Stolcke Yang Liu, Elizabeth Shriberg. Automatic disfluency identification in conversational speech using multiple knowledge sources. *Trans. Audio, Speech and Lang. Proc.*, 17(7):1263–1278, September 2003.

38 Jui-Feng Yeh and Ming-Chi Yen. Speech recognition with word fragment detection using prosody features for spontaneous speech. *Applied Mathematics and Information Sciences*, 2012.

Syntactic REAP.PT: Exercises on Clitic Pronouning*

Tiago Freitas[1], Jorge Baptista[2], and Nuno Mamede[1]

1 **IST – Instituto Superior Técnico**
L^2F – Spoken Language Systems Laboratory – INESC ID Lisboa
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
Tiago.Freitas@ist.utl.pt, Nuno.Mamede@ist.utl.pt

2 **Universidade do Algarve, FCHS/CECL**
Campus de Gambelas, 8005-139 Faro, Portugal jbaptis@ualg.pt

Abstract

The emerging interdisciplinary field of Intelligent Computer Assisted Language Learning (ICALL) aims to integrate the knowledge from computational linguistics into computer-assisted language learning (CALL). REAP.PT is a project emerging from this new field, aiming to teach Portuguese in an innovative and appealing way, and adapted to each student. In this paper, we present a new improvement of the REAP.PT system, consisting in developing new, automatically generated, syntactic exercises. These exercises deal with the complex phenomenon of pronominalization, that is, the substitution of a syntactic constituent with an adequate pronominal form. Though the transformation may seem simple, it involves complex lexical, syntactical and semantic constraints. The issues on pronominalization in Portuguese make it a particularly difficult aspect of language learning for non-native speakers. On the other hand, even native speakers can often be uncertain about the correct clitic positioning, due to the complexity and interaction of competing factors governing this phenomenon. A new architecture for automatic syntactic exercise generation is proposed. It proved invaluable in easing the development of this complex exercise, and is expected to make a relevant step forward in the development of future syntactic exercises, with the potential of becoming a syntactic exercise generation framework. A pioneer feedback system with detailed and automatically generated explanations for each answer is also presented, improving the learning experience, as stated in user comments. The expert evaluation and crowd-sourced testing positive results demonstrated the validity of the present approach.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases Intelligent Computer Assisted Language Learning (ICALL), Portuguese, Syntactic Exercises, Automatic Exercise Generation, Clitic Pronouning

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.271

1 Introduction

In the last decades, an increased appearance of targeted and adapted products has been seen replacing mass-oriented and generic ones in many areas, including advertising, news and information, and, recently, even "Personalized Medicine"[1] is being researched and applied. Technology has changed how people use and treat information, making them to expect

* This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2011 and under FCT project CMU-PT/HuMach/0053/2008.

[1] http://en.wikipedia.org/wiki/Personalized_medicine (last visited in October 2012)

increasingly personalized and dynamic information systems, as opposed to the static and generic means of obtaining and processing information of the past.

In the education area, these trends also apply and have had a high impact in the learning process, where attention and motivation are of utmost importance, and teaching materials must be appealing to the students.

It is in this context that the Computer Assisted Language Learning (CALL) research area has appeared, with the aim of developing tutoring tools adapted to the students' expectations and their specific needs, and thus improving the learning process.

The REAP (REAder-specific Practice) project[2] is one of such systems, developed at CMU[3] by the LTI[4] for the teaching of the English language. It aims at teaching vocabulary and practice reading skills (lexical practice), using dynamic games and exercises, adapted to each student learning level and interests, helping teachers to target and accompany each student individually. It uses real documents extracted from the web, providing recent, varied, and thus more motivating reading material. Automatic exercise generation, one of the most important and differentiating features of REAP, is made possible by the application of computational linguistics, which is one of the characteristics of the specialized CALL systems in the emerging interdisciplinary field of Intelligent Computer-Assisted Language Learning (ICALL)[5].

The REAP.PT[6] project aims to bring the REAP learning strategies to the Portuguese language. The lexical learning component, analogue to the original REAP system, is comprised of the text reading and question generation phases [13, 7]. More recently, a listening comprehension module was also developed [14]. The system was then extended to include syntax learning as well [12].

The goal of the present work is to continue the development of the syntactic module of the REAP.PT tutoring system, through the development of additional exercises. This exercises should exhibit the same features that make the tutoring tool compelling to both students and teachers. Namely, they should be automatically generated and use real texts as source.

In this context, a new module of exercises was developed in this project, focusing on the the pronominalization of syntactic constituents. This exercise is often presented in grammar drills in Portuguese textbooks, and also constitutes a challenging aspect for language learners.

2 Related Work

In this section, a brief review of the related work is made. Firstly, some automatic question generation systems for other languages are presented, then a succint description of current syntactic textbook exercises on pronominalization is done. Finally, section 3 describes the current architecture of the REAP.PT system, where this work is included.

2.1 ICALL Systems

There are not many ICALL systems that include automatic generation of exercises, and even less for syntactic exercises. FAST [6] (Free Assessment of Structural Tests) is an automatic

[2] `http://reap.cs.cmu.edu` (last visited in October 2012)
[3] Carnegie Mellon University - `http://www.cmu.edu` (last visited in October 2012)
[4] Language Technologies Institute - `http://www.lti.cs.cmu.edu` (last visited in October 2012)
[5] `http://purl.org/calico/icall` (last visited in October 2012)
[6] `http://call.l2f.inesc-id.pt/reap.public` (last visited in October 2012)

question generation system for grammar tests in the English language, using a method that involves representing the questions' characteristics as structural patterns (surface patterns made of POS tags), and applying those patterns in order to transform sentences into exercises (multiple-choice and error detection questions). Arikiturri [3, 2] is a modular and multilingual automatic question generation system. It is currently implemented for Basque and English language learning and science domains. It can generate several types of questions: error correction, fill-in-the-blank, word formation, multiple-choice and short answer questions. It uses a question model to represent the exercises (as well as the information relating to their generation process). It also has a web-based post-editing environment.

