3rd Symposium on Languages, Applications and Technologies

SLATE'14, June 19–20, 2014, Bragança, Portugal

Edited by

Maria João Varanda Pereira
José Paulo Leal
Alberto Simões

Editors

Maria João Varanda Pereira
CCTC
Escola Superior de Tecnologia e Gestão
Instituto Politécnico de Bragança
mjoao@ipb.pt

José Paulo Leal
CRACS & INESC TEC
Faculdade de Ciências
Universidade do Porto
zp@fcc.fc.up.pt

Alberto Simões
CEHUM & CCTC
Instituto de Letras e Ciências Humanas
Universidade do Minho
ambs@ilch.uminho.pt

Funding
This publication is funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

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

ISBN 978-3-939897-68-2

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/978-3-939897-68-2.

Publication date
June, 2014

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.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/legalcode.
In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:
- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: OASIcs.SLATE.2014.i

ISBN 978-3-939897-68-2 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

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.

Editorial Board

- Daniel Cremers (TU München, Germany)
- Barbara Hammer (Universität Bielefeld, Germany)
- Marc Langheinrich (Università della Svizzera Italiana – Lugano, Switzerland)
- Dorothea Wagner (*Editor-in-Chief*, Karlsruher Institut für Technologie, Germany)

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Semantics in Natural Language Processing

Natural Language Processing Tools and Resources

Preface

The communication from man to man evolved, from long time ago to the communication between man and machine. Communication is achieved when the receiver understands the words, the sentences and knows its meaning in a certain context. A successful communication depends on so many factors: the adequacy of the language type (considering the stakeholders), mutual agreement on the language to use, the ability of the issuer to express himself with the proper words and well-constructed sentences, the ability of the receiver to process the information received and react. The communication between man and computer implies preparing the machine with proper software to be able to receive source texts and perform actions. The study of formalisms and the creation of new approaches associated language processing tasks, is an important research topic in the area of Computer Science.

Techniques and approaches have been developed to speed up and make more efficient the use of the languages either improving the processing tasks of well-known programming languages, constructing new program comprehension tools to be used in the maintenance phase, creating domain specific languages or dealing with problems concerning with natural language processing (NLP) and other topics that relate languages with technology. In SLATE 2014 a challenge is proposed to all participants: update the state-of-the-art, discuss solutions for identified problems, present new ideas and have fun.

The symposium is divided in three tracks:

- The HHL (Human-Human Languages) track is concerned with natural language processing issues and their application in several contexts.
- The HCL (Human-Computer Languages) track is dedicated to exchange ideas about language design, processing, assessment and comprehension and an huge number of applications that can be created to deal with this.
- The CCL (Computer-Computer Languages) track whose main goal is to discuss the use and associated technologies of the XML markup language.

This volume contains the proceedings of the 3^{rd} edition of SLATE, held in the School of Technology and Management of Polytechnic Institute of Bragança, Portugal, during 19th–20th June, 2014. This year, SLATE received a total of 20 full paper submissions and 9 short paper submissions. Each submission was reviewed by at least three Program Committee members, from a global group of 63 researchers. At the end of the review process, 12 papers were accepted as full papers, 4 full papers were invited to submit as short papers, and 6 short papers were also accepted for publication and presentation at the symposium. So, SLATE 2014 had a 24% rejection and 22 papers presentations: 12 full papers (20 min + 5 min of questions) and 10 short papers (10 min + 5 min of questions).

This set of presentations is divided into the following five sessions:

Domain Specific Languages, includes one full paper and four short papers dedicated to the creation of new DSLs and techniques to implement this kind of languages.

Programming Languages and Compilers, includes three full papers and one short paper about programming language implementation, source code analysis, and target code generation.

Program Comprehension, includes two full papers and one short paper about different techniques used for program comprehension: identifier analysis, concern-based projections, and system dependency graph enriched with source code comments.

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Semantics in NLP, includes four full papers and one short paper related to the analysis of semantic in natural language processing, namely on the extraction of semantic relationships from texts, and on the use of semantic-rich structures;

NLP Tools and Resources, includes two full papers and three short papers on identification and analysis of natural language sentences, text alignment and databases.

Moreover, SLATE 2014 program also includes two keynotes: one on **Language-driven Software Development**, by José Luís Sierra from Complutense University of Madrid and another on **Open Information Extraction** by Pablo Gamallo from University of Santiago de Compostela.

The organizers of SLATE 2014 want to thank to many people without whom this event would never be possible: our sponsors Efacec, Computer Science and Technology Center (CCTC), Polytechnic Institute of Bragança (IPB) and Fundação para a Ciência e a Tecnologia (FCT, Portuguese Foundation for Science and Technology); Easychair conference management system; the Program Committee members for spending their time reviewing the papers and writing the reports; the authors of the submitted papers for their contribution and interest in the symposium and, finally, to all participants that came to Bragança to such a fruitful meeting.

Maria João Varanda Pereira
José Paulo Leal
Alberto Simões

List of Authors

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

Jorge Baptista
INESC-ID Lisboa, L2F
Universidade do Algarve – FCHS/CECL
Faro, Portugal
jbaptis@l2f.inesc-id.pt

Luís S. Barbosa
HASLab – INESC TEC
Universidade do Minho
Braga, Portugal
lsb@di.uminho.pt

Fernando Batista
INESC-ID Lisboa & ISCTE
Instituto Universitário de Lisboa
Lisboa, Portugal
fernando.batista@iscte.pt

Rui Brito
CCTC, Departamento de Informática
Universidade do Minho
Braga, Portugal
ruibrito@di.uminho.pt

Gaspar Brogueira
Laboratório de Sistemas de Língua Falada
INESC-ID, Lisboa, Portugal
gmrba@iscte.pt

Simon D. Byers
AT & T Labs
Bedminster NJ
United States of America
headers@gmail.com

João Paulo Carvalho
INESC-ID Lisboa
IST – Universidade de Lisboa
Lisboa, Portugal
joao.carvalho@inesc-id.pt

Nuno Ramos Carvalho
CCTC, Departamento de Informática
Universidade do Minho
Braga, Portugal
narcarvalho@di.uminho.pt

Pedro Carvalho
Departamento de Informática
Universidade do Minho
Braga, Portugal
pedrocarvalho@di.uminho.pt

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

Daniela da Cruz
CCTC, Departamento de Informática
Universidade do Minho
Braga, Portugal
danieladacruz@gmail.com

João Manuel Sousa Fonseca
CCTC, Departamento de Informática
Universidade do Minho
Braga, Portugal
jprophet89@gmail.com

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

Pedro G. Francisco
University of Aveiro, IEETA
Campus Universitário de Santiago
Aveiro, Portugal
goucha@ua.pt

Pablo Gamallo
Universidade de Santiago de Compostela
Galiza, Spain
pablo.gamallo@usc.es

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Paulo Gomes
CISUC, Department of Informatics Engineering
University of Coimbra
Coimbra, Portugal
pgomes@dei.uc.pt

Hawete Hattab
Umm Al-qura University
Department of Mathematics
Makkah, KSA
hshattab@uqu.edu.sa

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

Jan Janoušek
Department of Theoretical Computer Science
Faculty of Information Technologies
Czech Technical University in Prague
Prague, Czech Republic
Jan.Janousek@fit.cvut.cz

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

António Menezes Leitão
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

Jaroslav Málek
Department of Theoretical Computer Science
Faculty of Information Technologies
Czech Technical University in Prague
Prague, Czech Republic

Nuno Mamede
INESC-ID Lisboa, L2F
Instituto Superior Técnico,
Universidade de Lisboa,
Lisboa, Portugal
Nuno.Mamede@l2f.inesc-id.pt

Ilia Markov
INESC-ID Lisboa, L2F
Universidade do Algarve – FCHS
Faro, Portugal
Ilia.Markov@l2f.inesc-id.pt

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

Vítor T. Martins
CCTC, Departamento de Informática
Universidade do Minho
Braga, Portugal
vtiagovm@gmail.com

Rabeb Mbarek
Sfax University
Multimedia Information Systems and Advanced Computing Laboratory
Sfax, Tunisia
rabeb.hattab@gmail.com

Milan Nosáľ
Faculty of Elect. Eng. and Informatics
Technical University of Košice
Košice, Slovakia
milan.nosal@gmail.com

Hugo Gonçalo Oliveira
CISUC, Department of Informatics Engineering
University of Coimbra
Coimbra, Portugal
hroliv@dei.uc.pt

Nuno Oliveira
HASLab – INESC TEC
Universidade do Minho
Braga, Portugal
nunooliveira@di.uminho.pt

Maria João Varanda Pereira
CCTC, Instituto Politécnico de Bragança
Bragança, Portugal
mjoao@ipb.pt

Nuno Pereira
CCTC, Departamento de Informática
Universidade do Minho
Braga, Portugal
nuno.filipe.gomes.pereira@gmail.com

Jaroslav Porubän
Faculty of Elect. Eng. and Informatics
Technical University of Košice
Košice, Slovakia
jaroslav.poruban@tuke.sk

Ricardo Queirós
CRACS & INESC-Porto LA
Escola Superior de Estudos Industriais e de Gestão
Instituto Politécnico do Porto
ricardo.queiros@eu.ipp.pt

Amanda P. Rassi
Federal University of São Carlos-UFSCar
São Carlos, São Paulo, Brasil
aprassi@ualg.pt

Pedro Palma Ramos
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal
pedropramos@tecnico.ulisboa.pt

Flávio Rodrigues
HASLab – INESC TEC
Universidade do Minho
Braga, Portugal
pg22826@alunos.uminho.pt

Ricardo Rodrigues
Centre for Informatics and Systems of the University of Coimbra Coimbra, Portugal
rmanuel@dei.uc.pt

Hugo Rosa
INESC-ID Lisboa
IST – Universidade de Lisboa
Lisboa, Portugal
hugohrosa@gmail.com

António Paulo Santos
GECAD, Institute of Engineering
Polytechnic of Porto
Porto, Portugal
pgsa@isep.ipp.pt

Jose-Luis Sierra
Fac. Informática
Universidad Complutense de Madrid
Madrid, Spain
jlsierra@fdi.ucm.es

Miguel Oliveira e Silva
University of Aveiro, IEETA, DETI
Campus Universitário de Santiago
Aveiro, Portugal
mos@ua.pt

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

Mohamed Tmar
Sfax University
Multimedia Information Systems and Advanced Computing Laboratory
Sfax, Tunisia
mohamedtmar@yahoo.fr

Oto Vale
Federal University of São Carlos-UFSCar
São Carlos, São Paulo, Brasil
otovale@ufscar.br

Committees

Program Chairs

Maria João Varanda Pereira
Instituto Politécnico de Bragança, Portugal

José Paulo Leal
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

José João Almeida
Universidade do Minho, Portugal

Jorge Baptista
Universidade do Algarve, Portugal

Fernando Batista
ISCTE-IUL & INESC-ID, Portugal

Mario Berón
Universidad Nacional de San Luis, Argentina

Michele Bugliesi
Università Ca'Foscari Venezia, Italy

João M. P. Cardoso
Universidade do Porto & INESC TEC, Portugal

Nuno Ramos Carvalho
Universidade do Minho, Portugal

Matej Crepinsek
Univerza v Mariboru, Slovenia

Daniela da Cruz
Universidade do Minho, Portugal

Jürgen Ebert
Universität Koblenz-Landau, Germany

Gabriel David
Universidade do Porto & INESC TEC, Portugal

Daniel Diaz
Université Paris 1, France

Brett Drury
Universidade de São Paulo, Brazil

Jean-Marie Favre
Université Joseph Fourier, Grenoble, France

Luís Ferreira
Instituto Politécnico do Cávado e Ave, Portugal

Jean-Christophe Filliâtre
CNRS & Université Paris Sud, France

Niklas Fors
Lund University, Sweden

Pablo Gamallo
Universidade de Santiago de Compostela, Spain

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

Xavier Gómez Guinovart
Universidade de Vigo, Spain

Ulrich Heid
Universität Hildesheim, Germany

Pedro Rangel Henriques
Universidade do Minho, Portugal

Mirjana Ivanovic
University of Novi Sad, Serbia

Jan Janoušek
Czech Technical University in Prague, Czech Republic

Ján Kollár
Technical University of Košice, Slovakia

Tomaž Kosar
Univerza v Mariboru, Slovenia

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Eugenijus Kurilovas
Vilnius Gediminas Technical University, Lithuania

José Paulo Leal
Universidade do Porto, Portugal

António Menezes Leitão
INESC-ID & Universidade de Lisboa, Portugal

Giovani Librelotto
Universidade Federal Santa Maria, Brazil

João Correia Lopes
Universidade do Porto & INESC TEC, Portugal

Ivan Lukovic
University of Novi Sad, Serbia

Paulo Matos
Instituto Politécnico de Bragança, Portugal

Marjan Mernik
Univerza v Mariboru, Slovenia

Michal Krátký
VŠB – Technical University of Ostrava, Czech Republic

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

Thiago Pardo
Universidade de São Paulo, Brazil

Maria João Varanda Pereira
Instituto Politécnico de Bragança, Portugal

Jaroslav Porubän
Technical University of Košice, Slovakia

Ricardo Queirós
Instituto Politécnico do Porto, Portugal

José Carlos Ramalho
Universidade do Minho, Portugal

Sebastian Rahtz
University of Oxford, United Kingdom

Cristina Ribeiro
Universidade do Porto & INESC TEC, Portugal

Ricardo Rocha
Universidade do Porto, Portugal

Casiano Rodriguez-Leon
Universidad de La Laguna, Spain

Dietmar Seipel
Universität Würzburg, Germany

José Luis Sierra
Universidad Complutense de Madrid, Spain

Josep Silva
Universitat Politècnica de València, Spain

Alberto Simões
Universidade do Minho, Portugal

Boštjan Slivnik
Univerza v Ljubljani, Slovenia

Peter Sloep
Open Universiteit, Netherlands

Simão Melo de Sousa
Universidade da Beira Interior, Portugal

Ralf Steinberger
EC – Joint Research Centre, Italy

Kari Systä
Tampere University of Technology, Finland

António Teixeira
Universidade de Aveiro, Portugal

Jörg Tiedemann
Uppsala University, Sweden

Guido Wachsmuth
Delft University of Technology, Netherlands

Yorick Wilks
Florida Institute for Human and Machine Cognition, USA

Sub Reviewers

Mário Rodrigues
Universidade de Aveiro, Portugal

Paula Christina Figueira Cardoso
Universidade de São Paulo, Brazil

Marcos Garcia
Universidade de Santiago de Compostela, Spain

Liliana Ferreira
Fraunhofer AICOS, Portugal

Organization Committee

Maria João Varanda
Instituto Politécnico de Bragança, Portugal

José Paulo Leal
Universidade do Porto, Portugal

Alberto Simões
Universidade do Minho, Portugal

Pedro Henriques
Universidade do Minho, Portugal

Nuno Ramos Carvalho
Universidade do Minho, Portugal

José Eduardo Fernandes
Instituto Politécnico de Bragança, Portugal

Paulo Matos
Instituto Politécnico de Bragança, Portugal

Paulo Alves
Instituto Politécnico de Bragança, Portugal

Part I

Invited Talks

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Language-Driven Software Development

José-Luis Sierra

Fac. Informática, Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9, 28040 Madrid, Spain
`jlsierra@fdi.ucm.es`

Abstract

Language-driven software development consists in applying computer language design and implementation techniques to build conventional software. The keynote reviews two different language-driven development approaches: *domain-specific languages* (DLSs), and *language-oriented architectures* (LOAs). The DSL approach focuses on the provision of languages specialized in different application aspects, which are used by developers, and even by domain experts, during application construction and maintenance. The LOA strategy, in its turn, conceives applications themselves as coordinated collections of language processors, which can be developed using language implementation tools (parser generators, attribute grammar-based systems, etc.). The presentation of the approaches is supported by case studies from the fields of knowledge-based systems, e-Learning, semi-structured data processing, and digital humanities.

1998 ACM Subject Classification D.3.4 Translator writing systems and compiler generators, D.3.2 Specialized Application Languages, D.2 Software Engineering, D.2.11 Software Architectures

Keywords and phrases domain-specific languages; language-oriented architectures; parser generators; attribute grammars; application domains

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.3

Category Invited Talk

1 Introduction

Nowadays design and implementation of computer languages is a mature and well-understood field, which comprises a wide spectrum of precise and well-founded methods, techniques and tools grounded in strong theoretical and mathematical principles [2]. Regardless of their initial limited applicability to the specialized compiler construction arena, recently these approaches have been recognized as very valuable instruments in many mainstream software development scenarios [43, 10, 19, 23, 24], which leads to a distinguished paradigm of software construction: *language-driven software development*. In these scenarios it is meaningful to recognize the linguistic nature of different aspects of software development, and therefore to undertake these aspects as ones concerning the explicit conception, design and implementation of special-purpose computer languages. The paradigm is particularly suited to address complex development situations involving sophisticated and highly customizable architectures, interdisciplinary teams of developers and domain experts, clearly defined stacks of abstraction levels, etc., in which the language development effort can pay off. Quoting to [1] " *. . . seen from this perspective, the technology for coping with large-scale computer systems merges with the technology for building new computer languages, and computer science itself becomes no more (and no less) than the discipline of constructing appropriate descriptive languages*".


© José-Luis Sierra;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 3–12

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This document summarizes the contents addressed in the keynote *Language-Driven Software Development* given in SLATE'14. The keynote is focused on two different practices concerning this development approach: *domain-specific languages* and *language-oriented architectures*. While the first one is well-established in the research community, and in recent years also among practitioners, the second one is more speculative and inspired by the Speaker's own research at Complutense University of Madrid, Spain (UCM).

2 Software Development based on Domain-Specific Languages

Quoting to [43] a domain-specific language (DSL) is "*a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain*". In this way, the concept of DSL is a well-established one in software development scenarios, where software developers have used several sorts of DSLs for decades (e.g., SQL for updating and querying relational databases, *make* or *ant* for tracking the dependencies among the files of a software systems and for automating the production of these software systems, YACC-like tools for generating parsers from grammar-based specifications, etc.) [31]. In addition, many DSLs have been also provided to facilitate application development in concrete domains [43, 24].

DSLs tailored to concrete application domains are particularly attractive from a software development perspective. Indeed, being more expressive and easy-to-use than a general-purpose programming language, these DSLs make possible, to some extent, the active participation of domain-experts in the development process. In this way, and in an idealized world, the aim of DSLs is to upgrade software developers to DSL designers and implementers, as well as to promote domain-experts to application developers [10, 12]. Of course, in the real world this idealized scenario (sometimes referred as *end-user programming* [16]) is hardly reachable due the fuzziness and dynamic nature of application domains, as well as to the difficulty of capturing some aspects of a system in terms of suitable domain abstractions. Regardless this difficulty, it is a matter of fact that this kind of DSLs can facilitate the active participation of domain-experts during the development process (e.g., they can understand specifications prepared in DSLs with suitable notations, suggest modifications, or even take the responsibility of producing and maintaining specific parts of the applications using suitable DSLs) [10, 12, 44].

Finally, before going into the details of DSL-based development, it is worthwhile to highlight the relationships among this approach and *model-driven engineering* [40, 44]. Indeed, model-driven engineering can be understood as a particular incarnation of DSL-based development, in which DSLs take the form of domain-specific meta-models, and DSL sentences take the form of models resulting of instantiating these meta-models. Therefore, many of the reflections made in the following presentation can be also applied to model-driven engineering without substantial change.

2.1 DSL-oriented Process Models

From a process model perspective, DSL-oriented software development shares many features with generative approaches to software development [21, 6]. Common activities undertook during DSL-oriented software development are *Domain Engineering* Activities, *Language Design and Implementation* Activities and *Application Development* Activities.

2.1.1 Domain Engineering Activities

These activities are oriented to determine the commonalities and the variability of applications in the target application domain [6], and those can be carried out by adopting well-established domain engineering approaches [17, 39, 41]. In DSL-oriented software development variability, which is concerned with the differences among concrete applications, is usually captured in terms of *feature models* [6] that set the conceptual basis for subsequent DSL design. On the other hand, commonality (the core part shared by all the applications in a domain) is essential for providing DSL runtime support (e.g., as a specific object-oriented framework).

2.1.2 Language Design and Implementation Activities

These activities deal with usual aspects concerning the design and implementation of computer languages (lexical and syntactical specification, specification of the static and dynamic semantics, etc.) [11]. For this purpose, a common practice in DSL design is to invert the conventional workflow in computer language design (i.e., going from *concrete* to *abstract* syntax [2]). Indeed, modern tendencies in DSL design promote to start by an abstract syntax. Following the jargon used by the DSL community, abstract syntax is formalized in object-oriented terms, as a set of interrelated classes that makes up the *semantic model* of the DSL [10, 19, 44]. This model will be based on the variability analysis performed during the domain engineering activities.

Once a suitable semantic model is available, it is possible to provide one or several alternative concrete syntaxes. Depending of the intended use of the DSL, concrete syntaxes can be embedded in general-purpose programming languages (*internal* syntaxes) or those can be externally provided (*external* syntaxes). Still, each alternative can be accomplished using a wide range of techniques:

- Concerning internal syntaxes, their provision strongly depends on the features of the host programming language. For instance, LISP-like homogeneous syntaxes have proven to be specially amenable for supporting internal syntaxes for many embedded DSLs [1], while extensible syntaxes enabled by user-defined operators like the supported by Prolog-like languages or by modern functional languages can be particularly useful for better fitting domain notations[15]. Recently, dynamic languages with very expressive grammars have been also adopted as host languages for DSLs [7, 12, 29, 30]. In object-oriented languages, two usual design patterns for internal syntax design are *method nesting expressions* and *fluent* APIs [10, 12].
- In its turn, external syntaxes can be accomplished by using a general-purpose semantic agnostic notation (like XML), by defining a DSL-specific textual syntax (the classic approach promoted by compiler construction textbooks), or even by defining a visual syntax amenable for implementations based on DSL workbenches [4, 14].

Concerning semantics, it is worthwhile to notice that the term *semantic model* is somewhat confusing, since the model has little to do with semantic processing, but it is an explicit formalization of the language abstract syntax. Semantics themselves must be added as processes that operate on the instances of the semantic model. For this purpose:

- As a representation of the abstract syntax, the semantic model usually addresses the structure of the sentences of the language. *Static semantics* deals with additional constraints beyond these structural aspects. While in classic language design static semantics lead to type systems [35], which are subsequently implemented as type checking algorithms on the abstract syntax trees / graphs, in the DSL world it is usual to find

more pragmatic approaches based on constraint languages for object-oriented models, like OCL [19], or in ontology-aware semantic technologies [46].

- *Dynamic semantics*, in their turn, take either the form of translations to target programming languages, or operational semantics specifications. The first scenario leads to the subsequent provision of code generators, while the second one leads to the provision of interpreters operating on instances of the semantic model [10].

In this way, the final implementation of the DSL typically consists of:

- A way of *editing* DSL programs. It can be as simple as using an existing text editor or an existing IDE (e.g., in the case of internal syntaxes or XML-based syntaxes), or as complex as using a dedicated IDE for the DSL. In order to cope with the later scenario it is possible to base the implementation of the DSL in a language workbench [4, 10, 14].
- A *binding* component, which maps concrete syntax sentences in semantic model instances. This component is analogous to the parser of a classic language processing architecture [2]. The exact nature of the component will depend of the nature of the concrete syntax and the semantic model.
- A *static semantic analyzer*. This component will be in charge of ensuring the additional semantic constraints on the semantic model instances.
- A *dynamic semantic infrastructure*. This infrastructure will vary on whether DSL execution is supported by translation or by interpretation. However, in both cases it is common to find a runtime support in terms of the domain-specific library or framework that results of the commonality analysis performed during the domain engineering activities. In this way, translators generate code that makes use of this domain-specific framework, while interpreters directly perform the operations on this framework required to carry out the execution (e.g., on-the-fly object instantiation and assembling, method invocation on the instantiated objects. etc.)

2.1.3 Application Development Activities

Once a suitable DSL is available, it can be used by developers and by domain experts to develop applications in the domain. As indicated earlier, in an idealized situation a DSL could free developers of application construction and maintenance in favor of domain-experts. However, a more realistic approach promotes the tight collaboration or both types of stakeholders in interdisciplinary development teams.

2.2 Development Process Dynamics

In a realistic DSL-based development process, DSLs must evolve according the expressive needs of domain experts. In this way, new expressive needs that are made apparent during application development imply the extension of the DSL infrastructure to accommodate these needs. As a consequence, DSL-based development processes are iterative and incremental in nature, promoting the iterative enhancement and the incremental extension of the DSL as a consequence of application construction.

The iterative and incremental nature of DSL construction shifts the recurrent software maintenance and evolution concerns to the language design and implementation level. Indeed, DSL maintenance and evolution is a keystone aspect of the DSL approach [42]. In particular, maintenance and evolution of dynamic semantics related components (translators and interpreters) are particularly critical due to the semantic modularity problem: local changes in a language can imply global changes in the associated processors [25, 45]. In this way, since DSLs are exposed to constant evolution and enhancement, the construction of their

processors (translators, interpreters) can take benefit of modularization techniques used in semantic specification and language processor construction [8, 15, 18, 22].

2.3 Some DSL-based Experiences

The Speaker of this keynote has been involved in DSL-based development in several fields, including knowledge-based systems and e-learning:

- During the early nineties of the past century, the Speaker had the opportunity of working at the Intelligent Systems Research Group, led by Prof. José Cuena, one of the pioneers of knowledge-based systems and artificial intelligence in Spain. Instead of using general-purpose knowledge representation formalisms (e.g., rule-based systems) to build intelligent systems, Cuena promoted the provision of formalisms specially tailored to each application domain, adopting in this way the concept of DSL several years before to its popularization and applying it to the development of knowledge-based systems for real-time decision-making support (in particular, Cuena developed several decision-making intelligent systems in the fields of traffic management and watershed management) [5]. Cuena's systems usually included specialized knowledge editors, which supported specialized knowledge-representation languages, and which were used directly by domain experts to provide the knowledge required by the system, as well as inference engines (interpreters of the aforementioned languages) able to execute the provided knowledge models. As a consequence of these experiences, Cuena's team developed an environment called KSM (*Knowledge Structured Management*), which was used to build this kind of knowledge-based systems [26]. In this sense, KSM could be understood as a sort of language workbench specialized in the field of knowledge-based systems.
- In 1998 the Speaker moved to UCM, where he was involved in several research projects concerning information management in e-learning. Indeed, e-learning is a field rich in examples concerning special-purpose languages (e.g., *educational modeling languages* intended to be used by instructors to describe the design of their courses [20]). At UCM, the Speaker took contact with the works done by the team of Prof. Fernández-Valmayor in the production and maintenance of complex educational hypermedia applications for second language learning [9]. In order to facilitate application maintenance, contents and other critical structures of the application were provided by domain experts (experts in philology) as structured documents marked with a SGML-based notation specific for the applications being constructed. These documents were subsequently processed in order to automatically update the application. Building on this idea, during the first decade of the present century, the Speaker worked on an approach for the development of educational (and other content intensive) systems based on the explicit formulation of XML-based DSLs, as well as in the construction of application generators for the resulting DSLs [37, 38].

3 Language-Oriented Architectures

The DSL approach promotes the use of language-driven techniques in the provision of domain-specific development tools. Indeed, a DSL is intended to describe different aspects of an application, but the internals of this application do not necessarily include language processing components. It can be even true when a DSL interpreter is used, since in this case the interpreter can bind the DSL description into an instantiation of an underlying runtime framework, and then to activate this instantiation by invoking suitable methods in the resulting objects. On the contrary, *Language-oriented Architectures* (LOAs), an approach

that the Speaker's team is experiencing at UCM, promotes to upgrade language processing techniques to the core of conventional applications.

3.1 Anatomy of a LOA

A LOA encourages the organization of an application as a coordinated set of language processors. Each processor operates on an information domain (e.g., a type of XML documents, an object-oriented class model, an even stream in an interactive application, etc.) and consists of:

- A *reader* that is able to read information instances in this information domain. As a consequence, it transforms these instances into sequences of tokens. Therefore, the reader plays the role of a scanner in a conventional language processor.
- A syntax-directed processor, which processes the sequence of tokens directed by its underlying syntactic structure. It is analogous to a parser extended with semantic actions.

3.2 Language Implementation Tools as General-Purpose Development Tools

There are not significant differences among the syntax-directed processor of a LOA and the corresponding component in a conventional language processor, since both components act on sequences of tokens. Indeed, readers in a LOA adapt information domains to the requirements of classic syntax-directed language processing models [2]. As a consequence, language implementation tools (like parse generators or attribute grammar-based tools) [2, 28], traditionally used in specialized fields like compiler construction, adopt a new and unexpected role as general-purpose development tools for applications architected according to the LOA principle. Indeed, a LOA-conforming application can be developed in terms of:

- A set of readers, one for each language processor that integrate the application. Although the provision of these readers can require conventional programming, in many information domains it will be possible to take advantage of the information structure (e.g., the markup in an XML document, the structure of an object model) to easily produce these readers by customizing generic ones using high-level customization specifications.
- A set of syntax-directed specifications (e.g., YACC, JavaCC or ANTLR translation schemes, LISA attribute grammars, etc.). These specifications are keystone assets in the development of the application, since they will serve to automatically generate the syntax-directed processors by using suitable language implementation tools (YACC, ANTLR, LISA ...)
- Additional conventional software components used to support the semantic actions invoked by the syntax-directed translators.

The resulting development approach to LOA applications has been called *grammatical* approach by the Speaker's team at UCM, since it strongly relies on the use of grammarware as primary development support. In addition, contrarily to other approaches that promote the application of grammatical formalisms specially tailored to each application domain (e.g., tree grammars in the XML field, graph grammars in model-driven engineering scenarios), the grammatical approach promotes the use of classic string-oriented grammars. The adaptation to each information domain is, in turn, delegated to suitable readers (in other words, to apply the grammatical approach to a new information domain, the first thing to do is to decide how to read information elements in this domain). Once it is done, it is possible to facilitate the application of the grammatical approach by devising specific grammar-based

notations for each particular domain, as well as ways of transforming these notations into the basic model.

3.3 Experiences with the LOA Approach

The Speaker's team at UCM is currently working with the characterization and generation of several kinds of processors to be integrated in applications organized according to LOAs:

- XML *syntax-directed processors*. The team has devised several models for processing XML documents based on the combination of XML stream-oriented processing frameworks (SAX and STaX) with parser generation tools (JavaCC and CUP) [34]. They also have defined a specific grammar-based notation for describing XML processing tasks based on attribute grammars (XLOP: *XML Language Oriented Processing*), together with its transformation into the processing framework integrated by STaX + CUP [33].
- JSON *syntax-directed processors*. The work concerning JSON processing is similar to the work concerning XML. In this case the parser generator tool was ANTLR [32]. Currently the team is working on JLOP (JSON Language Oriented Processing) a meta-tool similar to XLOP.
- Model transformation based on attribute grammars. Attribute grammars in this proposal operate on spanning trees of object networks serialized by suitable readers (a prototype implementation is described in [13]).
- Syntax-directed processors of event streams in interactive applications. The idea is similar to the described in [27], and oriented to generate controllers for interactive applications from attribute grammar-based specifications (see [36] for an alternative approach based on structural operational semantics specifications)

In addition, the team is also working on the definition of a LOA for @note, a RIA for the collaborative annotation of digitized literary text [3]

4 Closing

This keynote has reviewed two different approaches for bringing computer language design and implementation technologies to mainstream software development scenarios: DSLs, which are oriented to provide domain-specific development tools, and LOAs, which promotes to architect applications as coordinated sets of language processors. Both approaches are oriented to enhance the production and maintenance of applications by providing specification of components to higher levels than the enabled by general-purpose programming languages: domain-specific notations in the case of DSLs, grammar-oriented specifications in the case of LOAs.

Currently the Speaker's team is working on refining the concept of LOA and in applying it to real case-studies in the field of digital humanities. Concerning future work, the relationships and synergies of DSL and LOA-based approaches arise as a very promising field. Also a more in-depth insight concerning the relationships of these language-driven approaches with model-driven ones appears to be a promising concern to explore.

Acknowledgements. Work partially supported by project grant TIN2010-21288-C02-01.

References

1 Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

2 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2007.

3 Juan Cigarrán-Recuero, Joaquín Gayoso-Cabada, Miguel Rodríguez-Artacho, María-Dolores Romero-López, Antonio Sarasa-Cabezuelo, and José-Luis Sierra. Assessing semantic annotation activities with formal concept analysis. *Expert Syst. Appl.*, 41(11):5495–5508, 2014.

4 Steve Cook, Gareth Jones, Stuart Kent, and Alan Wills. *Domain-specific Development with Visual Studio Dsl Tools*. Addison-Wesley Professional, 2007.

5 José Cuena. Architectures for second generation knowledge based systems. In *Proceedings of the International Summer School on Advanced Topics in Artificial Intelligence*, pages 373–403, London, UK, UK, 1992. Springer-Verlag.

6 Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

7 Fergal Dearle. *Groovy for Domain-Specific Languages*. Packt Publishing, 1st edition, 2010.

8 Dominic Duggan. A mixin-based, semantics-based approach to reusing domain-specific programming languages. In Elisa Bertino, editor, *ECOOP*, volume 1850 of *Lecture Notes in Computer Science*, pages 179–200. Springer, 2000.

9 Baltasar Fernandez-Manjon and Alfredo Fernandez-Valmayor. Improving world wide web educational uses promoting hypertext and standard general markup language content-based features. *Education and Information Technologies*, 2(3):193–206, 1997.

10 Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

11 Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 3 edition, 2008.

12 Debasish Ghosh. *DSLs in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

13 Juan-Pablo Gracia. Marco para la transformacion de modelos basado en gramaticas de atributos. Master's thesis, Facultad de Informatica, UCM, 2010.

14 Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Eclipse Series. Pearson Education, 2009.

15 Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

16 Capers Jones. End-user programming. *IEEE Computer*, 28(9):68–70, 1995.

17 Kio C. Kang, Sholom G. Cohen, Janes A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

18 Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. *Acta Inf.*, 31(7):601–627, October 1994.

19 Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

20 Rob Koper and Bill Olivier. Representing the learning design of units of learning. *Educational Technology & Society,*, 7(3):97–111, 2004.

21 Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

22 Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA, 1995. ACM.

23 Sjouke Mauw, Wouter T. Wiersma, and Tim A. C. Willemse. Language-driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14(6):625–663, 2004.

24 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.

25 Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

26 Martín Molina, José-Luis Sierra, and José Cuena. Reusable knowledge-based components for building software applications: A knowledge modelling approach. *International Journal of Software Engineering and Knowledge Engineering*, 9(3):297–317, 1999.

27 Albert Nymeyer. A grammatical specification of human-computer dialogue. *Comput. Lang.*, 21(1):1–16, 1995.

28 Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, June 1995.

29 Paolo Perrotta. *Metaprogramming Ruby: Program Like the Ruby Pros*. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2010.

30 Ayende Rahien. *DSLs in Boo: Domain Specific Languages in .Net*. Manning Pubs Co Series. Manning Publications Company, 2010.

31 Peter H. Salus. *Little Languages and Tools*. Macmillan Technical Publishing, 1st edition, 1998.

32 Antonio Sarasa-Cabezuelo and José-Luis Sierra. Grammar-driven development of json processing applications. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *FedCSIS*, pages 1545–1552, 2013.

33 Antonio Sarasa-Cabezuelo and José-Luis Sierra. The grammatical approach: A syntax-directed declarative specification method for xml processing tasks. *Comput. Stand. Interfaces*, 35(1):114–131, January 2013.

34 Antonio Sarasa-Cabezuelo, Bryan Temprado-Battad, Daniel Rodriguez-Cerezo, and José-Luis Sierra. Building xml-driven application generators with compiler construction tools. *Comput. Sci. Inf. Syst.*, 9(2):485–504, 2012.

35 Michael L. Scott. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2009.

36 José-Luis Sierra, Baltasar Fernández-Manjón, and Alfredo Fernández-Valmayor. A language-driven approach for the design of interactive applications. *Interacting with Computers*, 20(1):112–127, 2008.

37 José-Luis Sierra, Alfredo Fernández-Valmayor, and Baltasar Fernández-Manjón. A document-oriented paradigm for the construction of content-intensive applications. *Comput. J.*, 49(5):562–584, 2006.

38 José-Luis Sierra, Alfredo Fernández-Valmayor, and Baltasar Fernández-Manjón. From documents to applications using markup languages. *IEEE Softw.*, 25(2):68–76, March 2008.

39 Mark A. Simos. Organization domain modeling (odm): Formalizing the core domain modeling life cycle. In *Proceedings of the 1995 Symposium on Software Reusability*, SSR '95, pages 196–205, New York, NY, USA, 1995. ACM.

40 Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

41 Richard N. Taylor, Will Tracz, and Lou Coglianese. Software development using domain-specific software architectures: Cdrl a011—a curriculum module in the sei style. *SIGSOFT Softw. Eng. Notes*, 20(5):27–38, December 1995.

42 Arie van Deursen and Paul Klint. Little languages: Little maintenance. *Journal of Software Maintenance*, 10(2):75–92, March 1998.

43 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

44 Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

45 Philip Wadler. How to declare an imperative. *ACM Comput. Surv.*, 29(3):240–263, September 1997.

46 Tobias Walter, Fernando Silva Parreiras, and Steffen Staab. Ontodsl: An ontology-based framework for domain-specific languages. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 408–422, Berlin, Heidelberg, 2009. Springer-Verlag.

An Overview of Open Information Extraction

Pablo Gamallo

CITIUS
Universidade de Santiago de Compostela
Galiza, Spain
pablo.gamallo@usc.es

Abstract

Open Information Extraction (OIE) is a recent unsupervised strategy to extract great amounts of basic propositions (verb-based triples) from massive text corpora which scales to Web-size document collections. We will introduce the main properties of this extraction method.

1998 ACM Subject Classification I.2.6 Learning: Knowledge acquisition

Keywords and phrases information extraction, natural language processing

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.13

Category Invited Talk

1 Introduction

Recent advanced techniques in Information Extraction aim to capture shallow semantic representations of large amounts of natural language text. Shallow semantic representations are conceived as an intermediate level in the process of structuring textual information. In further processes, shallow semantics can be applied to more complex semantic tasks involved in text understanding, such as textual entailment, filling knowledge gaps in text, or integration of text information into background knowledge bases.

There is an emerging field in Information Extraction interested in applying shallow semantics techniques, namely Machine Reading [7], Learning by Reading [4], or Discovery Information[1]. In this new field, the different techniques used to perform the extraction are not bound by a pre-specified schema of information, but rather they discover relational or categorial structure automatically from given unstructured data using unsupervised strategies.

One of the most used strategies in this new field aimed at discovering shallow semantic representations is known as Open Information Extraction (OIE). The main goal of OIE is to extract a large set of verb-based *triples* (or *propositions*) from unrestricted text. An OIE system reads in sentences and rapidly extracts one or more textual assertions, consisting in a verb relation and two arguments, which try to capture the main relationships in each sentence [3]. Wu and Weld [13] define an OIE system as a function from a document d, to a set of triples, $(arg1, rel, arg2)$, where $arg1$ and $arg2$ are verb arguments and rel is a textual fragment (containing a verb) denoting a semantic relation between the two verb arguments. Unlike other relation extraction methods focused on a predefined set of target relations, the Open Information Extraction paradigm is not limited to a small set of target relations known in advance, but extracts all types of (verbal) binary relations found in the text. The main general properties of OIE systems are the following: (i) they are domain independent,

[1] http://www.aha-workshop.de/


© Pablo Gamallo;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 13–16

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(ii) they rely on unsupervised extraction methods, and (iii) they are scalable to large amouts of text [6].

2 Basic Propositions

An OIE system extracts different triples *(arg1, rel, arg2)*, representing basic propositions or assertions from each sentence in a text. Propositions are defined as coherent and non over-specified pieces of basic information. Consider for example the sentence:

> *In May 2010, the principal opposition parties boycotted the polls after accusations of vote-rigging.*

An OIE system must transform this sentence into a set of triples:

(“*the principal opposition parties*”, “*boycotted*”, “*the polls*”)
(“*the principal opposition parties*”, “*boycotted the polls in*”, “*May 2010*”)
(“*the principal opposition parties*”, “*boycotted the polls after*”, “*accusations of vote-rigging*”)

They represent coherent and non over-specified items of information organized by means of three different relations: “boycotted”, “boycotted the polls in”, and “boycotted the polls after”. Incoherent extractions would be for, instance, the following triples:

(“*parties boycotted*”, “*after*”, “*accusations of vote-rigging*”)
(“*May 2010*”, “*boycotted*”, “*the polls*”)

They are incoherent because, on the one hand, “parties boycotted” cannot be considered as the argument of any relation and, on the other hand, “May 2010” should not be taken as the subject argument of “boycotted”.

Examples of over-specified triples extracted from the same sentences are:

(“*the principal opposition parties*”, “*boycotted the polls in May 2010 after accusations of*”, “*vote-rigging*”)
(“*the principal opposition parties*”, “*boycotted*”,
“*the polls in May 2010 after accusations of vote-rigging*”)

They are over-specified since both the relation “boycotted the polls in May 2010 after accusations of” and the argument “the polls in May 2010 after accusations of vote-rigging” convey too much information to be useful in further semantic tasks, such as semantic entailment or ontology population.