2.2 Current Syntactic Exercises on Pronominalization

There are several pronominalization exercises in textbooks and on-line resources: given three forms of pronouns, choose the right one to replace the signalled constituent; correct and incorrect sentences, that must be classified according to clitic placement; given a small text with signalled pronouns, rewrite the text replacing the pronouns with their corresponding antecedents; given a declarative affirmative sentence with clitics, transform it to the corresponding negative sentence; and cloze questions, in which the student has to choose (multiple-choice) or fill in the correct pronoun to replace the signalled constituent.

The last type of exercise, cloze questions, is the easiest to automatically generate, since the exercises can be produced by manipulating the original sentence. Other exercises involve text generation, which is complex and less objective in their evaluation. [16] is a tool that helps teachers producing corpus-based cloze questions. However, no system was found for Portuguese that both automatically generates and corrects (cloze) questions, as it is aimed here.

3 REAP.PT Architecture

3.1 Old REAP.PT Architecture.

The initial REAP.PT architecture, focused on reading comprehension and vocabulary exercises, consists of several components. The Web Interface component is responsible for the user interaction with the system and information exchange between the database and the listening comprehension module. A listening comprehension module provides text-to-speech audio playback of text presented to the user, so that the students can also train their understanding of the spoken language. The database module is divided in two relational databases. The first, specific to REAP.PT, contains the system state such as user information, text information, focus words and related vocabulary questions and distractors (or foils). The second database stores the lexical resources. A filter chain is used to select a subset of the corpus that fits within certain practical and pedagogical constraints [13]. The topic and readability classifiers run on the output of the filter chain and classify the texts according to topic and reading level [13]. The question generation module is responsible for the generation of vocabulary exercises given to the students after each text reading.

The work on the question generation module started in Correia [7], with a focus on vocabulary *cloze* (*fill-in-the-blank*) questions, and the study of the distractors, the wrong multiple-choice alternatives. The existing exercises include definition questions, synonym questions, hyperonym/hyponym questions, cloze questions about the text, and syntactic exercises.

The current syntactic exercises in REAP.PT [12] are the 'Choice of mood in subordinate clauses' exercise and the 'Nominal Determinants' exercise. The 'Choice of mood in subordinate clauses' exercise aims to teach the syntactic restrictions imposed by the subordinative conjunctions on the mode of the subordinate clause they introduced. The rule-based parser XIP-PT [11], based on XIP [1], is used to extract relevant dependencies. Distractors are then generated using the L^{2}F VerbForms[7] word form generator for verbs, and a set of rule-based restrictions are applied to reduce ambiguity.

The 'Nominal Determinants' exercise aims to teach distributional constraints between a determinative noun and the noun it determines (e.g. *copo de leite* 'glass of milk'), and at the same time the relationship between collective names and common (e.g. *mata de cedros* 'wood of cedars'). A feedback system teaches the student the missed definitions, giving examples and images illustrative of the determinative nouns.

The architecture of the syntactic exercise generation can be seen in Figure 1. The result from the syntactic analysis of the corpus (output of the XIP-PT parser) consists of XML files containing the syntactic tree of each sentence and the syntactic dependencies between the sentences' nodes.

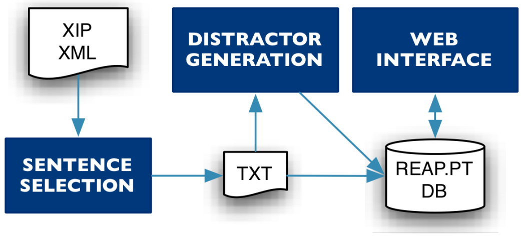


Figure 1 REAP.PT syntactic exercises architecture.

In the sentence selection phase, the XIP output is processed, and the syntactic features are analysed in order to select the stems that are to be used to generate the questions. This phase is performed using the Hadoop[8] Map-Reduce framework for distributed processing, in order to reduce the processing time. In each map operation one sentence is processed, using the DOM (Document Object Model), which represents the XML in a tree structure that is then traversed recursively, using flags when a relevant dependency is found.

3.2 New Exercises Generation

The previous exercise generation architecture and its implementation made it difficult to factorize and adapt it to the new exercise that is here proposed. The previous syntactic exercises used cloze questions (fill-in-the-blank). In the pronominalization exercise, the distractors are sentences built anew by manipulating the syntactic construction of the original stem sentence, namely by deleting and adding lexical material and by changing some of the stem's words (the verb), adjusting it to the pronoun shape (and vice-versa).

The following challenges were considered: selection rules complexity, several different sentence types, and generation metadata for a feedback system. The intention behind this new architecture was not only to simplify the implementation of the proposed exercise, but

[7] `https://www.l2f.inesc-id.pt/wiki/index.php/VerbForms` (last visited in October 2012)
[8] `http://hadoop.apache.org` (last visited in October 2012)

also that it be easily applied in the creation of future exercises, so that it may evolve into a framework for exercise generation. The general architecture is presented in Figure 2.

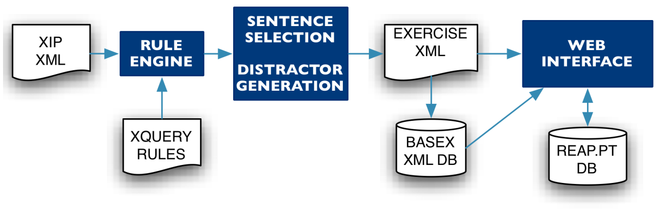


Figure 2 REAP.PT new syntactic exercises architecture.

In order to develop the exercises, the STRING [11] NPL processing chain is used to analyze the corpus sentences, which outputs the syntactic tree and dependencies in XML [10]. The need for a high-level XML processing language was identified, to replace the existing use of the DOM, one of the leading causes of complexity. In addition, to satisfy the requirement of generation metadata, the exercises themselves are to be generated in XML, making it easier process and add new attributes.

Several alternatives were considered, namely *Scala* [8], *XDuce* [9], *CDuce* [4], and *XQuery* [5]. Xquery was ultimately chosen, for several reasons: having a W3C recommendation, the available resources about the language are more widespread; there are many efficient and free implementations; high-level operators (union, document order comparison, and node selection XPath axis were useful for sentence selection and generation) ; there are several native XML databases that include XQuery processors (BaseX[9] was used to generate and store the exercises).

3.3 Rule Engine

Since the analyzed corpus (with the STRING processing chain) used to generate the exercises is approximately 165GB in size, the Hadoop[10] Map-Reduce framework for distributed processing was used. It had already been used in the previous syntactic exercises for sentence selection, using the DOM. But this required a new verbose Java program for each exercise, increasing complexity.

A new Java program was created, named *rule engine*, that uses the Hadoop framework and processes sentences (represented by XML *LUNIT* nodes), using the map function. It searches a *rules* folder for XQuery files, each representing a rule that selects and processes a sentence type. Since rules for each sentence type can become quite complex, it is useful to isolate them. Each *LUNIT* node is then processed with each rule, outputting the exercise XML generated from that sentence.

Each XQuery "rule" selects a type of sentence, using several features and dependencies, and generates the exercise according to that sentence type. Some examples are negative sentences, subordinate clauses or the presence of a verbal chain (with auxiliary verb).

Since in the proposed exercise the answer and distractor generation required the analysis of many syntactic features and dependencies, it was done at the same time as the sentence selection. The number of distractors was also limited for each type. When a distractor type does not require the analysis of syntactic information and has many possible variations, it

[9] `http://basex.org` (last visited in October 2012)
[10] `http://hadoop.apache.org` (last visited in October 2012)

can be generated on-the-fly by the interface (for example, if the variation is in pronoun or word form).

For the XQuery rules, a module was created factorizing the code common to all sentence-type rules. The rule engine program along with this function module could be used in the development of new exercises, and while untested in this regards as only one exercise was developed, could be the beginning of an exercise generation framework. As an example, the functions that output the exercise can receive in their arguments sequences of attributes to be present in the exercise (for example, with features explaining the exercise generation).

4 Pronominalization Exercise

The goal of this exercise is to learn how replace a constituent by a pronoun, in a given sentence. This goal is achieved by cloze question, consisting in a stem is provided where the target constituent highlighted, and a set of alternative answers, a correct form and three incorrect forms, or distractors.

Pronouns can have tonic or atonic forms. Atonic forms are prone to cliticization, when they are moved next to a verb. For this exercise we are interested in the atonic forms, because they are the most problematic to students, since they have more complex restrictions (involving a high number of features and dependencies).

The list of atonic pronouns is: *me, te, se, nos, vos* / *o, a, os, as* / *lhe, lhes*. Only the 3rd person pronouns will be considered, because those are the ones that can substitute a complement in the accusative or dative cases.

There are three grammatical aspects present in pronominalization exercises that are interconnected:

Form The form of the pronoun, according to the verb termination, and the spelling rules of the verb. Contractions of two pronouns also have to be considered.
For example, if the verb terminates with *-r*,*-s* or *-z*, the accusative, 3rd person pronouns *o, a, os, as* assume the form *lo, la, los, las*. In that case, the verb looses its last letter and it is accentuated according to general spelling rules. If the verb terminates with nasal sounds *-m, -õe* or *-ão*, the same pronouns assume the form *no, na, nos, nas*, but the verb remains unaltered.

Case The case of the pronoun, according to its syntactic function. The complement function is determined by the verb it depends on and the pronouns that replaces it takes the correspondent case.

Position The position of the pronoun in the sentence. It can appear at the left or right of the verb. In the future or conditional tenses, it appears between the verbal root form and the tense ending morphemes (*lavá-lo-ei* “I will wash it”; *lavá-lo-ia* “I would wash it”).

Example

Choose the right pronominalization of the constituent signaled in bold:

- Stem from the corpus:
- *O Pedro deu* ***o livro*** *à Ana.* (Pedro gave **the book** to Ana.)

  Correct answer:
- *O Pedro deu-**o** à Ana.* (Pedro gave **it** to Ana.) [The pronoun should be in the accusative case because the constituent is the direct complement. The correct position for the clitic is after the verb because this is a declarative, affirmative sentence, the verb the pronoun depends on is not in a sub clause and no special quantifiers on the subject nor any adverbs interfere with the clitic position.]