3 Overview of Different OIE Systems

A great variety of OIE systems has been developed in recent years. They can be organized in two broad categories: those systems requiring automatically generated training data to learn a classifier and those based on hand-crafted rules or heuristics. In addition, each system category can also be divided in two subtypes: those systems making use of shallow syntactic analysis (PoS tagging and/or chunking), and those based on dependency parsing. In sum, we identify four categories of OIE systems:

(1) Training data and shallow syntax: The first OIE system, TextRunner [2], belongs to this category. Two more recent versions of TextRunner, also using training data and shallow syntactic analysis, are ReVerb [9] and R2A2 [8]. Another system of this category is WOE^{pos} [13] whose classifier was trained with corpus obtained automatically from Wikipedia.

(2) Training data and dependency parsing: These systems make use of training data represented by means of dependency trees: WOE^{dep} [13] and OLLIE [12].

(3) Rule-based and shallow syntax: They rely on lexical-syntactic patterns hand-crafted from PoS tagged text: ExtrHech [15] and LSOE [14].

(4) Rule-based and dependency parsing: They make use of hand-crafted heuristics operating on dependency parses: ClauseIE [6], CSD-IE [5], KrakeN [1], and DepOE [11].

4 Evaluation and Conclusions

According to the experiments and evaluation we have performed [10], we showed that the rule-based systems perform better than the classifiers based on automatically generated training data. This is in accordance with previous work reported in [6, 5]. Moreover, the systems based on dependency analysis improve over those relying on shallow syntax (TextRunner and ReVerb). It follows that it is not necessary to make use of training data and machine learning strategies to perform open information extraction. We just require a dependency parser and a set of basic rules transforming the parses into triples.

Acknowledgments. This work was partially supported by Projects Celtic, Plastic (Innterconecta, FDTI), and HPCPLN (Xunta de Galicia).

References

1 Alan Akbik and Alexandre Loser. Kraken: N-ary facts in open information extraction. In *Joint Workshop on Automatic Knowledge Base Construction and Web-scale Knowledge Extraction*, pages 52–56, 2012.

2 Michele Banko, , and Oren Etzioni. The tradeoffs between open and traditional relation extraction. In *Annual Meeting of the Association for Computational Linguistics*, 2008.

3 Michele Banko, Michael J Cafarella, Stephen Soderland, Matt Broadhead, and Oren Etzioni. Open information extraction from the web. In *International Joint Conference on Artificial Intelligence*, 2007.

4 K. Barker, B. Agashe, S. Chaw, J. Fan, N. Friedland, M. Glass, J. Hobbs, E. Hovy, D. Israel, D.S. Kim, et al. Learning by reading: A prototype system, performance baseline and lessons learned. In *Proceeding of Twenty-Second National Conference of Artificial Intelligence (AAAI 2007)*, 2007.

5 Hannah Bast and Elmar Haussmann. Open information extraction via contextual sentence decomposition. In *ICSC 2013*, pages 154–159, 2013.

6 Luciano Del Corro and Rainer Gemulla. Clausie: Clause-based open information extraction. In *Proceedings of the World Wide Web Conference (WWW-2013)*, pages 355–366, Rio de Janeiro, Brazil, 2013.

7 Oren Etzioni, Michele Banko, and Michael J. Cafarella. Machine reading. In *AAAI Conference on Artificial Intelligence*, 2006.

8 Oren Etzioni, Anthony Fader, Janara Christensen, Stephen Soderland, and Mausam. Open information extraction: the second generation. In *International Joint Conference on Artificial Intelligence*, 2011.

9 Anthony Fader, Stephen Soderland, and Oren Etzioni. Identifying relations for open information extraction. In *Conference on Empirical Methods in Natural Language Processing*, 2011.

10 Pablo Gamallo and Marcos Garcia. Open information extraction based on argument structure detection. In *(submitted paper)*, 2014.

11 Pablo Gamallo, Marcos Garcia, and Santiago Fernández-Lanza. Dependency-based open information extraction. In *ROBUS-UNSUP 2012: Joint Workshop on Unsupervised and Semi-Supervised Learning in NLP at the 13th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2012)*, Avignon, France, 2012.

12 Mausam, Michael Schmitz, Stephen Soderland, Robert Bart, and Oren Etzioni. Open language learning for information extraction. In *Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 523–534, 2012.

13 Fei Wu and Daniel S. Weld. Open information extraction using wikipedia. In *Annual Meeting of the Association for Computational Linguistics*, 2010.

14 Clarissa C. Xavier, Marlo Souza, and Vera S. de Lima. Open information extraction based on lexical-syntactic patterns. In *Brazilian Conference on Intelligent Systems*, pages 189–194, 2013.

15 Alisa Zhilla and Alexander Gelbukh. Comparison of open information extraction for engish and spanish. In *Dialogue 2014*, 2014.

Part II

Program Comprehension

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Conclave: Writing Programs to Understand Programs

Nuno Ramos Carvalho[1], José João Almeida[1], Maria João Varanda Pereira[2], and Pedro Rangel Henriques[1]

1 **Departamento de Informática/CCTC**
Universidade do Minho, Braga, Portugal
{narcarvalho,jj,prh}@di.uminho.pt
2 **Escola de Tecnologia e Gestão/CCTC**
Instituto Politécnico de Bragança, Bragança, Portugal
mjoao@ipb.pt

Abstract

Software maintainers are often challenged with source code changes to improve software systems, or eliminate defects, in unfamiliar programs. To undertake these tasks a sufficient understanding of the system, or at least a small part of it, is required. One of the most time consuming tasks of this process is locating which parts of the code are responsible for some key functionality or feature.

This paper introduces Conclave, an environment for software analysis, that enhances program comprehension activities. Programmers use natural languages to describe and discuss the problem domain, programming languages to write source code, and markup languages to have programs talking with other programs, and so this system has to cope with this heterogeneity of dialects, and provide tools in all these areas to effectively contribute to the understanding process. The source code, the problem domain, and the side effects of running the program are represented in the system using ontologies. A combination of tools (specialized in different kinds of languages) create mappings between the different domains. Conclave provides facilities for feature location, code search, and views of the software that ease the process of understanding the code, devising changes. The underlying feature location technique explores natural language terms used in programs (e.g. function and variable names); using textual analysis and a collection of Natural Language Processing techniques, computes synonymous sets of terms. These sets are used to score relatedness between program elements, and search queries or problem domain concepts, producing sorted ranks of program elements that address the search criteria, or concepts respectively.

1998 ACM Subject Classification D.2.7 Distribution, Maintenance, and Enhancement – Restructuring, reverse engineering, and reengineering

Keywords and phrases software maintenance, software evolution, program comprehension, feature location, concept location, natural language processing

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.19

1 Introduction

Reality shifts, bug fixes, updates or introduction of new features often require source code changes. These software changes are usually undertaken by software maintainers that may not be the original writers of the code, or may not be familiar with the code anymore. In order to carry out these changes, programmers need to first understand the source code [41]. This task is probably the main challenge during software maintenance activities [10]. The


© Nuno Ramos Carvalho, José João Almeida, Maria João Varanda Pereira, and Pedro Rangel Henriques; licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 19–34

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

programmer is able to understand the program when he or she can explain the source code, and relate the code with the concepts in its problem domain [3].

Software reverse engineering is a process that tries to infer how a program works by analyzing and inspecting its building blocks and how they interact to achieve their intended purpose. Many of the techniques used in reverse engineering rely on mappings between human oriented concepts (described using natural language), and program elements (implemented using programming languages) [33]. These are often used to locate which parts of the program are responsible for addressing specific domain concepts [3], and are usually referred in the literature as feature location techniques [11].

Natural languages are used to describe and discuss real world problems, and programming languages are used to develop computer programs that address these problems. Although, programming languages have unambiguous grammars and limit the sentences that can be used to write software, still give some degree of freedom to the programmer to use natural language terms (e.g. program identifiers, constant strings or comments). These terms can give clues about which concepts the source code is addressing, and the meaningfulness of these terms can have a direct impact on future program comprehension tasks [24]. Most of the programming communities promote the use of best practices and coding standards that usually include rules and naming conventions that improve the quality of terms used (e.g. the *"Style Guide for Python Code"*[1]). Feature location techniques that exploit such elements and possible relations between different language domains are typically described as textual analysis, often combined with static analysis [11].

This paper introduces CONCLAVE[2], a system of tools for software analysis. The main goal of this system is to provide programmers with insight and information about software packages to enhance program understanding activities and ease software maintenance tasks. The system provides a set of facilities for searching and a feature location technique, that measures semantic relatedness between source code elements, and elements supplied by the maintainer as query searches. Several views provide mappings between source code and real world concepts, facilitating feature location activities. The underlying feature location technique uses source code static analysis to extract data from source code (e.g program identifiers, function definitions). The extracted data is loaded to an ontology that represents the program. Other ontologies can be added to the system if available (e.g. the problem domain ontology, dynamic traces information). Using a set of Natural Language Processing (NLP) techniques and textual analysis, *kind-of* Probabilistic Synonymous Sets (kPSS) are computed for every element present in the ontologies, and a scoring function is used to measure the semantic relatedness[3] between them. The main output of this tool is a list of ranks – sorted by relevance – of program elements that are prone to address some specific real world domain concept. The system also provides a Domain Specific Language (DSL), for writing search queries.

The next section introduces the CONCLAVE system, including a brief description about the major stages of the system workflow. Section 3 describes in more detail some tools and results that can be produced using the system. Section 4 presents related work, and introduces some state-of-the-art techniques for feature location. Section 5 describes the experimental validation held to do a preliminary evaluation of some CONCLAVE tools and

[1] Available from: `http://www.python.org/dev/peps/pep-0008/` (Last accessed: 29-01-2014).
[2] CONCLAVE website: `http://conclave.di.uminho.pt` (Last accessed: 10-03-2014).
[3] In ontologies the term similarity is used to refer how similar two concepts are, and is usually based on a hierarchy of *is-a* relations, in the context of this work concepts can be related in many ways, hence the adoption of the term relatedness.

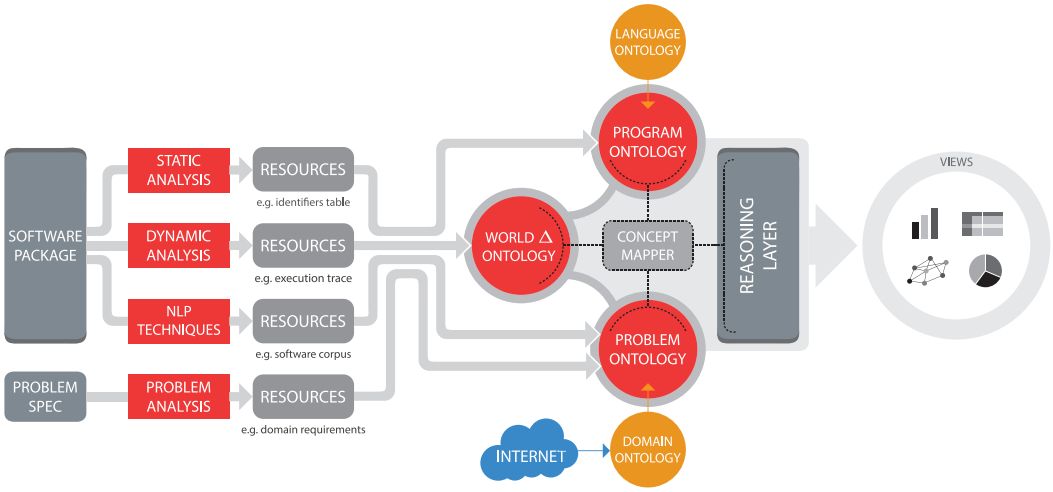


Figure 1 Overview of the major stages of the Conclave system workflow.

techniques, including results discussion. Finally, Section 6 presents some final remarks and trends for future work.

2 Conclave Architecture Overview

The Conclave environment provides a set of tools to perform software analysis. The main system workflow is divided in three stages: (a) collecting data; (b) processing collected data and loading ontologies; and, (c) reasoning about data in the ontologies and providing views of computed information. Figure 1 illustrates this workflow, and the next sections describe in more detail the different stages. All the tools implemented in the context of this system are modular (or work as plugins), and some provide web-services, so that they can be used as standalone applications, or composed together to create more complex applications or other workflows.

2.1 Collecting Data

This is the first stage of the main workflow; its goal is to collect data from a software package, and any kind of problem specification if available. It takes as input the complete package (and other available documents) and produces as output an heterogeneous collection of resources. The processing tools involved in this stage can use different type of analysis: static source code analysis (e.g. parsing code to extract identifiers and static call graphs), dynamic analysis (e.g. execution traces), Natural Language Processing (NLP) approaches (e.g. processing non-source code content for domain vocabulary), etc.

Any analysis can be used to collect information, and produce a resource. In the context of this work, some tools were implemented to provide some initial data to the system and contribute to PC in general, here are some examples:

Conc-clang: is a static analysis tool, based on the clang compiler library [22] for gathering identifiers and static functions calls information for `C/C++` programs;

Conc-antlr: is a static analysis tool, based on the ANTLR parser generator framework [29], for gathering program identifiers information for `Java` programs;

DMOSS: is a toolkit for software documentation assessment. It produces an attribute tree representation of a software package, and other software related resources like the *documentation corpus* that is used later to create a initial version of the problem ontology. For more details about this framework refer to [8].

The heterogenous set of tools used during this stage produce a multitude of resources in distinct formats. In order to take advantage of all these resources, all the information needs to be conveyed to a common format, more suitable for querying and processing. Ontologies were adopted as a common target format. Building ontologies from collected data is done during the second stage, which is discussed in the next section.

2.2 Normalizing Information, Populating Ontologies

The main goal of this stage is to convey the data collected during the previous stage into the system ontologies. The input of this stage is a collection of resources, and the output is a set of populated ontologies. Usually three ontologies are populated for each software package:

Program Ontology: abstract representation of some key program elements (e.g. methods, functions, variables, classes);

Problem Ontology: concepts and relations in the problem domain;

World $\triangle$ Ontology: runtime effects of executing the program (e.g. program run traces).

There are two important details about this stage. The first one is the format and technology chosen to store the ontologies. A RDF based triple-store technology was adopted to store the data. This allows for a scalable and efficient method for performing storing and querying operations, and also allows to export the data in several community accepted ontology formats (e.g. OWL, RDF/XML, Turtle) [19, 21]. Querying facilities are also readily available; for instance, SPARQL is a querying domain specific language for RDF triple-stores [30, 32].

Although these technologies provide scalable and efficient environments for handling information, development wise, they are far from the abstraction desired by the applications level implementation. To overcome this problem the Ontology ToolKit (OTK)[4] was developed, which provides an abstraction layer on top of the RDF technology, to develop ontology-*aware* applications. In practice, when applications developers want to perform an ontology related operation, instead of using triple-store low level primitives, they can use the abstraction layer. To motivate for the development of this abstract framework, consider the modern Object-Relational Mappers (ORM) in the context of relational databases. Which provide an abstraction layer and interface for programming languages to handle data (stored in databases) as objects, allowing the development of applications regardless of the underlying database technology used [20].

The second important detail is the data semantic shift. Resources tend to produce raw data, but the data stored in the ontologies conveys a richer semantic. Most resources require a specific tool to read the resource data, and translate it to information that is ready to store in the ontology, i.e. follows the semantic defined by the ontology. OTK has also proven useful to implement this family of tools.

[4] Implemented as a set of libraries for the Perl programming language.

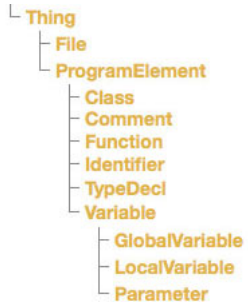


Figure 2 Program ontology sub-set of the class hierarchy.

A simple example to illustrate the previously discussed details follows. Imagine the CONC-CLANG tool was used to process a `C` source code file, included in a software package. The raw output of this tool is a set of lines that look something like[5]:

```
Function,source.c::add::6,add,,source.c,6,8
```

This line by itself conveys small to none semantic of the data being included in the final resource. In loose english this line states that: *"in the 'source.c' file there is a 'Function' definition which has a identifier represent by the string 'add' that starts in line '6' and ends in line '8' "*, and this is the kind of semantic that needs to be conveyed to the ontological representation of the program. The Program Ontology has a class to represent instances of elements that are functions in the source code, another for identifiers, and the line numbers are stored as data proprieties[6]. To illustrate the use of OTK, the following snippet illustrates a simplified version of the required code to load this information to the Program Ontology.

```
use OTK;
my $ontology = OTK->new($pkgid, 'program');
$ontology->add_instance('add', 'Function');
$ontology->add_instance('add', 'Identifier');
$ontology->add_data_prop('add', 'hasLineBegin', 6, 'int');
$ontology->add_data_prop('add', 'hasLineEnd', 8, 'int');
$ontology->add_obj_prop('add', 'inFile', 'source.c');
```

The Program Ontology used is in line with other authors' proposed descriptions (e.g. [35,43,44]). This also eases future integration processes with other tools that followed similar approaches. Figure 2 illustrates a subset of the class hierarchy exported to OWL. Once all the data is stored in the ontologies, the reasoning layer can be used to relate information gathered from different elements and domains to build semantic bridges between elements. More details about this stage are discussed in the next section.

2.3 Reasoning and Views

During this stage more knowledge about the system is built and provided to the system end-user. The tools in this stage use as input the ontologies built during the previous stage, and generally fall in one of the two categories, either they: (a) process information to

[5] More examples available in the tool website: `http://conclave.di.uminho.pt/clang` (Last accessed: 27-01-2014).

[6] Although a triple-store RDF approach is used to store the actual information, we are using OWL vocabulary and specification to make clear the aimed semantics for the program representation [2].

Table 1 Some ABCMIDI package characteristics.

Total Files	Size (KLOC[9])	Total Ids.	Multi-word Ids.
86	∼ 33	3437	2142 (62%)

compute new information and knowledge about the system – usually in this case the tool output is new content added to the ontologies; or (b) information or knowledge suitable for visualization is built – in this case the output is a view about a particular aspect of the package system.

Querying the ontology, and adding information if necessary, can easily be done using the OTK framework. Also note that the tools in this stage are language agnostic, in the sense that data about the source code (language dependent) has already been gathered, and OTK tools do not depend anymore on the source language. For example, if a tool processes identifiers, to get a list of the program identifiers simply query the Program Ontology using OTK, as follows:

```
use OTK;
my $ontology = OTK->new($pkgid, 'program');
my @identifiers = $ontology->get_instances('Identifier');
```

CONCLAVE-MAPPER, one of the tools described in the next section, is an example of tools that are used during this stage.

3 CONCLAVE Quick Tour

The goal of this section is to illustrate some practical applications of the CONCLAVE system. Two tools are introduced, and some features are illustrated. The software analyzed and used in the next examples in this section is ABCMIDI (version 2012.12.25)[7], a package that provides a set of tools to convert ABC[8] files to the MIDI format. Table 1 presents some characteristics about this software package.

Figure 3 illustrates the CONCLAVE web interface front page, the system is divided in blocks, and most of the applications use resources produced by other blocks. The tools presented in this section address two popular problems in the context of program comprehension: (1) splitting multi-term program identifiers, and (2) mappings between program elements and real world concepts.

3.1 Splitting Identifiers: LINGUA-IDSPLITTER

LINGUA-IDSPLITTER (henceforth abbreviated LIDS) is a simple and fast algorithm that addresses the problem of splitting *soft words*[10] that compose an identifier. It handles abbreviations, acronyms, or any type of linguistic short-cuts (for example, use only the first letter of a word). The algorithm calculates a ranked list of all the possible splits for an identifier, based on a set of dictionaries, and the top entry in the rank is proposed as the correct split.

[7] Available from `http://abc.sourceforge.net/abcMIDI/` (Last accessed: 11-03-2014).
[8] A text notation to represent music.
[9] Thousands Lines of Code.
[10] Usually refers to words that are combined together to create an identifier without using an explicit mark between them (e.g., *"timesort"*) [24].

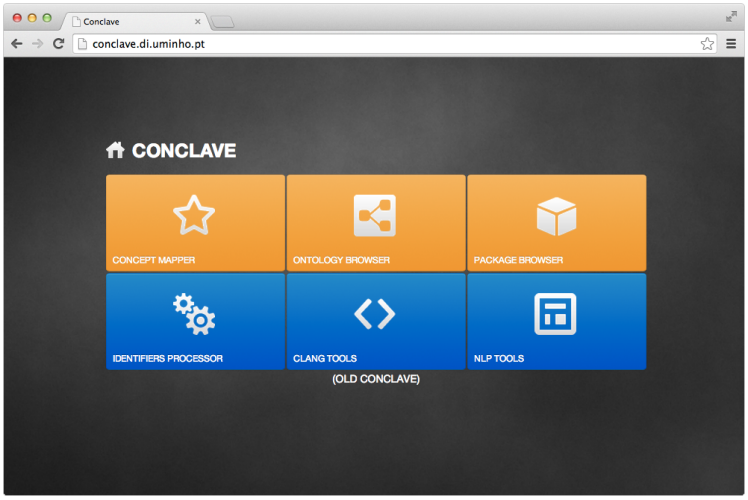


Figure 3 CONCLAVE system web interface front page, main applications are divided in blocks.