Distractors:

- *O Pedro deu-**lhe** à Ana.* (Pedro gave **to_him** to Ana.) [Dative case instead of accusative.]
- *O Pedro deu-**lo** à Ana.* (Pedro gave **it** to Ana.) [Wrong pronoun form.]
- *O Pedro **o** deu à Ana.* (Pedro **it** gave to Ana.) [Wrong clitic position.]

4.1 Specific Exercise Architecture

For this exercise, the rule engine program was used to process the sentences with several XQuery "rules". One rule was used for each set of sentence features that affect the complement to be pronominalized. These rules are associated with the pronoun positioning rules (loosely referred to as *sentence types* in this document). This allows to better isolate the sentence type selection that affects clitic positioning, since it is a major linguist problem and the most complex for this exercise, involving the higher number of features and dependencies (see section 4.5).

Each sentence could in principle be selected by more than one rule, for two reasons:

- Each sentence can have several complements that can be pronominalized, thus generating more than one exercise. The complements can be in different clauses, and so can be affected by sets of features belonging to different rules / sentence types. In this case, each complement is processed by the corresponding rule and ignored by the others.
- It is possible that more than one rule applies to a single complement, because the feature sets can overlap. For example, a negative clause that attracts the clitic to the pre-verbal position, and a clitic-attracting adverb after the verb. These combinations complicate the exercise both in terms of coding and to the student, so they were not explored in the present work. Since the rules are complex, it is arguably better to teach them to the students separately and not in combination. The rules are therefore coded as mutually exclusive, eliminating sentences with complements in clauses that are affected by multiple rules. However, solutions to this problem were considered. In this case, most of the combinations can be solved by setting rule precedence, which can be done in the rule engine program, by ordering the rules names alphabetically. The rules would cease to be mutually exclusive, and when a rule were matched, the others would be discarded. This feature can be used in future exercises that may require it, or to teach the precedence of the clitic positioning rules.

4.2 Sentence Selection

The generation process starts with a sentence from the corpus, from where target patterns (constituents) are extracted. Several filters were added to eliminate unsuitable exercises, such as maximum word number and presence of clitics of the same case being taught. There are also filters to prevent sentences with NLP analysis errors to be proposed for generation. One example are sentences with the ambiguous word *que* (that/which), which in many cases introduces a subclause. However, parsing errors sometimes ignore the subclause status, introducing errors in the exercise generation. Such sentences are filtered. Other filters apply to each phase of the generation, described on the following sections.

4.3 Complement Selection and Analysis

The pronoun case is an argument of the rules, and it is used to get the complement dependencies corresponding to the accusative ("*CDIR*" dependency) or dative ("*CINDIR*") cases.

In the evaluation, only the accusative case was tested, using the direct complement dependency, because the indirect complement dependency was not present in enough sentences in testing, and because it is not fully implemented in the STRING processing chain yet.

Some filters were applied to the complement selection: complements have to be noun phrases; complements is subclauses should not be pronominalized; indefinite complements cannot be pronominalized; complements cannot have appositions; the complement cannot be followed by a relative clause introduced by (a facultative preposition and) *que/o qual/cujo*.

The complement dependencies in STRING only detect the head of the constituent. To recover the entire constituent, several steps were taken. The basic selection consists of including the whole node in which the complement head appears. Then, for each complement head, modifiers are added in a recursive fashion. The modifiers can be adjectives or prepositional phrases which start with *de* (of). When there is a conjunction of several complement dependencies on the same verb, they are joined. If a proper noun immediately follows (without punctuation) the whole complement, it is also added, since there is a very high probability of belonging to it. The modifiers can only be included in the complement if they immediately follow it, ignoring punctuation and conjunctions, as in *os próximos ministros **de** a Defesa **e de** as Relações Exteriores* (the next ministers of Defense and of Foreign Affairs), since there can be adjective modifier dependencies that apply to the complement head that are separated from it and do not belong to the constituent.

Finally, there can be recursive modifiers to the modifiers, which must also be included. This is why the attachment must be done in a recursive and incremental method. In the sentence *A GF confiscou ainda **a viatura ligeira de marca Bedford**.* (The GF also seized the Bedford car), the PP *de marca* was added because it starts with *de*, and *Bedford* was added for being a proper noun that follows the complement.

When a PP is attached to the complement incorrectly, or when a PP should be part of the complement but is not for lack of linguistic information, the well-known *PP-attachment* problem occurs. This problem cannot currently be solved using the information provided by the STRING processing chain. The first case can be exemplified in the sentence *Importante é acima de tudo a noção de servir* [***o utente de forma***] *eficaz.* (It is important, above all, the concept of serving the user in a effective way), in which the PP should not have been included in the complement. The second case can be seen in the sentence *As exportações serviriam para justificar* [***a saída dos materiais***] *comprados por Joaquim Oliveira.*(The exports would serve to justify the exit of the materials bought by Joaquim Oliveira), in which the last PP was not attached to the complement as it should.

In order to be pronominalized with correct agreement, the gender and number of the complement need to be calculated. In principle, the gender and number of the head of the complement are used for this calculation. If the determiner is an article, its gender/number are used. And if there is a determiner quantifier, the decision depends on its partitive nature. If the quantifier is partitive (`SEM-MEASOTHER` feature), the gender/number are that of the complement head (ex: *metade do **investimento** total* (half of the total investment), pronoun:*o*). Otherwise, the gender/number comes from the quantifier (ex: *fardos de **palha*** (straw bales), pronoun: *os*).