Besides the actual split, the result includes the set of full terms that compose the identifier, in case abbreviations were used for example. This technique can use an arbitrary set of dictionaries, but one of the benefits introduced by this approach is the use of a software specific dictionary computed automatically from a software package corpus – also computed automatically and specific to each software package – using a combination of Natural Language Processing (NLP) techniques. This dictionary enables the algorithm to correctly handle identifiers splitting using arbitrary abbreviations or combinations of term specific to the application domain, not prone to be present in more general programming dictionaries. this technique can also cope with identifiers that use explicit marks, like underscores (e.g., *"time_sort"*) or the *CamelCase* notation (e.g., *"timeSort"*).

This tool is implemented as a Perl library that can be used in other tools and contexts, and is available for download in the official Perl library archive[11]. In the context of PC this is relevant when dealing with program identifiers (e.g., variable names, function names) that were created using a combination of abbreviation and words. Correctly splitting program identifiers has a direct impact on future programming comprehension techniques [24]. Even a simple program can have thousands of identifiers, undertake this task manually would be unfeasible, so the literature is rich on techniques to address this problem (e.g., [13, 14, 23]).

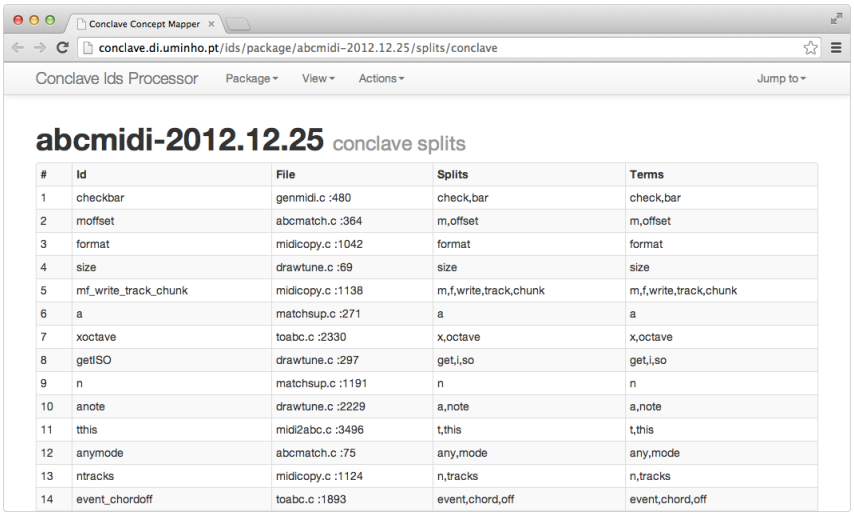
In the CONCLAVE system this tool retrieves the identifiers from the ontology, making it independent of the programming language used. The resulting split and expansion sets are loaded to the ontology and related to each identifier, so they are readily available for other applications to use. Table 2 illustrates the split and term sets computed for some ABCMIDI identifiers, and Figure 4 is a screenshot of CONCLAVE identifiers table for ABCMIDI including the splits and terms sets computed by LIDS.

LIDS algorithm for computing *soft splits* starts by computing all the possible valid strings that can be found starting at every position of the identifier. A string is considered valid if it is successfully found in any of the dictionaries being used. The next step is to build an automaton, with all the strings found, to calculate all the possible sequence of nodes (paths), that concatenate to rebuild the original identifier. The set of paths in the automaton defines

[11] Available from: `http://search.cpan.org/dist/Lingua-IdSplitter/` (Last accessed: 18-03-2014).

Table 2 ABCMIDI identifiers examples, and corresponding splits and abbreviation expansions.

Identifier	Splits	Expands
mrest	*m* | *rest*	{ *multibar*, *rest* }
timesig	*time* | *sig*	{ *time*, *signature* }
chan	*chan*	{ *channel* }



#	Id	File	Splits	Terms
1	checkbar	genmidi.c :480	check,bar	check,bar
2	moffset	abcmatch.c :364	m,offset	m,offset
3	format	midicopy.c :1042	format	format
4	size	drawtune.c :69	size	size
5	mf_write_track_chunk	midicopy.c :1138	m,f,write,track,chunk	m,f,write,track,chunk
6	a	matchsup.c :271	a	a
7	xoctave	toabc.c :2330	x,octave	x,octave
8	getISO	drawtune.c :297	get,i,so	get,i,so
9	n	matchsup.c :1191	n	n
10	anote	drawtune.c :2229	a,note	a,note
11	tthis	midi2abc.c :3496	t,this	t,this
12	anymode	abcmatch.c :75	any,mode	any,mode
13	ntracks	midicopy.c :1124	n,tracks	n,tracks
14	event_chordoff	toabc.c :1893	event,chord,off	event,chord,off

Figure 4 CONCLAVE interface to view the splits and terms set for all identifiers in the package being analyzed.

the set of string sequences that are candidates to be the identifier correct split. Next, the algorithm computes the score for each candidate, creating a rank, where the top element (the sequence with the higher score) is the resulting split.

The formula to calculate the score for a given sequence is analytically defined as:

$$score(S) = \frac{(\prod_{i=1}^{length(S)} factor(S_i)) + length(m)}{length(S)^2}$$

where the multiplicand of factors (a factor is calculated for each element in the sequence) plus the length of the longer string in the sequence, is normalized by the squared sequence length. Each factor is calculated according to the formula:

$$factor(s, t, w) = length(s) \times w$$

i.e., the length of the string found times the dictionary weight that validated the string.

The final result sets of terms are loaded to the program ontology, and related with the corresponding identifier. These sets of terms, can then be used to compute relatedness with words from other domains by other applications, like the one described in the next section. More details about this technique in [7].

3.2 Creating Mappings: CONCLAVE-MAPPER

CONCLAVE-MAPPER is an application that relies on data computed by other tools (see Sec. 2.1 and 2.2), to create relations between elements of any of the ontologies available for a given package. The input for this application is a set of ontologies, and either a

search query, or a mapping query; and the output is a sorted rank of element relations, or a mapping of element relations respectively. The Program Ontology represents the elements of the program, a software maintainer can ask the application to compute the relations between elements in the program and either a set of keywords provided in a search query, or elements in other ontologies (e.g. Problem Ontology) using a mapping query. In the first case, the result is a sorted rank of the program elements that are related with the keywords provided in the search query, and in the latter a matrix of relatedness score between the elements selected from both ontologies. Both approaches can be used to find which parts of the code are responsible for implementing a domain concept – feature location.

A rank is defined as a collection of entries, where each entry contains the semantic relatedness score, between the element and the search query. An element represents an instance in any ontology (if elements of the Program Ontology are being used all other data is also available: source file, begin and end line, identifier, etc.), so elements in different ontologies can be related. A map is defined as a matrix, with an element for each row and column; each cell in the matrix (besides its position information) contains the semantic relatedness measure score for the corresponding elements.

The application implements two main functions to compute each one of the available output types. The *locate* function creates a rank and has the following signature:

$$locate :: Query \rightarrow Rank$$

This function, given a query, computes a rank, by iterating over all the elements being analyzed (defined by the search query), and for each element computing a semantic relatedness score, and adding it to the rank as a new entry. The element set being searched and the scoring function are defined by the search query. The *mapping* function creates a map and has the following signature:

$$mapping :: Query \rightarrow Query \rightarrow ScoreFunction \rightarrow Map$$

This function, given two queries, and a scoring function, calculates a matrix of elements where each cell includes the relatedness score between the corresponding row and column element. This provides a matrix of relations between all selected (program, application domain, etc.) elements, that can be sorted by relevance. Figure 5 illustrates a possible view of these mappings, highlighting the best relevance ranking between the application domain and functions.

The *Query* type used before describes a query supplied by the user (a pre-defined set of queries is also available via the system interface). A DSL was developed to describe these queries (either search or mapping). Each query has at least three main components: (a) keywords; (b) domain and range constrains (e.g. search only functions, or variables); and, (c) the scoring function used to compute the relatedness score between the elements (all except keywords have default values). To illustrate the DSL some query examples are given below.

The following query performs a search for the words `"color"` and `"schema"`, but only analyses elements that are instances of the class `Function`.

```
[ word=color word=schema class=Function ]
```

The next query searches variables for the word `"color"`, and uses the `levenshtein` word distance algorithm [25], to compute the score. By default, the scoring function based on kPSS is used.

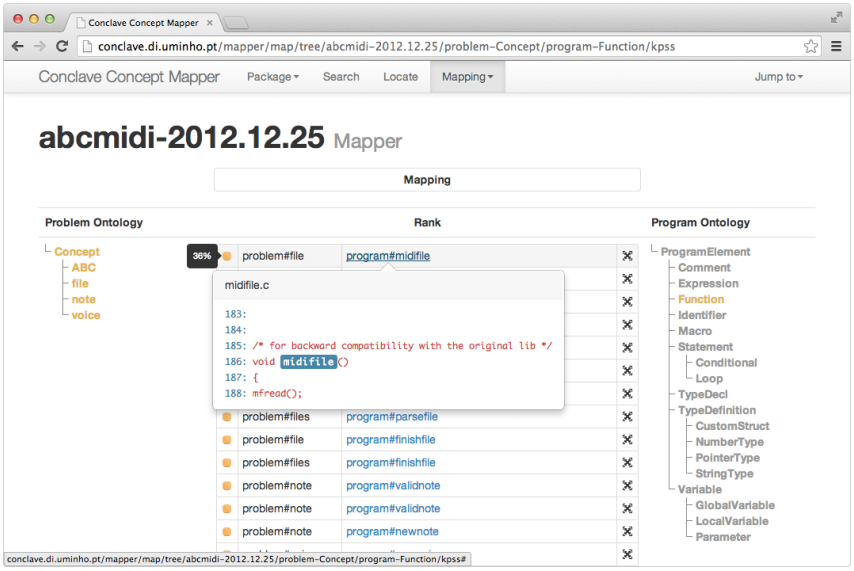


Figure 5 A mapping produced by CONCLAVE-MAPPER: on the left the Problem Ontology can be used to constrain the concepts being searched, on the right the Program Ontology can be used to constrain the range of which program elements are being analyzed, and in the center the resulting rank sorted by relevance (hovering the program element shows the corresponding zone in the file where the element appears).

```
[ word=color class=Variable score=levenshtein ]
```

The following query, searches all the functions, and for each function also considers all the elements that are related with that function by the relation `inFunction` (defined in the ontology):

```
[ word=color class=Function aggr=inFunction ]
```

The `inFunction` relation is used to link all the local variables and parameters to all the functions (or methods depending on programming language) where they are defined and used. In practice, the score for each element (function) is the average between computing the score for the element itself, and the score for every local variable and parameter defined in that function.

The score between two elements (or an element and a word) quantify how close they are semantically related. This score is used to sort the ranks computed by the *locate* function by relevance, or to highlight the cells that express close relatedness between elements in the matrixes computed with the *mapping* function.

The main scoring function available in the CONCLAVE system is the *kpss* function (used by default), and is based on kPSS, which defines a formalism to describe synonymous sets based on Probabilist Synonymous Sets (PSS) [5, 40]. These define synonymous sets based on statistical analysis of parallel corpora.

Once a kPSS is available for a pair of words, the relatedness score between these words can be calculated. The *kpss* function is used to compute this score (as a *Float*) and is defined as:

```
kpss        :: kPSS → kPSS → Float
kpss k1 k2 = ∑ [ min (prob x) (prob y) |
                 x ← flatten k1, y ← flatten k2, word x == word y ]
```

This function iterates over the flattened version of the kPSS, and sums the minimum probabilities for terms that are common. The flattened version of the kPSS is simply a single list of terms and corresponding probabilities.

Other scoring functions can be used to produced different ranks and mappings. The *levenshtein* function is another example, this calculates the score as the word distance between terms. Another function implemented in the system is the *match* function (this helps simulating techniques based on grep[12]), that simply returns 1 if the words match, or 0 otherwise. Full details about Conclave-Mapper available in [6].

4 Related Work

Program Comprehension (PC) is a field of research concerned with devising ways to help programmers understand software systems. In this context, feature (or concept) location is the process of locating program elements that are relevant to a specific feature implementation. This is typically the first step a programmer needs to perform in order to devise a code change [3, 33].

Feature location techniques are usually organized by types of analysis: (a) dynamic analysis, which is based in software execution traces, and examines programs runtime (e.g. [1, 38]); (b) static analysis, based on static source code information, such as slicing, control or data flow graphs (e.g. [9, 27, 37]); and (c) textual analysis, explore natural language text found in programs like comments or documentation. This last type can be based on Information Retrieval (IR) methods (e.g. [4, 26]), NLP (e.g. [18, 39]), or pattern matching (sometimes also referred as grep-*like*) based approaches (e.g. [12]). For more details about different trends and other approaches please refer to surveys [11] and [42].

The Conclave-Mapper underlying feature location technique uses a combination of static and textual analysis, and ontologies. Examples of other approaches that explore the same combination of analysis include: in [45], Zhao *et al* use a static representation of the source code named BRCG (branch-reserving call graph) to improve connections between features and computational units gathered using an IR technology; in [17], Hill *et al* present a technique that exploits the program structure and also program lexical information; in [34], Ratiu and Florian establish a formal framework that allows the classification of redundancies and improper naming of program elements, which is used as a based to represent mappings between the code and the real world concepts in ontologies; in [16], Hayashi *et al* proposed linking user specified sentences to source code, using a combination of textual and static analysis domain ontologies. Other applications of ontologies in software engineering in [15].

State-of-the-art feature location approaches involve combining techniques taking advantage of having data produced from different types of analysis (e.g. [23, 26]).

5 Experimental Validations

This section describes two evaluations done for the tools illustrated in Section 3.

[12] `http://www.gnu.org/software/grep/` (Last accessed: 29-01-2014)

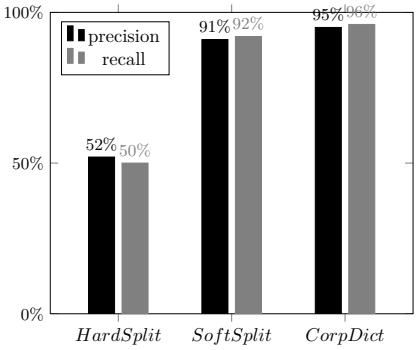


Figure 6 Precision and recall means for correct splits.

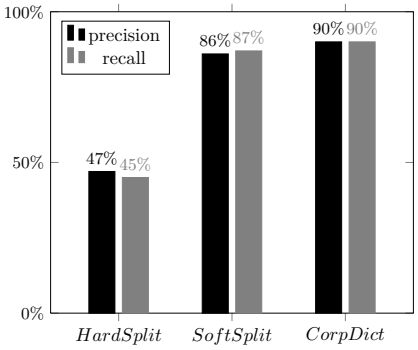


Figure 7 Precision and recall means for correct terms.

5.1 Splitting Evaluation

Section 3.1 describes the identifier splitting technique available in CONCLAVE. This section briefly describes the experimental study undertaken to evaluate the technique ability to correctly split and expand multi-term identifiers. The following research questions were defined:

RQ1: What is the percentage of identifiers in a program that LIDS can correctly split?

RQ2: What is the percentage of identifiers in a program that LIDS can correctly split and expand in case abbreviations were used?

To help answering these questions the following experience was performed:

Step 1: Create the *oracle*, i.e., for every ABCMIDI identifier manually create the correct split set, and correct term set. (this was done by the authors, and in cases where the was not an agreement, or the original programmer purpose was not clear, the identifier was not included).

Step 2: Compute the split and terms sets for every identifier in the *oracle* using LIDS hard-split function.

Step 3: Compute the split and terms sets for every identifier in the *oracle* using LIDS soft-split function, providing general purpose dictionaries.

Step 4: Compute the split and terms sets for every identifier in the *oracle* using LIDS soft-split function, providing general purpose dictionaries and the software specific dictionary.

Step 5: Compare sets computed in *Step 2-4* and the sets manually created in *Step 1* and measure precision and recall.

For a given identifier *id* to split let the *oracle* split set be: $o = \{o_1, o_2, ..., o_n\}$, and $s = \{s_1, s_2, ..., s_n\}$ the computed split, then the precision and recall are calculated as:

$$precision = \frac{|o \cap s|}{|s|} \qquad recall = \frac{|o \cap s|}{|o|}$$

where $|x|$ represents the cardinality of x. The same formulae are applied when calculating the measures for correct terms, but using the calculated sets of terms instead of splits.

Figures 6 and 7 illustrate the measurement results. For this software package the proposed technique was able to correctly split and expand almost all identifiers (precision and recall in the order of 90%). More details about this evaluation and comparisons with other techniques in [7].

Table 3 Better effective measure for different approaches for the jEdit benchmark.

Scoring Function	Analyzed Bugs	Better Eff. Measure
match	150	22
kPSS	150	51

5.2 Query Search Evaluation

Section 3.2 describe the underlying technique used in the Conclave system for feature location, based on kPSS. This section describes the preliminary evaluation done, to verify if this technique introduces benefits over other common techniques. In current available IDEs, common search facilities available to programmers, are still grep-like approaches, so the following research question was formulated:

RQ1: How does the *kpss* scoring function performs, when compared to the *match* scoring function, for finding relevant elements of the code given a search query?

To help answering this question the following experience was performed:

Step 1: in order to ease the process or replicating this experience the benchmark provided by Dit *et al*[13] for the jEdit[14] editor (version 4.3) was used, instead the devising a new data set. The benchmark contains a set of 150 bug reports, including the function set that was changed to resolve the bug (referred as the gold set) – more details about the benchmark in [11];

Step 2: the title for each bug report was extracted, stop words[15] were removed, and the resulting set was archived as keywords;

Step 3: for each bug report, the *locate* function to compute a rank was called, using the *match* scoring function, the keyword set computed in *Step 2*, and setting as range the `Function` program element;

Step 4: replicate *Step 3* but using the *kpss* scoring function;

Step 5: calculate the effectiveness measure for each resulting rank.

The effectiveness measure is calculated by analyzing the computed rank in order, and its value is the first position of the rank that is a relevant function. Functions that are part of the set of functions changed to resolve the bug (the gold set) are considered relevant. The rank position can be compared for different scoring functions to measure which rank produced the best results. This approach was also used in [31] and [36] for comparing feature location techniques performance.

The results of this experience are presented in Table 3. They show that for this software package the kPSS based scoring approach produced a better result 51 times, outperforming the 22 better results achieved by the simple *match* function. The remaining times either both approaches scored the same, or none of the relevant functions were found in the resulting rank.

Although these results are satisfactory, they do not provide enough empirical data to generalize the performance of kPSS based techniques. Also, the keywords used to build the queries and the functions gold sets are a threat to validity because: (a) the keywords set was built automatically from reports titles that sometimes lack relevant terms, or use only

[13] Available from: `http://www.cs.wm.edu/semeru/data/benchmarks/` (Last accessed: 29-01-2014).
[14] Available from: `http://www.jedit.org/` (Last accessed: 29-01-2014).
[15] Common words that tend to express poor semantics (e.g. *"the"*, *"a"*, *"too"*) [28].

ambiguous words (e.g. "*bug*"), a human would be more prone to devise a set of terms (after reading the report) that would create a more accurate rank; (b) sometimes, when fixing bugs, the actual defect is really not related to the concepts functions are addressing, which translates in changing code unrelated to search queries. Full details about this experiment and other case studies in [7].

6 Conclusion

Systems like CONCLAVE enables software engineers to devise mappings between the source code and problem domain concepts. These relations help the programmer to understand quicker the software, and discover which areas of the code need changing to address a specific feature or bug fix.

Many tools and techniques can be used to gather information about programs and the problem domain. The quicker the information is abstracted, the quicker other applications can use it. Using ontologies allows the combination of heterogenous results and data in a single representation format. Also, applications can take advantage of a panoply of tools available (e.g. inference engines, descriptive logics, OTK-*like* frameworks), to perform data analysis and relate elements in different domains. kPSS based feature location is a sound example of such applications. The OTK framework for abstracting ontology operations from the underlying technology has proven a valuable asset during applications implementation.

The main trends for future work include devising new functions to score relations between elements in the different ontologies, as well as combinations of approaches to produce more resources, and to convey more semantic information to ontologies with current available resources.

Acknowledgements. This work is funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

References

1 Giuliano Antoniol and Y.-G. Guéhéneuc. Feature identification: An epidemiological metaphor. *Software Engineering, IEEE Transactions on*, 32(9):627–641, 2006.

2 Sean Bechhofer, Frank Van Harmelen, et al. Owl web ontology language reference. *W3C recommendation*, 10:2006–01, 2004.