If there is more than one complement head, the number is plural, and the masculine gender takes precedence over the feminine, e.g. *O João levou **a Teresa e o Carlos** ao cinema.* (João took Teresa and Carlos to movies) becomes *O João levou-**os** ao cinema.* (João took them to the movies).

4.4 Pronoun Case and Form Generation

The case is an argument of the generation and depends on the complement dependency. In the dative case, since only 3^{rd} person pronouns were considered for this exercise, so that only two are used, which differ in number. In the accusative case, the pronouns are selected in agreement with gender and number, using a map. However, when they occur connected to the verb by an hyphen, they assume different forms. A function calculated the right form according to the basic accusative pronoun and the verb termination, additionally changing the verb termination according to spelling rules.

4.5 Pronoun Positioning Rules

There are 6 rules for complement pronouning, common to both accusative and datives pronouns. All rules record generation information (e.g. for feedback purposes), such as the verb and its complement, pronoun case and position, etc.

Rule 1: Simplest case of affirmative main clauses without verbal chains

The clitic is placed after the verb and linked by an hyphen, if the verb is the main verb in an affirmative clause; this phenomenon is called *enclisis*. For example:

Mário Soares, por seu lado, elogiou ***a personalidade do visitante***..
*Mário Soares, por seu lado, elogiou-**a**.*
Mário Soares, in his turn, praised ***the visitor's personality/-it***.

Rule 2: Verbal chains

This is the most complex rule, since the constraints are different for each auxiliary verb, and there are many possible variations. In this exercise only verbal chains with one auxiliary verb are considered. There can be four possible positions:

- The clitic is attached to the main verb (enclisis);
- the clitic is moved to the front of the main verb (proclisis);
- the clitic is attached to the auxiliary verb (enclisis);
- the clitic is moved to the front of the auxiliary verb (proclisis).

Only the first tree apply to main clauses, while all four can apply to subclauses and in negative sentences, giving a total of 12 combinations of sentence types and positions.

There are 12 possible combinations of sentence types (main, negative or subclause) and clitic positions. There can be more than one correct position for each verb and feature set.

The constraints on clitic position were obtained mostly by introspection, using example sentences to derive the correct positioning for each feature set. However, given the complexity of the positional constraints, an introspective experimental protocol alone may not be enough to guarantee a high level of confidence in agreement with real language use. As such, a study using the corpus and the STRING NLP processing chain was performed in this work, counting the number of occurrences of clitic positions in each of the auxiliary verbs and recording the presence of the same features used in the introspective study.

Rule 3: Clitic attraction by negation

In negative sentences with negation adverbs *não* 'no/not', *nunca/jamais* 'never', *nem* 'not even/nor', and the like, the clitic is attracted to the pre-verb position. The negation is checked by looking at the *NEG* feature in the verb modifier dependencies `MOD`. This case can be seen in the following example taken from the corpus:

*Não copiamos **os nossos vizinhos**, mas tentamos ser um exemplo.*

*Não **os** copiamos, mas tentamos ser um exemplo.*

Rule 4: Indefinite and negative subjects

This rule deals with pronouns and determiners that modify the subject. Indefinite pronouns, e.g. *alguém* 'somebody' and negative indefinite pronouns e.g. *ninguém* 'nobody', attract the clitic pronoun to the pre-verb position. This also happens when the subject is a common noun with some quantifier determiners and some indefinite determiners. However, some of this pronouns and determiners allow both clitic positions, and so don't generate position distractors.

The subject itself can also be one of these pronouns, instead of being modified by one, as seen in the following examples:

- *Todos os rapazes jogam à bola.* (All boys play football) [quantifier determiner *todos* modifies the subject `NP` *os rapazes*].
- *Todos jogam à bola.* (All play football) [the subject is the quantifier determiner alone].

The `DETD` (definite determiner) and `QUANTD` (quantifier determiner) syntactic dependencies on the subject head were used to get these pronouns. In order to differentiate between them, both for positional and feedback purposes, specific lists were used, since the features from the analysis were not conclusive to determine the type: indefinite pronouns, indefinite determiners, and quantifier determiners.

The pronoun and its type were recorded as attributes in the exercise output, for generation information used in the feedback interface.

Rule 5: Clitic-attracting adverbs

Adverbs allowing both pre- and post-verbal position, attract or leave clitic in its basic position, respectively, depending on the position they occupy in the sentence in relation to the verb they modify.

When there are both pre- and post-verbal clitic-attracting adverbs, the clitic position in the right answer defaulted to the post-verbal position (enclisis), since it is the general position in affirmative main clauses. When this default happens, the position distractor is not presented. As mentioned before, rule combinations are not currently generated. If combinations were used, negation would take precedence over clitic-attracting adverbs (in a negative sentence with an adverb in the post-verbal position).

The clitic-attracting adverb was recorded as an attribute in the exercise output, for generation information used in the feedback interface.

Rule 6: Subordinate clauses

In subordinate clauses, clitics are attracted to pre-verbal position. This takes place in completives, relatives and adverbial subordinate clauses; it is also a feature of direct partial interrogatives.

4.6 Distractor Generation

There are four types of distractors: wrong case, wrong position, combination of wrong case and position, and wrong accusative form distractors.

The case and position distractors are generated by the same function that generates the correct answer, by changing the arguments of the case and position. This is done during the generation phase, since their number is low enough, and the generation needs syntactic information available in that phase. However, the accusative case form distractors are generated during the presentation, by the removal or addition of one character in the clitic from the correct answer.