3 T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1994.

4 David Binkley and Dawn Lawrie. Information retrieval applications in software development. *Encyclopedia of Software Engineering*, 2010.

5 Nuno Ramos Carvalho, José João Almeida, Maria João Varanda Pereira, and Pedro Rangel Henriques. Probabilistic synset based concept location. In *SLATE'12 – Symposium on Languages, Applications and Technologies*, June 2012.

6 Nuno Ramos Carvalho, José João Almeida, Maria João Varanda Pereira, and Pedro Rangel Henriques. Conclave: Ontology-driven measurement of semantic relatedness between source code elements and problem domain concepts. In *14th International Conference on Computational Science and Its Applications (ICCSA)*, 2014. [forthcoming].

7 Nuno Ramos Carvalho, José João Almeida, Maria João Varanda Pereira, and Pedro Rangel Henriques. From source code identifiers to natural language terms. *Journal of Systems and Software*, 2014. [under review process].

8 Nuno Ramos Carvalho, Alberto Simões, and José João Almeida. Open source software documentation mining for quality assessment. In *Advances in Information Systems and Technologies*, volume 206 of *Advances in Intelligent Systems and Computing*, pages 785–794. Springer Berlin Heidelberg, 2013.

9 Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *8th International Workshop on Program Comprehension*. IEEE, 2000.

10 T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

11 Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 2013.

12 George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

13 Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. Tidier: an identifier splitting approach using speech recognition techniques. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

14 Latifa Guerrouj, Philippe Galinier, Y. Gueheneuc, Giuliano Antoniol, and Massimiliano Di Penta. Tris: A fast and accurate identifiers splitting and expansion algorithm. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*. IEEE, 2012.

15 Hans-Jörg Happel and Stefan Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pages 5–9. Citeseer, 2006.

16 Shinpei Hayashi, Takashi Yoshikawa, and Motoshi Saeki. Sentence-to-code traceability recovery with domain ontologies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 385–394. IEEE, 2010.

17 Emily Hill, Lori Pollock, and K. Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of 22nd IEEE/ACM international conference on Automated software engineering*, pages 14–23, 2007.

18 Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 2009.

19 I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: the making of a web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, 2003.

20 Wolfgang Keller. Mapping objects to tables. In *Proc. of European Conference on Pattern Languages of Programming and Computing, Kloster Irsee, Germany*. Citeseer, 1997.

21 Graham Klyne, Jeremy J. Carroll, and Brian McBride. Resource description framework (rdf): Concepts and abstract syntax. *W3C recommendation*, 10, 2004.

22 Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

23 D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *27th IEEE International Conference on Software Maintenance*, pages 113–122, 2011.

24 D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension*, 2006.

25 Vladimir Iosifovich Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

26 A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 214–223. IEEE, 2004.

27 Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on*, pages 33–42. IEEE, 2005.

28 James H Martin and D Jurafsky. Speech and language processing, 2000.

29 Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

30 Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. In *The Semantic Web-ISWC 2006*, pages 30–43. Springer, 2006.

31 Denys Poshyvanyk, Y.-G. Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 2007.

32 Eric Prud'Hommeaux, Andy Seaborne, et al. Sparql query language for rdf. *W3C recommendation*, 15, 2008.

33 V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002.

34 Daniel Ratiu and Florian Deissenboeck. How programs represent reality (and how they don't). In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 83–92. IEEE, 2006.

35 Daniel Ratiu and Florian Deissenboeck. From reality to programs and (not quite) back again. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 91–102. IEEE, 2007.

36 Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 14–23. IEEE, 2010.

37 Martin P. Robillard. Topology analysis of software dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(4):18, 2008.

38 H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *14th IEEE International Conference on Program Comprehension*, 2006.

39 David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.

40 Alberto Simões, José João Almeida, and Nuno Ramos Carvalho. Defining a probabilistic translation dictionaries algebra. In *XVI Portuguese Conference on Artificial Inteligence – EPIA*, pages 444–455, September 2013.

41 A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

42 Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, and LaTreva Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 2003.

43 Michael Würsch, Giacomo Ghezzi, Gerald Reif, and Harald C. Gall. Supporting developers with natural language queries. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010.

44 Yonggang Zhang. *An Ontology-based Program Comprehension Model*. PhD thesis, Concordia University, Montreal, Quebec, Canada, 2007.

45 Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, April 2006.

Leveraging Program Comprehension with Concern-oriented Source Code Projections

Jaroslav Porubän and Milan Nosáľ

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

Abstract

In this paper we briefly introduce our concern-oriented source code projections that enable looking at same source code in multiple different ways. The objective of this paper is to discuss projection creation process in detail and to explain benefits of using projections to aid program comprehension. We achieve this objective by showing a case study that illustrates using projections on examples. Presented case study was done using our prototypical tool that is implemented as a plugin for NetBeans IDE. We briefly introduce the tool and present an experiment that we have conducted with a group of students at our university. The results of the experiment indicate that projections have positive effect on program comprehension.

1998 ACM Subject Classification D.2.6 Programming Environments, D.2.11 Software Architectures

Keywords and phrases concern-oriented source code projections, program comprehension, projectional editing, code projections, programming environments

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.35

1 Introduction

Program comprehension is a process of retrieving information and knowledge about a software system by studying its source code. It is a process of recreating the mapping between the problem and the solution (implementation) domain that was created during the implementation phase of the software. Solutions in this area aim to reduce the amount of time needed for understanding the program source code. More radical solutions, like for example literate programming [12] or elucidative programming [14], were not adapted in the industry, probably because they were too distant from the industrial practice. In our previous work in this area presented in [11] we proposed a concept of *concern-oriented source code projections*. Concern-oriented source code projections overcome static structure of the source code by providing means to dynamically request a concrete view of the source code based on its concerns. In our method proposal specific concerns are associated with the source code by a metadata facility.

In this paper we continue in research presented in [11] and we discuss the process of creating projections. In addition we provide a case study that explains projections on a Minesweeper implementation in Java programming language. We have also implemented a tool prototype to experimentally evaluate the significance of concern-oriented source code projections for program comprehension.


© Jaroslav Porubän and Milan Nosáľ;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 35–50

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Concern-oriented Source Code Projections

Many times one source code is considered good by one programmer and bad by another one. In our work we recognize that the problem of multiple viewpoints to the source code quality is, to some degree, a consequence of *static structuring* of the source code. As *time* and programmer's *experience* changes, the programmer would choose different design, different code structure as best. The evaluation also depends on the *problem* that the programmer currently deals with. The problem of current approaches such as the object-oriented programming or aspect-oriented programming is that they allow the structure to meet some concrete needs, but adaptation of the structure to the new needs requires rebuilding the whole solution. Each new programmer has to work with the design that was previously chosen by someone else. They have to grasp a mental model of someone else even when the original design decisions are not relevant anymore. The comprehension of such a code is significantly impeded.

Our code projections are based on dynamic structuring of program's source code. By dynamic source code structuring we mean multiple different structures of the source code at the same time. In a specific situation the programmer would be able to choose the structure that he/she currently considers the most relevant. One structure is *base structure* (we can look at a base structure as a serialization structure of the system). Base structure is used to initially implement the system. Other concern-oriented source code structures are dependent on the base structure. A concern-oriented source code structure is a view of the source code that takes into account source code concerns (int this way it relates to an aspect from AOP). Concern-oriented source code projections can overlap; the same piece of code can belong to multiple projections.

These concern-oriented structures have to be properly presented to programmer; otherwise they would be useless for program comprehension. A code projection maps a set of base source code structures to a set of *view*s. The special *Identity* projection defines a view that is identical to base source code structure; therefore it has to fully describe the system. A single view consists of source code fragments that share a concern (or a set of concerns). We will call these fragments *view member*s. Relations between view members may be explicitly expressed in the view – e.g., a view can be graphical. A concrete code projection is specified by a sentence in a program query language (PQL). Practically any PQL can be used; however, it has to support querying some form of custom metadata. E.g., then a projection can query for code that implements logging, etc.

A programmer creates a *projection query* that specifies which concerns are relevant to his/her current situation. Projection queries can be shared and stored for later reuse, or modified if necessary. The concept of the code projections is outlined in Figure 1.

To provide code projections there has to be a tool that would be able to create a view while managing the source code in its base structure. In case of code projections we see as a best option utilization of the IDE thanks to its approach to handle language in its infrastructure (considering IDE is an integrated set of language tools). An IDE usually works with language on three levels – notation, model and view. The notation level is the language concrete syntax that is used to serialize a sentence of the language to the file system. The model is basically a form of abstract syntax tree that is obtained by parsing a sentence. The view is a presentation syntax of the language that is shown to the programmer. IDE components usually work upon the language model. The transition from the model to the view is done by the editor component of the IDE (there may be multiple different editors for one language). To provide code projections all that needs to be done is a substitution or

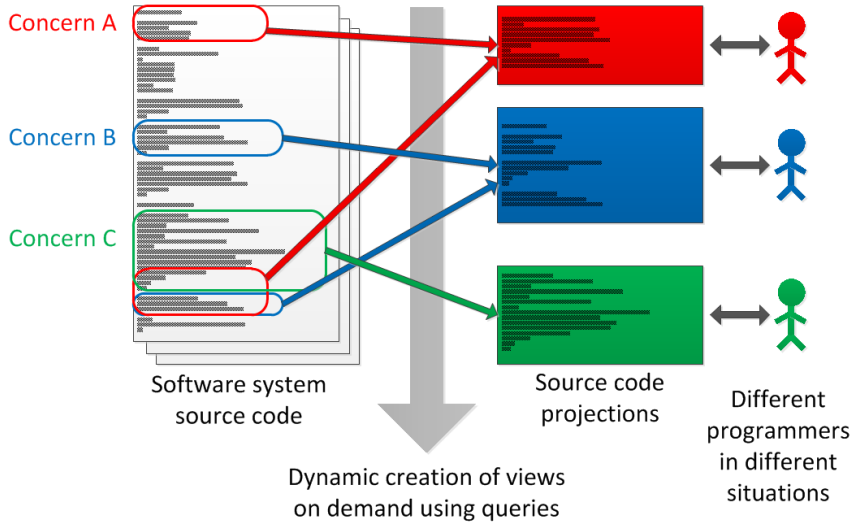


Figure 1 The concept of the concern-oriented source projections.

modification of an editor within the existing IDE. Since the editor works mainly with language model (abstract language representation), the concern-oriented source code projections are a specialized case of projectional editing discussed by Fowler in [3].

3 How to Create Projection Specification

As we have argued in previous section, our motivation for having different views of the software implementation is the variability of forces that affect the evaluation of the source code quality. In this section we will discuss the process of creating projections that provide programmers with those views.

Considering how a particular projection is created, we have following projection types:

- *annotation-based projections* that are based on explicit embedded metadata that are attached to source code just to create a projection,
- *configuration-based projections* that are based on explicit configuration or code conventions that are used to configure some framework or a tool, and
- *source intrinsic projections* that are based on the analysis of the rest of the source code.

A special case of projection is a projection composition that composes multiple different projections. Component projections may belong to different categories according to their creation. An example of a composite projection is a projection that shows all controllers (MVC pattern) that work with user profiles. First component of the projection is a projection showing all controllers and the second one a projection showing classes working with user profiles.

3.1 Need of Annotations

One of the biggest problems of program comprehension is the *semantic gap* between program representations on different abstraction levels. This covers also notoriously known semantic gap between model and implementation. However, the same can be observed on more fine-grained levels, such as in different source code representation. For example, let

us consider a software system implemented in Java. The implementation represented in Java source code will provide the reader with comments. Compilation, however, discards comments. Compilation will reduce explicit information about the program from its current representation. Moving from higher abstraction level to lower one decreases the amount of problem domain information.

The process of mapping the program representation from its current abstraction level to a higher one is the topic of program comprehension. Although there are attempts to aid comprehension with automatic program analysis (reverse engineering) these attempts are still not effective enough to significantly help the programmer. Usually the reverse engineering tools just provide a few source intrinsic projections of the program (e.g., visual projection showing the class diagram).

This is the reason why we believe that we have to record the mapping between the problem domain (model) and the solution domain (implementation) explicitly in every program representation. To record this knowledge in the source code any appropriate embedded metadata format can be used. Source code annotations, structured comments, code conventions or any other embedded format that will preserve the knowledge close to the source code (a brief comparison of metadata formats can be found in [9]). Explicit recording of the design decisions and high level semantic properties is a basis for annotation-based projections and moreover, it tightly couples the requirements and model to the implementation.

Annotation-based projections are the most expensive to create because they require the programmer to explicitly add declarative marks to the source code. These embedded metadata do not have semantics in a formal meaning, their presence in the source code does not change the behaviour of a program. However, they significantly narrow the semantic gap between the problem domain and the solution domain and this fact is a motivation for their creation. Even comments can be used to create annotation-based projections if they are structured. Using the method we proposed in [10] we can also use the concern-oriented annotations to generate documentation for the source code.

There are two scenarios for creating annotation-based projections. In the first scenario source code author records his *design decisions* and *program requirements* implemented by particular source code element seamlessly along with implementation. In the second scenario a new programmer is trying to comprehend existing source code and records his current understanding (in other words he is making notes).

3.2 Configuration-based Projections

Configuration-based projections use additional information in the source code that was not primarily intended for projections. For example, let us assume that a programmer uses Java Persistence API (JPA) compliant ORM mapping framework (such as Hibernate) for object persistence. To define mapping between the database and classes he/she has to provide an appropriate configuration. This could mean marking all the entity classes with the `@Entity` and `@Table` annotations and their fields with `@Column`, `@Id`, `@Basic`, etc., respectively. This configuration information is processable by a projection tool to provide views that are specific for a domain for which the framework was implemented (in case of JPA the data persistence domain).

This category of projections covers also code conventions. If we have classes to follow Java Beans convention we can easily identify all the methods that are used to access fields of classes. They all start with '`get`' prefix. In [9] we have included naming and type conventions as a metadata format. Here we argue that every convention can be considered a metadata

format and in consequence is a viable input for concern-oriented source code projections. Using conventions we can provide multiple projections for existing projects without requiring any additional effort from their authors.

Conventions have been and still are used as a configuration format in multiple frameworks and tools and their popularity grows with application of the *Convention over Configuration* (COC) design pattern. COC requires users of a framework to follow some conventional decisions (such as a particular naming convention, etc.) and for that they do not have to specify configuration for the framework that is usually complex. If the programmers do not want to spend a lot of time configuring the framework, they can choose to go with default settings and enjoy fast product delivery. Let us take a look on the Hibernate framework again. Since Hibernate uses COC design pattern all we have to do to have an entity class persisted is to mark it with the `@Entity` annotation. If we do not configure deviations from the default the framework conventionally maps class names to the identically named database tables and the fields to its columns, respectively. Therefore the configuration knowledge about the entity class is present in the source code (although less explicit). Since COC is usually applied with the *Principle of Least Astonishment*, the information "recorded" by conventions can be easily extracted from the source code. E.g., if we want to extract the names of persisted fields of an entity class, we can just take names of all the field of the class that are not marked with `@Exclude` annotation (`@Exclude` explicitly marks a field as not persisted).

This type of projections requires additional work from the programmers as well, however, this effort is motivated by the benefits gained from using external frameworks or tools[1]. However, the semantic gap between the problem and solution domains is wider. A configuration domain is usually an implementation domain (e.g., model-view-controller pattern) or a very specific problem domain (e.g., object-relational mapping domain).

3.3 Source Code Queries

The last type of projections are source intrinsic projections. In this case there is no need for any additional effort in form of conventions, annotations or comments from the programmer to enhance the source code. The bare implementation is projected using just queries in a PQL. An example of such a projections are projections implemented in current IDEs. Find usages projection shows all the usages of a particular program element (e.g., all method calls). Reverse engineering deals with this category of projections. E.g., a reverse engineering tool can build a class diagram of the implementation thus realizing a graphical projections of the system.

In this case we expect each line of code to directly effect the execution of the program. This is a direct benefit of writing the source code. However, the semantic gap is wider then in the other two projection types. A programmer transforms requirements in the problem domain to implementation in the solution domain (general purpose language). During this transformation the problem domain dissolves in the implementation.

3.4 Summary

These three projection types differ in effort and costs needed to prepare them. Figure 2 summarizes their nature. Source intrinsic projections can be done upon any implementation

[1] A programmer is not marking an entity class with `@Entity` to include it to a projection. He/she does it so the class would be processed by the JPA tool. The projection is just a by-product.

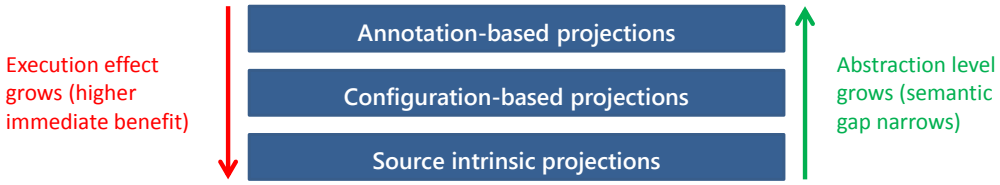


Figure 2 Different projection types.

but they lack proximity to the problem domain. Annotation-based projections require the code author to explicitly record his decisions and requirements that are implemented by the source code. This way annotation-based projections push the implementation closer to the problem domain. However, they require additional effort from the programmer. A compromise between the two are configuration-based projections that use conventions or configuration metadata. These metadata slightly narrow the semantic gap and their use is motivated enough even without projections.

4 Minesweeper Case Study

To illustrate concern-oriented source code projections on a meaningful example we will take a look at a Minesweeper game implemented in Java. The game is a simple reincarnation of the notoriously known Microsoft Minesweeper game that was distributed with the Windows OS family. Our Java Minesweeper has a text-based console user interface and has standard features such as monitoring the game state, opening tiles, marking tiles, etc.

In Figure 3 there is a simplified class diagram of the whole project. All the classes of the project represent the base structure of the implementation. However, Figure 3 is already a manually created projection – it hides the implementation of the methods and shows only the overall structure of the program.

4.1 Game State Semantic Concern

One of the core concepts of the game is the need to monitor the current state of the game and to behave according to that. The game starts with the state of generating the playing field. In this state the game is not playable. After the field is populated it gets the playing game state. In this state a player can play the game, he/she can open and mark tiles. Then we need at least two other states representing victory and game loss. These game states indicate that the game ended and that the player should not be able to interact with the game field anymore. A change of the state triggers some additional actions, such as recording a new high score or notifying the player that the game is over. In Figure 4 there is a state diagram of the game that represents a requirement for the implementation.

We had explicitly recorded the 'Game state management' concern of the source code in the implementation. To illustrate the 'Game state management'-oriented projection we have created a concrete view with our prototype implementation and it is shown in Figure 5. Its view members include the enumeration type for the game state and an attribute and methods of the `Field` class that are used to manage the current game state. Comments highlighted in green explain mapping the view back to the base structure of the program.

A skilled programmer might argue that in this case all we needed to do was to find all places where the `Field.state` is set. This is true. However, the knowledge that `Field.state` attribute is the right place where to start can be done only by code examination (unless we

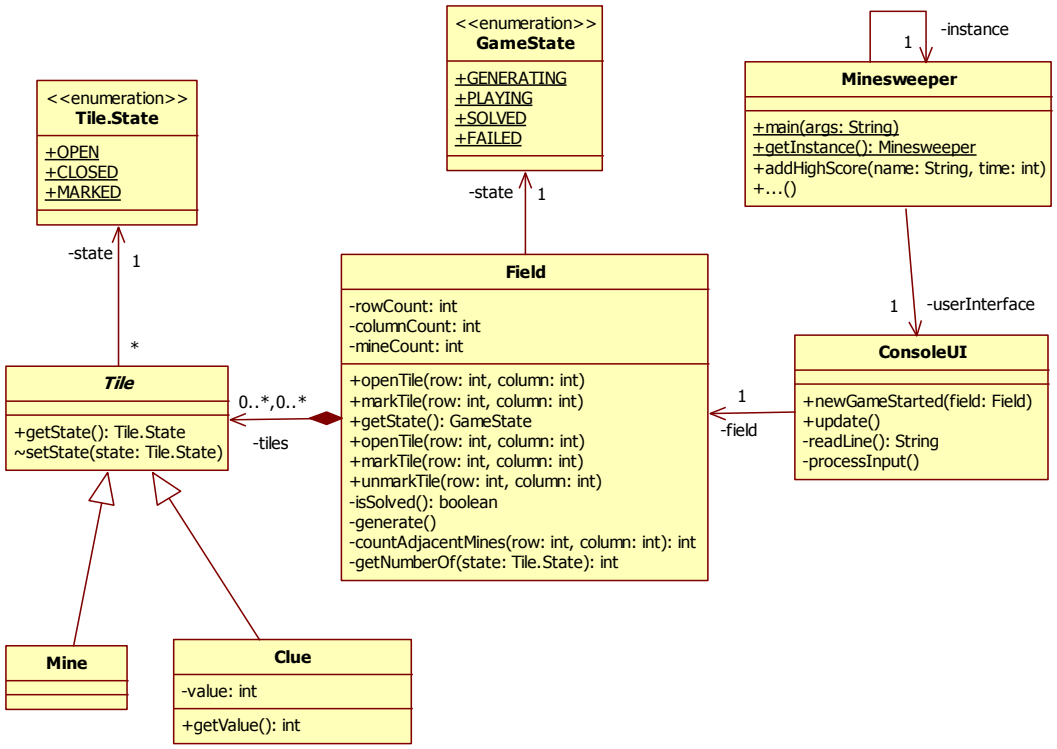


Figure 3 Simplified class diagram of the Minesweeper game.