4.7 Exercise Interface

In the question interface, the original sentence, correct answer and distractors are presented to the student as a multiple-choice selection. Four options are always presented, the correct answer and three types of distractors. A button is present for the student to indicate he/she thinks the exercise has errors, in order for the flagged exercises to be examined by the teacher later. A feedback interface based on templates presents explanations about the answer to the student, along with examples from the sentence, so he/she can understand and learn all the aspects pertaining to the pronominalization (case, position and form). Several grammatical explanations are also included in tool-tips that appear when the user hovers the mouse cursor over the underlined words.

5 Evaluation

5.1 Evaluation Setup

The exercises were generated from the CETEMPublico [15] newspaper corpus, that includes approximately 8 million sentences, according to its official website [11]. Only sentences with less than 20 words were used for this evaluation, because longer sentences would be more difficult for the students to read, and increased the probability of NLP analysis errors in the STRING processing chain. For all sentences, 1,292,888 exercises were generated, and 206,967 exercises for sentences with less than 20 words.

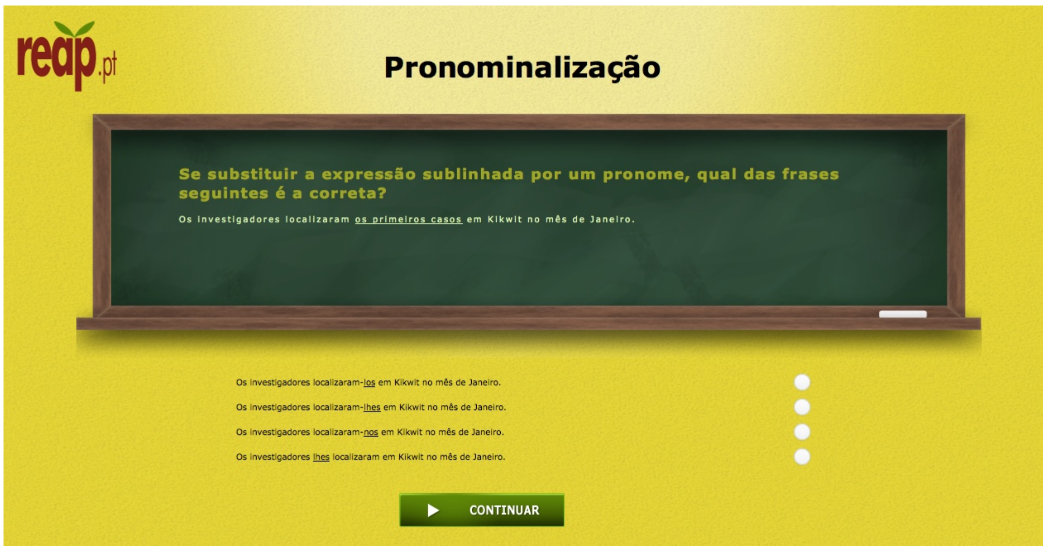
The evaluation of exercises generated from the corpus cannot encompass all generated exercises, as the number of generated exercises is too large for manual inspection. An expert linguist analyzed a random sample of exercises generated from the whole corpus. The exercises were classified by grammatical correction, and annotated with error cause classes. A total of 240 exercises (20 for each of the 6 rules) were evaluated.

Precision was chosen as the evaluation measure, defined as the number of correct exercises by the total number of evaluated exercises.

A website was made available for testing by both native speakers and non-native Portuguese students (Fig. 3).

Native speakers were used because the exercise difficulty is high enough to be a challenge even for natives, and to analyze agreement with the expert analysis in error detection, since the users were given the option to signal that the presented exercises had errors. Six randomly chosen exercises were presented to each user, one for each rule that governs clitic choice and positioning (refer to section 4.5). One of the factors to be analyzed was the nature of

[11] `http://www.linguateca.pt/CETEMPublico` (last visited in October 2012).



Pronominalização

Se substituir a expressão sublinhada por um pronome, qual das frases seguintes é a correta?

Os investigadores localizaram os primeiros casos em Kikwit no mês de Janeiro.

- Os investigadores localizaram-los em Kikwit no mês de Janeiro.
- Os investigadores localizaram-lhes em Kikwit no mês de Janeiro.
- Os investigadores localizaram-nos em Kikwit no mês de Janeiro.
- Os investigadores lhes localizaram em Kikwit no mês de Janeiro.

Figure 3 REAP.PT new syntactic exercises interface.

the errors that are committed by speakers of different levels, namely the distractor type in the wrong answers.In the end of the crowd-sourced testing website, a usability and user satisfaction questionnaire was done, in order to identify aspects that could be improved.

5.2 Evaluation Results

Expert Analysis Results

From the 240 manually analyzed exercises, 75 were found to have errors, and 165 were considered correct. Therefore, the system precision in this evaluation was 68.8%. As it will be seen bellow, significant percentage of the errors are related to shortcomings or errors in the NLP analysis of the corpus. When only taking into consideration the errors directly related with the present work, the precision of the generation module was 86.7% in this evaluation.

For each incorrect exercise, the error causes were annotated by the expert. The following causes were found: PP-attachment problem (in the complement delimitation); *verbum dicendi* (incorrect identification of the inverted subject in a *verbum dicendi* construction); wrong clitic positioning; incorrect POS tagging; incorrect attachment of the pronoun to the verb; and other (corpus errors, fixed expressions, etc.).