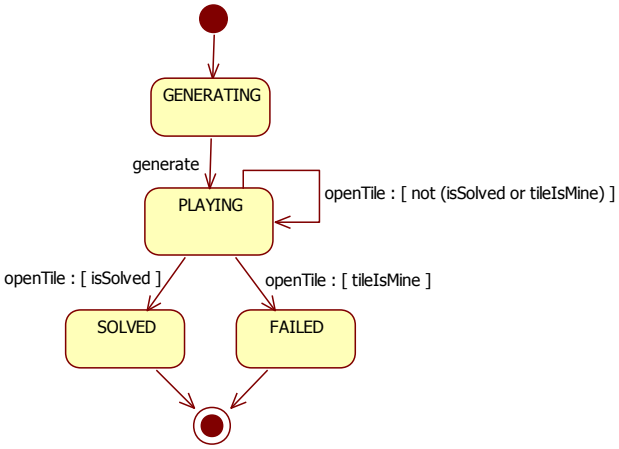


Figure 4 State diagram of the game state.

are authors of the code and we remember it). Here we used an explicit mark that annotates all program elements that manage the game state.

A source code projection based on this concern might be useful in multiple situations. It would simplify searching for a possible bug in a game state management, ease its comprehension for a new contributor to the project or simplify adding a new state to the game if that would be needed. Locating all the source code artefacts that would be affected by a state addition will be much faster with a projection based on this concern.

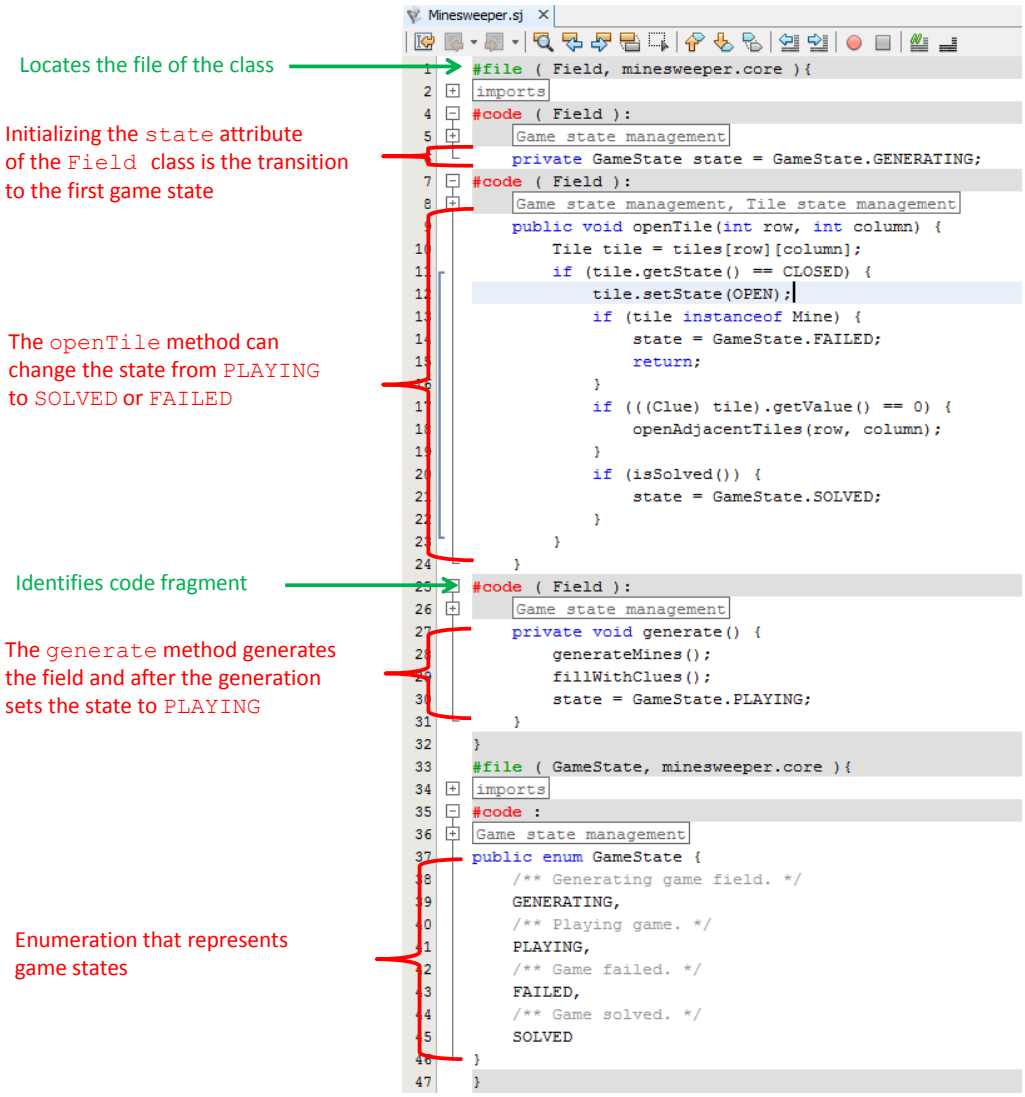


Figure 5 'Game state management'-oriented view of the Minesweeper game.

4.2 Singleton Design Concern

The same way we can use annotations to record also design decisions. As an example of a design decision for Minesweeper implementation we decided to implement `Minesweeper` class as a singleton. The `Minesweeper` class is a main class of the application. It manages game instances, it is responsible for the user interface choice (in future we want to support graphical UI), it will monitor game time and manage connection to database (to store high scores), etc. Although it did not have to be a singleton, it was our design decision and the class was implemented that way.

In Figure 6 there a 'Singleton design decision'-oriented view of the Minesweeper implementation. `Minesweeper` class is the only singleton in the implementation. For this projection we could also use just the naming convention of the `public static getInstance()` method. However, then if somebody would create a method with this signature that would not be a part of the singleton pattern the projection would be incorrect.

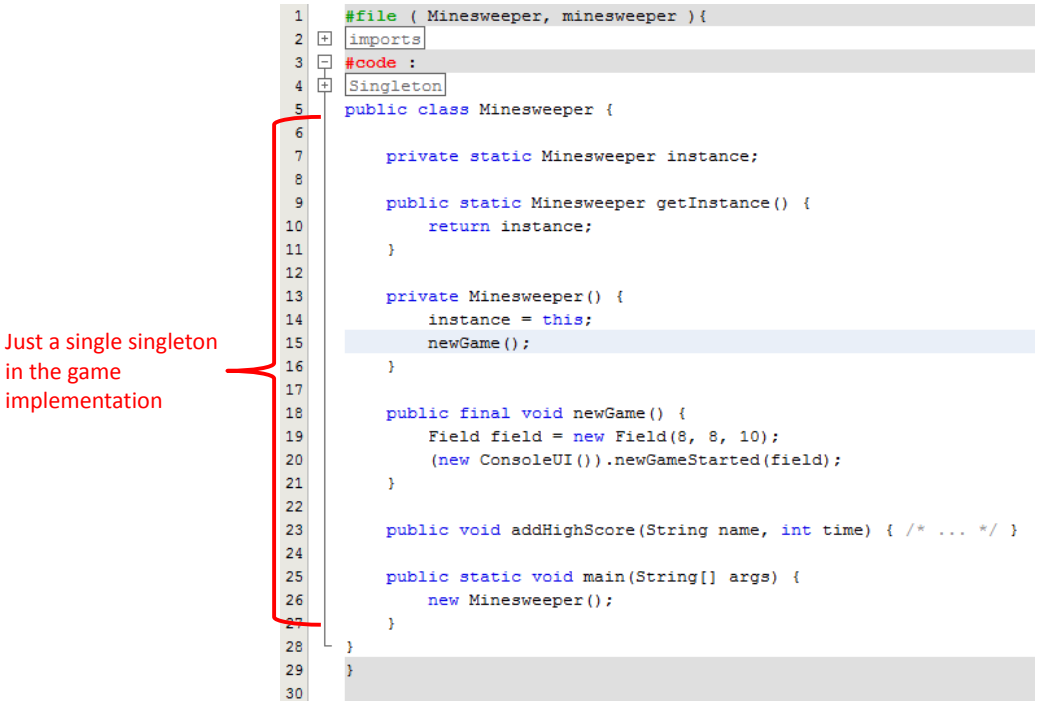


Figure 6 Singleton projection of the Minesweeper game.

This projection could be useful for software architect that wants to be sure that all the singletons in the implementation are indeed implemented as singleton, or in a similar scenario.

5 Prototype Implementation

In this section we present a prototype of concern-oriented source code projections tool. The tool was named *Sieve Source Code Editor* (SSCE) and it was used in an experiment with the students at our university. The SSCE tool was designed and implemented as a plugin for the NetBeans IDE[2]. This tool works with the code written in the Java programming language.

5.1 Expressing Concerns

In our prototype tool the concerns and the design decisions are preserved using an embedded form of software system metadata – structured comments. For our prototype tool we have use special comments that starts with the `SsceIntent` keyword. The follows enumeration of concerns separated by semicolon that crosscut the commented program element. For example we can take a look at a source code in listing 1 with the `addHighScore` method that besides other things is also responsible for persisting the current high score to database. These comments map the implementation details (JDBC connection, statement, etc.) to the problem domain (high score implementation, high score persistence).

[2] `https://netbeans.org/`

Listing 1 'Database connection' concern recorded using SSCE structured comment.

```
//SsceIntent:High score; High score persistence;
public void addHighScore(String name, int time) {
  ...
  Connection connection =
                DriverManager.getConnection(URL, USER, PASSWORD);
  Statement stm = connection.createStatement();
  ...
}
```

5.2 SSCE User Interface

In the time of the experiment we supported only marker structured comments that do not have parameters. A programmer can mark program elements with tags expressing their design or semantic properties – concerns. Then when another programmer is interested in a particular concern the SSCE can be used to select ("sieve") the relevant code according to these markers. To provide such a view SSCE defines an editor that is designed to present the result of the projection. An example of a view by a code projection using Sieve editor was already presented in Figure 5 where the user selected the Game state management source code concern. Another example was presented in Figure 6 where the selected concern was the Singleton design pattern application.

The result of the SSCE concern-oriented projection is a view that contains relevant code fragments (view members), which are presented in a single stand-alone document. View members in this document are wrapped in the contextual frames `#file()` and `#code()` to make clear the connection of the presented fragments to the base structure of the source code and to express the context of the code fragments. In Figure 5 these frames are annotated with comments highlighted in green. The `#file()` frame encapsulates code fragments from the same base file. The `#file()` frame is divided to the `#code()` frames that specify a connection of the code fragment to its parent in program elements tree (e.g., to which class a method belongs, etc.). These frames are used to provide context of the code fragments and to allow consistent editing of the source code in concern-oriented views. This way any code can be unambiguously mapped back to the base code structure. The mapping frames are generated and protected by the editor (protection is done using guarded sections feature of NetBeans IDE).

The Sieve editor utilizes code folding, guarded sections and syntax highlighting techniques for better orientation and readability of the result of the projection. E.g., by default all code fragments in the view are collapsed to provide better initial overview.

Concerning code projections, the SSCE tool provides two more user interface components for the user. One interface is called *Intents manager* and it can be used to edit concern tags of the selected program element. The Intents manager is shown in Figure 7. A programmer can mark program elements directly through the editor using the SSCE structured comments, or he/she can use this Intents manager to pick existing intents (concerns) and the SSCE will mark currently selected program element for him/her. This interface was designed to provide easier manipulation of the explicit concerns in the source code.

The last interface component is the *Intents filter* that serves for creating simple concern-oriented queries. The Intents filter is shown in Figure 8. Using the Intents filter the programmer can pick a concern or concerns from the list of all concerns present in the source code. The selected concern or concerns define a projection that will be used by the

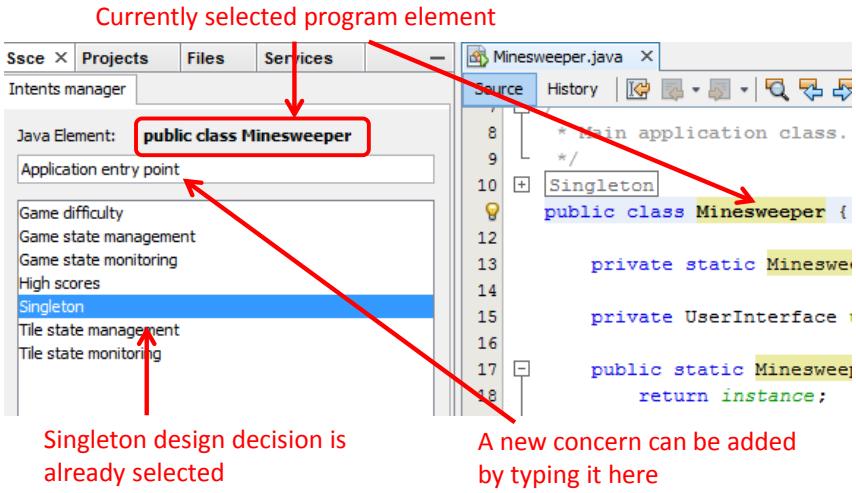


Figure 7 The SSCE Intents manager.

SSCE Sieve editor to create a view of the project's source code. The available concerns are presented in a simple multi-selection list. The set of concerns present in the source code is obtained by analysing the source code base.

So far to keep things simple we decided to use such a simple interface instead of a full-fledged PQL[3]. The Intents filter allow querying the code for code fragments that share some single concern, code fragments that all share a set of intents or code fragments that are a union of code fragments with at least one of the specified intent. This is done by selecting a single concern, selecting multiple concerns and choosing the `AND` composition mode or selecting multiple concerns and choosing the `OR` composition mode respectively. `AND` and `OR` composition modes allow simple composition of annotation-based projections.

6 Experiment

The idea of the concern-oriented source code projections was introduced to 40 respondents. These respondents were in the time of experiment bachelor's degree students in the 2^{nd} year of study. 28 of them were working with Java for less than a year. The rest stated that they were working with Java for a year or two. Only one of the respondents had experience with Java for more than two years. Thanks to their relatively short experiences with the language and programming we could assume that they would not be biased against the new technique. Since we use the NetBeans IDE on our practical lessons, all of them were familiar with it.

6.1 Experiment Setup

Our main intention in this experiment was to verify our hypothesis about code projections and their contribution to program comprehension. We therefore designed two tasks, in which the respondents had to fix a bug and change some implementation in an unfamiliar source

[3] Our experiment was performed on a group of our bachelor's degree students in the 2^{nd} year of study. Therefore we decided to use this simple mechanism instead of a full-fledged PQL, so they would not be overwhelmed by a complexity of a PQL and could rather focus on the contributions of the code projections method.

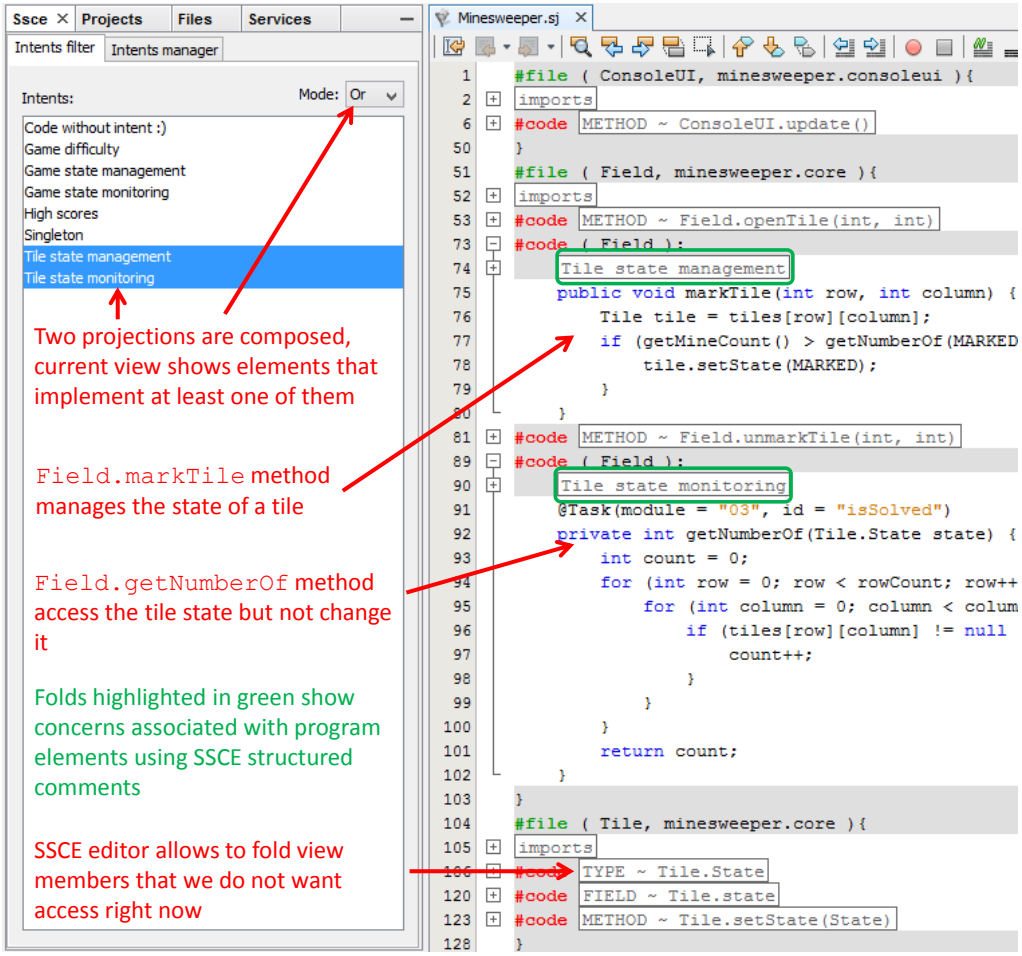


Figure 8 The SSCE Intents filter used to create simple composite projection.

code. As a source code sample we used the implementation of the SSCE plugin. The version of the plugin that was used in the experiment consisted of 33 classes with approximately 10 kLOC. Its character provided required complexity that could be compared to complexity of real world systems. The implementation was annotated with SSCE comments recording 28 different high level concerns (e.g., 'GUI update', 'Querying and concerns configuration', 'Annotations for recording concerns', 'Monitoring changes in Java classes', and more).

In their first task, they had to fix a bug with the `OR` composition mode of intents querying. In the implementation we have commented out few lines that took care of the `OR` mode. The main problem therefore did not have algorithmic character. It was rather a problem of program comprehension. Students had to browse the 10000 lines of code and search them for the implementation of composition modes. However, they could (and were encouraged to) use the plugin to prepare a view that would reduce the code needed to be searched. In this case it would mean selecting the 'Querying and concerns configuration' concern in the SSCE Intents filter. The SSCE editor would show them a single document with 830 LOC that consists of 1 whole class and 2 fields and 17 methods of 5 other classes. Instead of browsing all the classes the respondents could focus solely on the code that authors identified as relevant to querying the source code and concerns configuration. The view reduced the code to be searched more than 10 times.

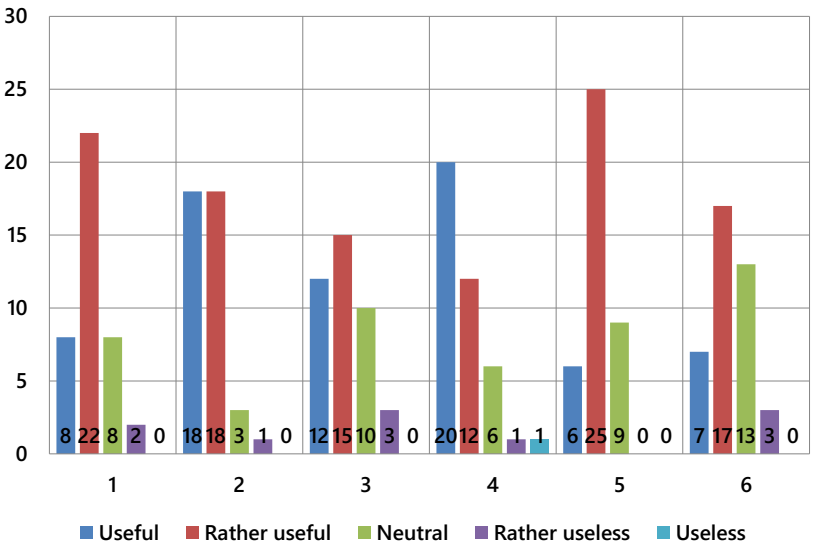


Figure 9 Usability rating of the code projections.

The second task was oriented more towards system evolution than to fixing bugs. The students had to change the structure of the SSCE structured comments. In this case it was again a matter of program comprehension, because they had to identify source code that implemented processing the SSCE structured comments. Here selecting the 'Annotations for recording concerns' concern for the view reduced the code to be searched to 500 LOC ($\approx$ 20x less than the whole code base).

6.2 Survey

After finishing these tasks the respondents were given a questionnaire about the method. They were asked to rate the significance of the code projections for the following activities:

1. Development of a new software system.
2. Maintenance of an existing system.
3. Program comprehension and understanding of the source code.
4. Orientation and navigation in the source code.
5. Testing and fixing bugs.
6. Documenting the source code.

All of these activities are related to program comprehension. The results are presented in Figure 9. In general, the code projections were rated positively, being considered useful or rather useful for all mentioned activities. Activities 2 and 4 were rated the best, indicating a positive impact on program comprehension. This experiment verified the importance of source code projections in the context of program comprehension. The results about activities 1 and 6 were more or less theoretical, because the respondents did not have to develop a whole new system and explicitly mark the source code with the SSCE structured comments. They could only experiment with tagging using the Intents manager. The problem of development and implementation of concern-enriched source code is a matter of our future research.

In summary the questionnaire results indicate that concern-oriented source code projections are beneficial for program comprehension. We believe that the relevance of the results

is even higher since the experiment was conducted with a very simple and feature-limited prototype of the tool.

7 Related Work

Our previous work presented in [11] provides motivation for source code projections and detailed description of our proposal. Section 2 provides brief introduction that covers the topic. Considering that with projections we aim to bring the formal source code representation closer to programmer's mental model, a related work to our is also the work of Kollár [5] who analyses the opposite direction in human-computer interaction. He discusses binding of informal semantics (human thinking) to semantic symbols (symbols processable by computer). His work is based on concept of language metalevels that can be used for slicing the abstraction gap between model and executable program (he uses the concept also in [6] for genetic evolution of programs).

Similar approach is used by Desmond et al. [2] in so called *Fluid source code views*. They allow viewing method bodies in place of their calls, thus reducing the need of browsing the source files. It is kind of similar to Go To Declaration projection of current IDEs, however using fluid source code views the body is shown directly in place of call using a tooltip. No explicit navigation across the source files is needed.

In modern Integrated Development Environments (IDE) there are already some built-in projections that use non-standard metadata, such as TODOs view that can show all lines of code marked with TODO comments. Or there is a projection that crosses out all the program elements marked with `@Deprecated` annotation, indicating that these program elements are deprecated and are not supposed to be used. In Jetbrains IntelliJ Idea there are multiple code projections that increase code comprehension. An example might be folding an anonymous class into lambda-like notation that is introduced by Java 8.

Intentional source code views [8] are sets of related program elements that share some intention. In this sense they are very similar to concern-oriented code projections. In Intentional views the intentions of the source code are specified using logic metaprogramming. Although they are close to our approach by providing means of defining architectural and conceptual information about source code, they differ in few rather important aspects. Intentional views require knowledge of logic metaprogramming. It is hard to expect every programmer to be a logic programmer. The Intentional View Browser is a new tool. In our code projections we want to utilize common programmer's natural environment – code projections are to be made integral part of a modern IDE. Intentional views use code conventions that tend to be fragile (see [4]).

Source code views are also proposed by Lommerse et al. [7]. However, their approach provides static view – merely according to standard source code properties (such as a Navigator view of the modern IDEs). They provide three views: the *syntactic view* showing syntactic constructs; the *symbol view* showing objects available after compilation; and the *evolution view* showing different version of source file. In this context it is worth to mention *Concern Graphs* [13], an abstraction used for better understandability of concerns in source code.

Callau et al. [1] introduce a concept of *ghost classes*. A ghost class is a class that is used but not yet defined. host classes exist so programmers can use them even if they have not defined them yet. Their usage raises no errors and IDE acts as if the classes existed. Callau et al. argue that ghost classes are useful for incremental programming where they represent flexible class prototypes.

Wada et al. [16] work in model driven development research. They use source code annotations to preserve links from the generated source code to the model used for generation. Their annotations therefore can be used to create our concern-oriented source code projections.

Besides Fowler another well known name in the field of projectional editing is Markus Voelter. His research is mostly centered around JetBrains MPS[4] language workbench. MPS utilizes projectional editing. Each language implemented in MPS defines also 'editors' for language concepts that define how are the concepts projected and edited in the MPS. One of Voelter's works is presented in [15] where he describes implementation of feature variability using MPS projectional language workbench. The MPS-like concept of projectional editing is focused on removing the parsing process from language editing. Instead of text-based editor that requires the editor to parse the input to provide any help MPS uses structural editor (structure-aware editor) that directly creates language AST. Their aim is to improve notation flexibility, error prevention, etc. In our work we understand projections rather as a mean to provide *multiple different views of the same system*.

8 Conclusion

In this paper we have discussed the basics of creation of our concern-oriented source code projections. The paper included a case study that shows usage of projections in a meaningful context and therefore explains motivation for using projections. We have also presented a prototypical tool that is an implementation of simple annotation-based source code projections. This tool was used in an experiment to get feedback on our concern-oriented source code projections from our students.

Contributions of this paper are as follows:

- A discussion of creation of concern-oriented projections with respect to costs.
- A case study explaining source code projections.
- Experimental evaluation by a survey confirming projections' positive effect on program comprehension.

Our plans for future work include improving the tool to work with Java annotations and to be able to support source intrinsic and configuration-based projections. From the methodical point of view we want to analyse how to prepare annotation-based projections so that they would cost as little as possible and still give programmers benefit. We perceive that the problem of motivation for explicit recording of concerns in the code is the main obstacle for using concern-oriented projections in practice.

Acknowledgements. Research presented in this paper is supported by VEGA Grant No. 1/0341/13 "Principles and Methods of Automated Abstraction of Computer Languages and Software Development Based on the Semantic Enrichment Caused by Communication".

References

1 Oscar Callau and Éric Tanter. Programming with ghosts. *IEEE Software*, 30(1):74–80, 2013.

2 Michael Desmond, Margaret-Anne Storey, and Chris Exton. Fluid source code views. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, ICPC'06, pages 260–263, Washington, DC, USA, 2006. IEEE Computer Society.

[4] `http://www.jetbrains.com/mps/`

3 Martin Fowler. Projectional editing. Blog entry, available at `http://martinfowler.com/bliki/ProjectionalEditing.html`, January 2008.

4 Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 195–213, Berlin, Heidelberg, 2005. Springer-Verlag.

5 Ján Kollár. Formal processing of informal meaning by abstract interpretation. In *KES IDT 2014 – 6th International Conference on Intelligent Decision Technologies*, KES IDT 2014, Chania, Greece, June 2014. Accepted.

6 Ján Kollár and Emília Pietriková. Genetic evolution of programs. In *Proceedings of the Twelfth International Conference Informatics 2013*, pages 127–132, November 2013.

7 Gerard Lommerse, Freek Nossin, Lucian Voinea, and Alexandru Telea. The visual code navigator: An interactive toolset for source code investigation. In *Proceedings of the 2005 IEEE Symposium on Information Visualization*, INFOVIS'05, pages 4–, Washington, DC, USA, 2005. IEEE Computer Society.

8 Kim Mens, Bernard Poll, and Sebastián González. Using intentional source-code views to aid software maintenance. In *Proceedings of the International Conference on Software Maintenance*, ICSM'03, pages 169–178, Washington, DC, USA, 2003. IEEE Computer Society.

9 Milan Nosáľ. Software system metadata alternatives to annotations. In *Proceedings of Poster 2013: 17th International Student Conference on Electrical Engineering*, pages 1–5, May 2013.

10 Milan Nosáľ and Jaroslav Porubän. Software documentation through source code annotations. In *Informatics 2013 : Proceedings of the Twelfth International Scientific Conference on Informatics*, Informatics'2013, pages 180–185, 2013.

11 Matej Nosáľ, Jaroslav Porubän, and Milan Nosáľ. Concern-oriented source code projections. In *Federated Conference on Computer Science and Information Systems (FedCSIS), 2013*, pages 1541–1544, Sept 2013.

12 James Dean Palmer and Eddie Hillenbrand. Reimagining literate programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA'09, pages 1007–1014, New York, NY, USA, 2009. ACM.

13 Martin P. Robillard and Gail C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE'02, pages 406–416, New York, NY, USA, 2002. ACM.

14 Thomas Vestdam. Elucidative programming in open integrated development environments for Java. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, PPPJ'03, pages 49–54, New York, NY, USA, 2003. Computer Science Press, Inc.

15 Markus Voelter. Implementing feature variability for models and code with projectional language workbenches. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD'10, pages 41–48, New York, NY, USA, 2010. ACM.

16 Hiroshi Wada, Junichi Suzuki, and Katsuya Oba. Modeling Turnpike: A model-driven framework for domain-specific software development. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA'05, pages 128–129. ACM, 2005.

Comment-based Concept Location over System Dependency Graphs

Nuno Pereira[1], Maria João Varanda Pereira[2], and Pedro Rangel Henriques[1]

1 **Centro de Ciência e Tecnologia da Computação (CCTC)**
Departamento de Informática, Universidade do Minho
Braga, Portugal
{nuno.filipe.gomes.pereira,pedrorangelhenriques}@gmail.com

2 **Centro de Ciência e Tecnologia da Computação (CCTC)**
Departamento de Informática e Comunicações,
Instituto Politécnico de Bragança
Bragança, Portugal
mjoao@ipb.pt

Abstract

Software maintenance is one of the most expensive phases of software development and understanding a program is one of the most important tasks of software maintenance. Before making the change to the program, software engineers need to find the location, or locations, where the changes will be made, they need to understand the program. Real applications are huge, sometimes old, were written by other person and it is difficult to find the location of the instructions related to a specific problem domain concept.

There are various techniques to find these locations minimizing the time spent, but this stage of software development continues to be one of the most expensive and longer. The concept location is a crucial task for program understanding.

This paper presents a project whose main objective is to explore and combine two Program Comprehension techniques: visualization of the system dependency graph and concept location over source code comments. The idea is to merge both features in order to perform concept location in system dependency graphs. More than locate a set of *hot* instructions (based on the associated comments) it will allow to detect the other instructions (the whole method).

1998 ACM Subject Classification D.2.7 Maintenance

Keywords and phrases program comprehension, concept location, comment analysis, system dependency graph

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.51

1 Introduction

It is known that the software maintenance task is the most expensive phase of software development: 80% to 90% of the overall costs [5]. According to [6] we conclude that half the time spent in software maintenance is used to understand the program code and the instructions that have to be changed. We are aware that this task would be easier when a model driven software development is used [10] but this is not the most usual case.

These conclusions are easily understandable because before making the change to the program, software engineers need to find the location, or locations, where the changes will be made. These programs tend to be huge, in terms of lines of code and number of files, are usually written by different software engineers with different visions of the problem and


© Nuno Pereira, Maria João Varanda Pereira, and Pedro Rangel Henriques;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 51–58

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

different forms of thinking. Moreover the variable and method names in the source code may not be explicit and usually the program does not have good documentation.

There are various techniques to find these locations minimizing the time spent searching but the most used techniques consists in navigating through the statements dependencies or search for keywords that can indicate where the concept is implemented.

According to Kernighan and Plauger [7], the best documentation for a program includes comments; software engineers make a lot (and relevant) comments in the source code [12]. About 19% of the source code are comments. Comments can explain source code in a natural language connecting the program domain with the problem domain [1]. Aware of that, software engineers search for certain keywords in the code (related to the change that will be made) that can indicate the location, or locations, where the software engineer needs to modify source code.

Static dependency search consists in navigating through the dependencies among elements. Software engineers usually begin at the main function and follow a specific path in order to find the desired concept implementation. If the search is not successful, they must backtrack to the previous point (e.g., class, method or conditional structure) and choose a different path.

For all these reasons it becomes clear that it is challenging and useful to create a tool with a friendly interface that allows to perform concept location over a System Dependency Graph (SDG). The main idea is to visualize a dependency graph (control flow and data dependencies) and locate the nodes that can be related to a given term.

The term to search usually belongs to the problem domain and the related nodes are identified through the source code comments associated to each node/statement. To detect the comments that are related to a given term/concept a tool called Darius is used. Darius, which was developed in the context of our research group by Freitas [3], is based on natural language techniques applied to the comment words; it also produces a set of statistics concerning those comments. The System Dependency Graph is automatically generated from the software package.

The final objective of the tool, DariusSDG, proposed in this paper, is to decrease the time spent to locate concepts in source code in order to reduce the cost of maintenance tasks.

The paper will have four more sections. In Section 2 are discussed the area of Program Comprehension: some concepts, definitions and techniques which are fundamental to this work. Section 3 describes the tool architecture. The methodology and all the work related to the construction of the tool will be described in Section 4, which is divided in three subsections:

- Comments Analysis: where Darius will be introduced.
- System Dependency Graph (SDG): this subsection will describe in more detail this technique along with a tool that help building the graph.
- Integration: this subsection will describe the proposed DariusSDG that integrates the tool that builds the SDG with the tool that analyses source code comments (Darius) as well as other new features.

This document ends in Section 5 where conclusions and future work are described.

2 Contextualization: Program Comprehension

Program Comprehension [13] is a component of Software Engineering discipline whose principal purpose is to study how software engineers understand programs.

Every software engineer has its own way to understand a program, to capture information from the source code [14]. To help him on this task there are several tools that can be used to explore the source code. The choice depends on the needs of the user: static or dynamic source code analysis, use of visualizations or textual information to show the results and so on. Moreover, almost tools are programming language dependent and some of them adopt invasive approaches that modify the source code with code inspection instructions [2].

Understanding a program depends on the knowledge the software engineers has about the program that is being analysed, on his experience and on his knowledge about the real world problem that the program solves. These two concepts, *real world problem* and *how it is solved in a programming language*, are known in Program Comprehension as: Problem Domain and Program Domain.

We can see Problem Domain as the concepts related to the problem, the relations between them and how the problem can be solved. For instance, if a teacher needs to manage a school class there are various concepts related to that problem like *students*, *grades*, *faults*, *summaries* and so on. In a similar way there are various tasks to be accomplished like adding a summary, register the grades of the students, register the information about the students.

Program Domain is concerned with the programming language and with the implementation techniques used to solve the problem in a computer. Taking the example above, we can say that in Program Domain the concern is what data structures will be used to store the information (an array?), how the information about the grades will be implemented (as attribute of the school class or the student?) among others.

The time spent understanding a program also depends on the program that is being analysed: how it was created, how it is being maintained, in which programming language is written. The changes that should be implemented also have a strong influence in the effort and time required for program comprehension. When a software engineer analyses a program he constructs a Mental Model [14] of the program, which is updated when new information are collected.

The software engineer needs to know the flow of the program, which methods are called, by who, what these methods do, the data dependencies and the effect a change may cause. As said before, programs tend to be huge (many lines of code, many methods and many files) and it causes to be unworkable the task of knowing all these details about the program by hand. Every time a change need to be made in the program the software engineer need to navigate in the methods and discover the location where the change will take place. This is an enormous waste of time. The System Dependency Graph [8] is a visual artefact, used by program comprehension researchers, that shows all the static dependencies of a program in the form of a graph. Software engineers can see the flow of the program and data dependencies in a very easy and intuitive way.

From the text (comments, variable names, method names, constant Strings) in the program software engineers can obtain various kinds of semantic information about a program. If a method is called "return_average_grade" it is almost certain what the method returns. There are many techniques of Information Retrieval that can be used to retract important information about a program. The software engineer can search for certain keywords and the system retrieves this information (associated whit other important information like the file, the method or the line where the information was located).

It is interesting and makes sense to join these two techniques in one tool: showing semantic and structural information about a program. This means the Information Retrieval techniques can be associated to the System Dependency Graph showing concepts that appear in a certain comment associated with a certain method.

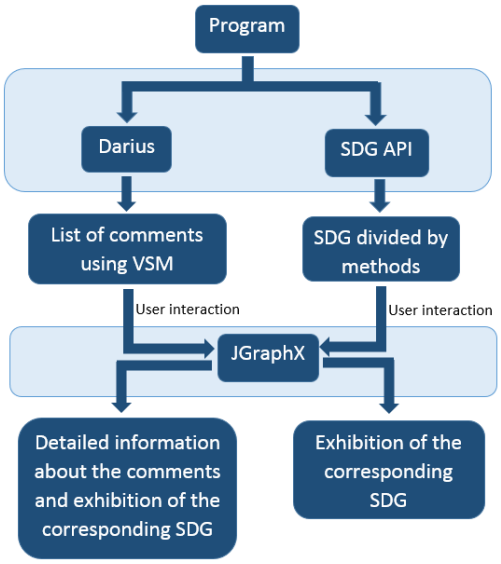


Figure 1 Architecture of DariusSDG.

3 DariusSDG: Architecture

As mentioned before the objective of DariusSDG, the tool proposed in this paper, is join two technique of program comprehension reducing the time spent to understand a program. To achieve this goal DariusSDG will be composed of three main components (that will be described in more detail in the next section):

- Darius: A comment analyser tool
- SDG API: A tool to extract flow and data dependency graphs from a program
- JGraphX: A tool that can be used to draw graphs

In Figure 1 we can see the diagram that depicts the tool architecture, its structure, components and connections among them. From this diagram it is possible to understand the steps that are taken by the tool to build the result, which is the construction of the System Dependency Graph and the mapping of the extracted information from source code comments with the nodes of the graph. The tool receives a program as input and uses the SDG API to build the System Dependency Graph of the program. As mentioned in Figure 1 the SDG is divided in methods allowing to have big programs as input. The input program will be also analysed by Darius, constructing a list of comments based on Vector Space Model (VSM) technique. The final result is a conjugation of the outputs of Darius and SDG API in a form of a graph, using JGraphX.

4 DariusSDG: Development

This section is divided in three subsections where will be described the tools and the steps used to build DariusSDG.

4.1 Comments Analysis

Comments in the code are one of the most important source of information about the program. It is one of the best ways to understand what the software engineer was thinking and how

the Problem Domain and Program Domain were related. Studies [17, 15] conclude that programs that have more comments are more easily understandable by software engineers.

To extract and search information in the comments we need to use Information Retrieval techniques. As we can see in [11], an Information Retrieval System analyses several documents processing its text with the assistance of some tools like:

Sentence tokenizer Separates the text into sentences.
Word tokenizer Separates the sentences into words.
Stemming Reduce the word to its grammatical root.
Elimination Eliminates words that do not have significance or value.

After the system complete all the tasks the user only needs to insert the query (set of keywords) to execute. The system perform the same process mentioned before in the query, to assure consistence, and retrieves a set of documents that satisfy the search ranking them by relevance.

There are various algorithms designed to rank documents, however in our work we will be concerned mainly with Vector Space Model [18] (that is used by Darius).

Darius, built by José Luís Freitas, in his master work [3] at our research group, uses several techniques of Information Retrieval to analyse the various types (inline, singleline, multiline and Javadoc) of comments presented in the source code.

The lack of available tools that can perform these actions, the quality of the tool and the fact that José Luís provided its source code were the reasons to choose Darius.

Darius is composed by four main modules: a comment extractor; a statistic calculator; a word analyser and a concept locator. As mentioned before, Vector Space Model is one of the most used algorithms to rank documents and it is used by Darius. There are many algorithms to calculate the weight of a word like the frequency of the word in the document but the one adopted is the Term Frequency – Inverse Document Frequency.

4.2 System Dependency Graph

The System Dependency Graph [8] is a visual artefact representing the static dependencies of a program as a graph. The System Dependency Graph (SDG) is composed of two components: the Control Flow Graph (CFG) and the Data Dependency Graph [4] (DDG).

CFG shows all the dependencies and calls between methods. It shows all the statements of the program and how they are related. If a statement B is called by a statement A, statement B is connected to statement A and it is positioned one level above statement A. Just looking to the CFG, a software engineer can see the flow of the program and discover in which class, method and statement, he will do the change (when a maintenance is needed).

DDG shows where the variables in the code are changed (variable dependencies). As in CFG, when a variable is changed in one statement it is connected to that statement showing that there is a dependency between them.

As mentioned above, when joined, these two graphs form the System Dependency Graph. A graph where the software engineers can see all the static dependencies and workflow of the program in a very easy and intuitive way.

In Figure 2 we can see the System Dependency Graph of the source code in Listing 1 (example extracted from [16]).

Creating by hand a SDG for a given program is not an easy task. And despite the fact that Java is one of the most famous programing language, there is a very small amount of tools to analyse Java code and build its dependencies (packages, classes, variables, methods), and an even more small number to build the SDG.