Some causes are related to errors or shortcomings in the STRING processing chain analysis (the PP-attachment problem, the incorrect parsing of the subject of the *verba dicendi*, and POS tagging errors). Others are directly related to the present work (clitic positioning and mesoclisis). The PP-attachment problem was the most prevalent, with 44% of the incorrect exercises. The linguistic information in the corpus analysis is not sufficient to solve this problem.

Crowd-sourced Test Results

The native speakers (NS) results were obtained from 114 users, with an average age of 31.5, ranging from 18 to 61 years old. The non-native speakers (NNS) results were obtained from 19 users, with an average age of 31.8, ranging from 20 to 60 years old.

For NS, main clauses had the fewest incorrect answers (10.9%), being the simpler sentences. While verbal chains have the most complex structures and rules, they do not exhibit a higher error percentage than average (20.8%). The highest number of incorrect answers, for both NS (50.5%) and NNS (33.3%) happens with sentences that have indefinite subject (pronouns or determiners). These sentences also happen to be the ones with more exercises deemed erroneous by the users. For NNS, the incorrect answers appear uniformly distributed among the positioning rules, with an average of 29%. Clauses with adverbs had the fewest incorrect answers.

The distribution of incorrect answers by distractor type was also analyzed. For NS, most errors occur with position distractors (45.5%), as expected, since this is the linguistic phenomenon exhibits the most complex set of restrictions. However, though the choice of the pronoun case can be considered to constitute a simpler set of restrictions (agreement with the complement case), the case distractors are the second most common error found (27.9%). For NNS, the position and case combination errors were the most common, showing that this combination is more challenging for NNS than for NS (51.9% vs 9.1%). The form distractor error rate was similar for NNS and NS (22.2% vs 17.5%).

Questionnaire Results

The majority users, both NS and NNS, agreed that the system was easy to use, and that they quickly understood the objective of the exercises.

The statement about exercise difficulty had less agreement between evaluation subjects. 38% of the NS and 13% of the NNS thought the difficulty was acceptable; 37% of NS and 40% of NNS disagreed, noting that the exercises may be difficult. On the other side, 26% of NS and 47% of NNS agreed that the exercises were too easy.

The majority of the users also agreed that the feedback (Fig. 4) was sufficient explanation for the answers. None of the NNS disagreed, compared to the 6% NS that found the feedback could be more detailed, or with more examples as seen in the comments.

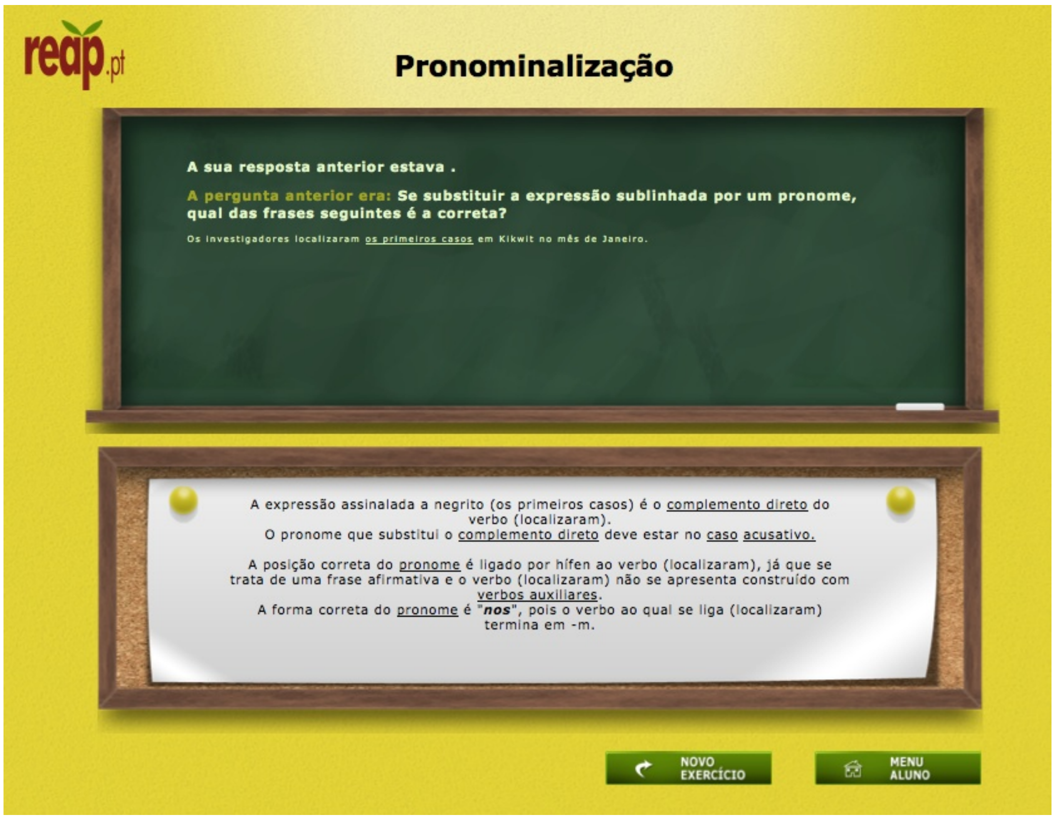
More notably, 71% of the NS and 80% of the NNS agreed or strongly agreed that the system is useful and they learned something by using it. Every NNS considered to have learned something, compared to 10% of NS that did not considered the system useful. As for the global appreciation of the system, the vast majority (85% for both groups) were somewhat or very satisfied.

Questionnaire Comments

In the free-form text comments at the end of the questionnaire, several problems were raised and suggestions were made. The most common were about the lack of context for some sentences, and the complexity of the feedback explanations (on the other hand, many praised the feedback system).

6 Conclusion and Future Work

In an increasingly competitive and dynamic world, it is essential that innovative approaches are developed in the education area and in language education in particular.


Pronominalização

A sua resposta anterior estava .

A pergunta anterior era: Se substituir a expressão sublinhada por um pronome, qual das frases seguintes é a correta?

Os investigadores localizaram os primeiros casos em Kikwit no mês de Janeiro.

A expressão assinalada a negrito (os primeiros casos) é o complemento direto do verbo (localizaram).
O pronome que substitui o complemento direto deve estar no caso acusativo.

A posição correta do pronome é ligado por hífen ao verbo (localizaram), já que se trata de uma frase afirmativa e o verbo (localizaram) não se apresenta construído com verbos auxiliares.
A forma correta do pronome é "***nos***", pois o verbo ao qual se liga (localizaram) termina em -m.

Figure 4 REAP.PT new syntactic exercises interface.

We believe that this work is a valuable new asset for the creation of new syntactic exercises for the European Portuguese language. The general architecture of the REAP.PT syntactic module is expected to make a relevant step forward in order to ease the development effort of future exercises. The pioneer feedback system with detailed and automatically generated explanations for each answer is also believed to be an asset for future exercises.

Some pitfalls were also uncovered during the development, such as the unapparent complexity of some aspects of syntactic exercise generation, and the heavy reliance on correctness and completeness of the NLP analysis of the text. Therefore, the analysis of the exercise generation approach and NLP analysis of the information needs are very important for the success of this exercise's development, and should be performed thoroughly in the initial phases.

This work contributed to the improvement of the STRING processing chain, by identifying shortcomings, such as focus adverbs, and areas of future work, including some whose importance was not evident before their practical application, namely the importance of the identification of the subject in *verbum dicendi* constructions.

Regarding the future work, the errors detected during the evaluation should be corrected; the future-indicative and conditional tenses should be implemented; and exercises could be generated from other corpora, to add variety.

References

1 S. Aït-Mokhtar, J.-P. Chanod, and C. Roux. Robustness beyond shallowness: incremental deep parsing. *Nat. Lang. Eng.*, 8(3):121–144, June 2002.

2 Itziar Aldabe. *Automatic Exercise Generation Based on Corpora and Natural Language Processing Techniques*. PhD thesis, Euskal Herriko Unibertsitatea (University of the Basque Country), San Sebastian, Basque Country, September 2011.

3 Itziar Aldabe, Maddalen Lopez de Lacalle, Montse Maritxalar, and Edurne Martinez. The Question Model inside ArikIturri. In J. Michael Spector, Demetrios G. Sampson, Toshio Okamoto, Kinshuk, Stefano A. Cerri, Maomi Ueno, and Akihiro Kashihara, editors, *Proceedings of the 7th IEEE International Conference on Advanced Learning Technologies, ICALT 2007, July 18-20 2007, Niigata, Japan*, pages 758–759. IEEE Computer Society, 2007.

4 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. *SIGPLAN Not.*, 38(9):51–63, August 2003.

5 Don Chamberlin. XQuery: a query language for XML. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 682–682, New York, NY, USA, 2003. ACM.

6 Chia-Yin Chen, Hsien-Chin Liou, and Jason S. Chang. FAST: an automatic generation system for grammar tests. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, COLING-ACL '06, pages 1–4, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

7 Rui Correia. Automatic Question Generation for REAP.PT Tutoring System. Master's thesis, Instituto Superior Técnico - Universidade Técnica de Lisboa, Portugal, 2010.

8 B. Emir. Extending pattern matching with regular tree expressions for XML processing in Scala. Master's thesis, RWTH Aachen, 2003.

9 Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, May 2003.

10 Nuno Mamede, Jorge Baptista, and Caroline Hagège. Nomenclature of Chunks and Dependencies in Portuguese XIP Grammar 3.1. Technical report, L2F/INESC-ID, Lisbon, May 2011.

11 Nuno J. Mamede, Jorge Baptista, Cláudio Diniz, and Vera Cabarrão. STRING: An Hybrid Statistical and Rule-Based Natural Language Processing Chain for Portuguese. `http://www.propor2012.org/demos/DemoSTRING.pdf`, April 2012.

12 Cristiano Marques. Syntactic REAP.PT. Master's thesis, Instituto Superior Técnico - Universidade Técnica de Lisboa, Portugal, 2011.

13 Luís Marujo. REAP em Português. Master's thesis, Instituto Superior Técnico - Universidade Técnica de Lisboa, Portugal, 2009.

14 Thomas Pellegrini, Rui Correia, Isabel Trancoso, Jorge Baptista, and Nuno J. Mamede. Automatic Generation of Listening Comprehension Learning Material in European Portuguese. In *INTERSPEECH 2011, 12th Annual Conference of the International Speech Communication Association, Florence, Italy, August 27-31, 2011*, pages 1629–1632. ISCA, 2011.

15 Diana Santos and Paulo Rocha. Evaluating CETEMPublico, a Free Resource for Portuguese. In *Association for Computational Linguistic, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference, July 9-11, 2001, Toulouse, France*, pages 442–449. Morgan Kaufmann Publishers, 2001.

16 Alberto Simões and Diana Santos. Ensinador: corpus-based portuguese grammar exercises. *Procesamiento del Lenguaje Natural*, 47:301–309, September 2011.