Listing 1 Excerpt of a Java program.

```
public static void main(String[] args) {
    int sum = 0, i = 0;
    while (i < 11) {
        sum = add(sum, i);
    }
    System.out.println("sum = " + sum);
    System.out.println("i = " + i);
}
```

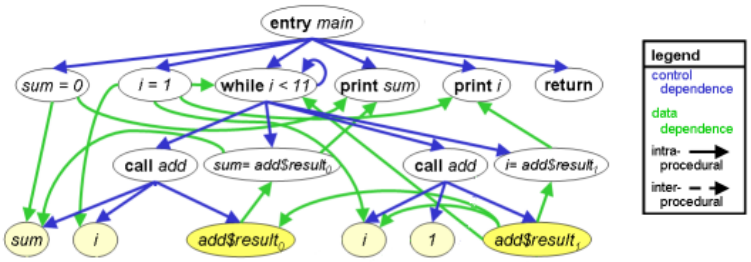


Figure 2 A System Dependency Graph.

The Java System Dependence Graph API [16], as the name indicates, is a tool that constructs the SDG of a program and provides methods to access it. The tool provides methods to navigate throw the nodes using algorithms like the Breadth First Search. We choose this tool because it is simple to use (does not need configurations or databases), it is open source and after some tests we have confirmed that the tool does what it compromises.

4.3 Integration

As mentioned before, DariusSDG is a combination between tools (Darius, JGraphX and SDG API). These tools were built independently and not with the purpose of being joined, therefore there is some adjustments to be made.

As said above, SDG is an important and useful tool due to its graphical representation. The System Dependence Graph API only offers methods to access the different nodes and connections between them (edges) in a text representation. On account of that, it is necessary a tool to build the graphical representation of the SDG. JGraphX [9] is a Java Swing library that provides functionality for visualisation and interaction with graphs. It offers many methods to construct the graph, change the colour of the nodes or the edges and can order the graph in a hierarchical form (which is the typical form of a System Dependency Graph). By combining these two tools it is possible to build a System Dependency Graph of any Java program.

As the SDG of a program can be very huge, hindering the work of the software engineer instead of helping, DariusSDG provides the System Dependency Graph divide by methods. As we can see in Figure 3 DariusSDG provides a list (on the right) of all methods used by the program and the name of the class they belong. By clicking on the method the corresponding part of the System Dependency Graph is shown on the centre of the window.

As mentioned before, DariusSDG show the SDG divided by methods, decreasing the size

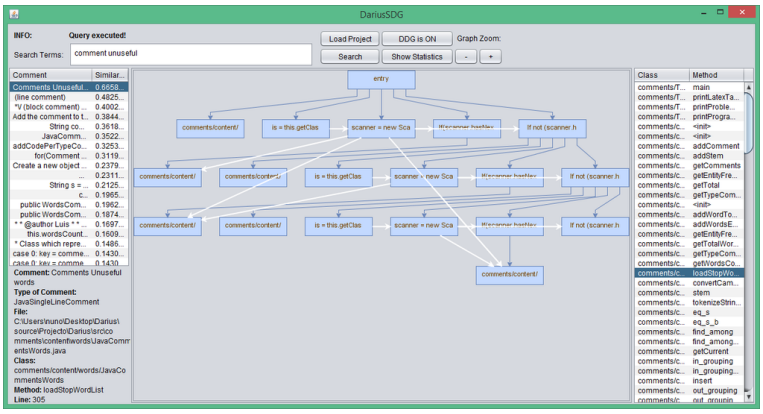


Figure 3 DariusSDG GUI.

of the graph shown, but, if the method has many instructions and dependencies it can be a little confuse. DariusSDG has some options to help the software engineer, for example the Data Dependency edges can be hidden; it is possible zoom in or zoom out in the graph; or drag the nodes changing its original position.

Darius retrieves a list of comments ordered by similarity according to the list of searched terms. Each comment has associated the comment itself, the file where it is located, the type of comment (inline, singleline, multiline and Javadoc) and the line after the comment. We decided to follow the same logic and return the method associated with the comment (if exists) and the line where the comment start.

When the software engineer searches for terms a list of comments is presented and its similarity with the searched terms on the left side of the window (see Figure 3). These comments are ordered by similarity and if the user clicks in one of them DariusSDG shows the information associated with the comment and the part of the System Dependency Graph associated with the respective method. The list in the right automatically changes the selected row to the row that corresponds with the graph that is being shown.

5 Conclusion

As mentioned along this paper, software maintenance is one of the most expensive parts of software development, and the time spent by software engineers to understand the program (an compulsory but unproductive phase) is the main reason for that.

Program Comprehension researchers studied and develop many techniques and tools to decrease the time spent to understand a program, but software maintenance still a very demanding task. The number of tools found, for the Java programming language, which can assist software engineers in program comprehension, is small and focused in one technique of program comprehension.

DariusSDG try to combine two techniques of Program Comprehension that can show semantic and structural information about a program. DariusSDG also was built to be easy to use and understand, avoiding even more wasted time, perform concept location over System Dependency Graph.

In the future the tool will infer the exact instruction associated with the comment. With this we can emphasize the node of the System Dependency Graph associated with the comment and the instruction associated. As future work we will also perform tests, with real

programs and software engineers, to verify if the time spent using the tool is smaller than without the tool.

Acknowledgements. This work is funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

References

1 Ruven E. Brooks. Using a behavioral theory of program comprehension in software engineering. In Maurice V. Wilkes, Laszlo A. Belady, Y. H. Su, Harry Hayman, and Philip H. Enslow Jr., editors, *ICSE*, pages 196–201. IEEE Computer Society, 1978.

2 Daniela da Cruz, Mario Béron, Pedro Rangel Henriques, and Maria João Varanda Pereira. Code inspection approaches for program visualization. *Acta Electrotechnica et Informatica*, 9(3):32–42, Jul-Sep 2009. ISSN: 1335-8243.

3 José Luís Figueiredo de Freitas. Comment analysis for program comprehension. Master's thesis, University of Minho, 2011.

4 Lin Du, Guorong Xiao, and Daming Li. A novel approach to construct object-oriented system dependence graph and algorithm design. *JSW*, 7(1):133–140, 2012.

5 L. Erlikh. Leveraging legacy system dollars for e-business. In *IT Professional*, volume 2. IEEE Computer Society, 2000.

6 R. K. Fjeldstad and W. T. Hamlen. Application program maintenance study: Report to our respondents. In *Proceedings of GUIDE 48*, April 1983.

7 Brian W. Kernighan and P. J. Plauger. *The elements of programming style*. McGraw-Hill, second edition edition, 1978.

8 Panos E. Livadas and Theodore Johnson. An optimal algorithm for the construction of the system dependence graph. *Inf. Sci.*, 125(1–4):99–131, 2000.

9 JGraph Ltd. JGraphX. `https://github.com/jgraph/jgraphx`, 2014.

10 I. Luković, S. Ristić, S. Aleksic, and A. Popović. An application of the MDSE principles. In *III Workshop on Model Driven Software Engineering (MDSE 2008)*, 2008.

11 Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2009.

12 I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris. Code quality analysis in open source software development. *Information Systems*, 12(1):43–60, 2002.

13 Margaret-Anne D. Storey. Theories, methods and tools in program comprehension: Past, present and future. In *IWPC*, pages 181–191. IEEE Computer Society, 2005.

14 Margaret-Anne D. Storey, F. David Fracchia, and Hausi A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

15 T. Teny. Procedures and comments vs. the Banker's algorithm. In *SIGCSE Bull*, pages 44–53, 1985.

16 Eric Chi Lik Tong, Chun Yinand Lo and Ming Hay Luk. Java system dependence graph API. `http://www4.comp.polyu.edu.hk/~cscllo/teaching/SDGAPI/`, 2010.

17 Scott N. Woodfield, Hubert E. Dunsmore, and Vincent Yun Shen. The effect of modularization and comments on program comprehension. In Seymour Jeffrey and Leon G. Stucki, editors, *ICSE*, pages 215–223. IEEE Computer Society, 1981.

18 Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. SNIAFL: towards a static noninteractive approach to feature location. *ACM Trans. Softw. Eng. Methodol.*, 15(2):195–226, 2006.

Part III

Domain Specific Languages

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ReCooPLa: a DSL for Coordination-based Reconfiguration of Software Architectures

Flávio Rodrigues, Nuno Oliveira, and Luís S. Barbosa

HASLab – INESC TEC & Universidade do Minho
Braga, Portugal
pg22826@alunos.uminho.pt, {nunooliveira,lsb}@di.uminho.pt

Abstract

In production environments where change is the rule rather than the exception, adaptation of software plays an important role. Such adaptations presuppose dynamic reconfiguration of the system architecture, however, it is in the static setting (design-phase) that such reconfigurations must be designed and analysed, to preclude erroneous evolutions. Modern software systems, which are built from the coordinated composition of loosely-coupled software components, are naturally adaptable; and coordination specification is, usually, the main reference point to inserting changes in these systems.

In this paper, a domain-specific language—referred to as ReCooPLa—is proposed to design reconfigurations that change the coordination structures, so that they are analysed before being applied in run time. Moreover, a reconfiguration engine is introduced, that takes conveniently translated ReCooPLa specifications and applies them to coordination structures.

1998 ACM Subject Classification D.2.11 Software Architectures, F.3.2 Semantics of Programming Languages

Keywords and phrases domain-specific languages, architectural reconfiguration, coordination

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.61

1 Introduction

For the last few years, Service-Oriented Architecture (SOA) has been adopted as the architectural style to support the needs of modern intensive software systems [9]. SOA systems are based on services, which are distributed, loosely-coupled entities that offer a specific computational functionality via published interfaces. Within SOA, services are coordinated, so that the ensemble delivers the system required functionality. Coordination is the design-time definition of a system behaviour. It establishes interactions between software building blocks (services, in SOA systems), including their communication constraints and policies. Such policies may be encapsulated in a multitude of ways [3], but point-to-point communication approaches (*e.g.*, channels [4]), gain relevance by fomenting the desired decoupling between computation and coordination concerns. This separation of concerns makes SOAs flexible, easier to analyse and naturally dynamic. Although policies are pre-established, services with similar interface may be discovered and bound to the architecture at run time, rather than fixed at design time.

Flexibility and dynamism are desired features in production environments where change is the rule rather than the exception. Constant environment evolution brings new requirements to the system, may contribute to degradation of contracted Quality of Service (QoS) values, or introduce failure [24, 28]. These changes raise the need for systems to adapt to new contexts while running.


© Flávio Rodrigues, Nuno Oliveira, and Luís S. Barbosa;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 61–76

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Reconfigurations upon SOA systems usually target the manipulation of services: dynamic update of service functionality, substitution of services with compatible interfaces (but not necessarily the same behaviour) or removal of services [26, 23, 11]. However, in some situations, this may not be enough. For instance, when a substituting service has incompatible interface, it may be necessary to target, with further detail, the way services interact with each other. This sort of reconfiguration goes into the coordination layer and usually substitute, add or remove interaction components (*e.g.*, *communication channels*), move communication interfaces between components and may even rearrange a complex interaction structure [13, 14]. Thus, there is a mismatch between project needs and what is currently offered in practice. More worryingly it is the lack of rigorous (formal) methods to correctly design and analyse this sort of reconfigurations.

In the authors' previous work [19, 20], a formal framework for modelling and analysing coordination-based reconfigurations in the context of SOA was defined. In this framework, a coordination structure (referred to as a *coordination pattern*) is regarded as a graph whose nodes reprsent interaction points (with either services or other coordination patterns), and edges are communication channels with a specific behaviour. However, this framework lacks mechanisms to express and apply reconfigurations, in practice. Such is the purpose of this paper: to introduce a Domain-specific Language (DSL), referred to as ReCooPLa, to express coordination-based reconfigurations, materialising the formal model presented in [19] and briefly discussed in further sections.

DSLs [27, 18, 21] are languages focused on particular application domains and building on specific domain knowledge. Their level of abstraction is tailored to the specific domain, allowing for embedding high-level doamin concepts in the language constructs, and hiding low-level details under their processors. In addition, they allow for validation and optimisation at the domain level, offering considerable gains in expressiveness and ease of use, compared with General-purpose Programming Languages (GPLs) [12].

In this spirit, ReCooPLa is a simple and small language that provides a precise, high-level interface for reconfiguration designers. The reconfiguration construct plays, then, a main role in ReCooPLa. It resembles functions, as in GPLs, with a header and a body. The header defines the reconfiguration identifier and its arguments; the body is composed of instructions, where coordination-specific notions are embodied in constructs that manipulate the graph structure which underlies coordination patterns.

A suitable reconfiguration engine, for application of the reconfigurations expressed in ReCooPLa is also proposed in this paper. It is regarded as a *machine* that executes reconfigurations over the target coordination patterns. To this end, a translation of ReCooPLa constructs into the engine's running code is carefully defined.

Outline. Related work is presented in Section 2 and background notions are introduced in Section 3. In Section 4 the ReCooPLa language is introduced with a detailed way and illustrated by small examples. Then, Section 5 introduces the reconfiguration engine along with a suitable translation of ReCooPLa constructs into it. Section 6 discusses an example. Finally, Section 7 concludes and proposes some topics for future work.

2 Related Work

Domain-specific languages constitute an important tool to tackle the specificities of particular application domains. The design of reconfigurations in the context of SOA is the domain underlying this work. Typical design approaches to reconfigurability in software architecture and component based design are discussed in this section.

Fractal [6] is a hierarchical and reflective component model intended to implement, deploy, and manage complex software systems, which embodies mechanisms for component composition and dynamic reconfiguration. It counts on FPath and FScript [8], which are DSLs, to securely apply changes. The former eases the navigation inside a Fractal architecture trhough queries. The latter, which embeds FPath, enables the definition of adaptation scripts to modify the architecture of a Fractal application, with transactional support.

Reference [7] proposes Architectural Design Rewriting (ADR) as a declarative rule-based approach for modeling reconfigurable Software Architectures (SAs). It is based on an algebraic presentation of graph structures and conditional rewrite rules, suitable to model. hierarchical designs, and inductively defined reconfigurations.

In general, Architecture Description Languages (ADLs) provide a rigorous foundation for describing SAs, specifying syntax and semantics to describe components, connectors, and their configurations. Numerous ADLs have been developed, each providing complementary capabilities for architectural development and analysis. Their use has been limited to static analysis and generation focused on static issues and, therefore, unable to support architectural changes. However, a few ADLs, such as Darwin [16], Rapide [15], Wright [2] and Acme [10] can express run time architectural provided they have been previously specified.

While ADLs aim at describing SAs for the purposes of analysis and system generation, Architectural Modification Languages (AMLs) focus on describing changes to architecture descriptions and are, thus, useful for introducing unplanned changes to deployed systems. The Extension Wizard's modification scripts, C2's AML [17], and Clipper [1] are examples of such languages. Similarly, Architectural Constraint Languages (ACLs) have been used to restrict the system structure using imperative [5] as well as declarative [16] specifications.

These languages endow SA design approaches with mechanisms to specify reconfigurations. However, the latter focus on the high-level entities of architectures, rather than on the coordination glue code. A ReCooPLa, in contrast, is targets the whole coordination pattern of a system and is oriented towards reconfiguration analysis.

Also related to ReCooPLa is the GP programming language presented in [25]. It is a language for solving graph problems, based on a notion of graph transformation and four operators shown to be Turing-complete. Like GP, ReCooPLa actuates over a graph-based structure to perform modifications. While GP does so with program rules, ReCooPLa defines reconfiguration methods based on primitive (coordination-oriented) constructs.

3 Reconfiguration Model

This section provides an informal account of the reconfiguration model, which has been introduced and formalised in [19, 20]. In particular, it introduces the notions of a coordination pattern and coordination-based reconfiguration, which are later embodied in the constructs of ReCooPLa.

3.1 Coordination Protocols

A coordination protocol works as a *glue code* to define and constrain the interaction between components or services of a system. In this model, it is called a *coordination pattern* and regarded as a reusable and composable architectural element. It is formalised as a graph of channels whose nodes are interaction points through which it can plug to other coordination patterns or services; edges are uniquely identified point-to-point communication devices with

a specific behaviour given by a channel typing system. Formally,

$$\rho \subseteq \mathcal{N} \times \mathcal{I} \times \mathcal{T} \times \mathcal{N},$$

where $\mathcal{N}$ is a set of nodes (to be precise, a node in a coordination pattern corresponds to a set of channel ends), $\mathcal{I}$ is a set of channel identifiers and $\mathcal{T}$ is a channel typing system. Set $\mathcal{T}$={sync, lossy, fifo, drain} is adopted in the sequel as the working channel typing system in the spirit of the Reo coordination language [4]. Nodes that are used exclusively for data input (respectively, output) constitute the input (respectively, output) ports of the pattern. All the others are classified as internal or mixed nodes.

Listing 1 presents two coordination patterns. Coordination pattern `cp1` comprises two channels: a channel `x1` of type sync, and channel `x2` of type lossy. Channel `x1` has an input node `a` and an output node `b.c`[1]. In turn, channel `x2` has an input node `b.c` (corresponding to output node of channel `x1`, once they are connected), and an output node `d`.

Listing 1 Two simple coordination patterns.

```
cp1: {(a, x1, sync, b.c), (b.c, x2, lossy, d)}
cp2: {(g, x3, sync, h.i.j), (h.i.j, x4, lossy, k),
      (h.i.j, x5, fifo, l)}
```

3.2 Coordination-based Reconfigurations

A reconfiguration is a modification of the original structure of a coordination pattern obtained through sequential or parallel application of parametrised elementary operations, which are called reconfiguration *primitives*.

Let ρ be a coordination pattern.The simplest reconfigurations are the identity (`id`) and the constant (`const(`ρ`)`) primitives. The former returns the original coordination pattern, while the latter replaces it with ρ.

The `par(`ρ`)` primitive sets the original coordination pattern in parallel with the ρ, without creating any connection between them. It is assumed, without loss of generality, that nodes and channel identifiers in both patterns are disjoint. Listing 2 presents the resulting coordination pattern, after applying `par(cp2)` to `cp1`.

Listing 2 Resulting coordination pattern after applying the `par` primitive.

```
cp1: { (a, x1, sync, b.c), (b.c, x2, lossy, d), (g, x3, sync, h.i.j),
       (h.i.j, x4, lossy, k), (h.i.j, x5, fifo, l)}
```

The `join(`N`)` primitive, where N is a set of nodes, creates a new node by merging all nodes in N, into a single one.

For instance, applying `join(a,g)` to `cp1` (*c.f.*, Listing 2) creates a connection on node `a.g`, as presented in Listing 3.

Listing 3 Resulting coordination pattern after applying the `join` primitive.

```
cp1: {(a.g, x1, sync, b.c), (b.c, x2, lossy, d),
(a.g, x3, sync, h.i.j), (h.i.j, x4, fifo, k), (h.i.j, x5, drain, l)}
```

The `split(`n`)` primitive, where n is a node, is dual to the `join` combinator because it breaks connections within a coordination pattern by separating all channel ends coincident

[1] Notation `b.c` is used to express the node {`b`,`c`}, where `b` and `c` are channel ends.

in n. Listing 4 presents the resulting coordination pattern, after applying `split(h.i.j)` to `cp1` from Listing 3. Notice that the ends composing node `h.i.j` are assigned to each channel that previously shared this node (viz. channels `x3`, `x4` and `x5`), in a non-deterministic way.

Listing 4 Resulting coordination pattern after applying the `split` primitive.

```
cp1: {(a.g, x1, sync, b.c), (b.c, x2, lossy, d), (a.g, x3, sync, h),
(i, x4, fifo, k), (j, x5, drain, l)}
```

Finally, the `remove(`c`)` primitive, where c is a channel identifier, removes channel c, if it exists, from the coordination pattern. In addition, if c was connected to other channel(s), these connections are also broken as it happens with `split`. Listing 5 presents the resulting coordination pattern, after applying `remove(x2)` to `cp1` from Listing 4. Notice how node `b.c` was split and its end `c` was removed along with channel `x2`. Again, this process is non-deterministic.

Listing 5 Resulting coordination pattern after applying the `remove` primitive.

```
cp1: { (a.g, x1, sync, b), (a.g, x3, sync, h), (i, x4, fifo, k),
       (j, x5, drain, l)}
```

These primitive operations are assumed to be applied in sequence. Their parallel application is also valid, but only when they can be shown to be mutually independent: *i.e.*, affecting separated substructures of the target coordination pattern. This possibility of composing primitive operations in sequence or parallel, allows for the definition of complex reconfigurations, referred to as *reconfiguration patterns*. Actually, they affect significant parts of a coordination pattern at a time, and are expected to be generic, parametric and reusable. The ReCooPLa language offers a way of specifying such combinations in an imperative-like style.

4 ReCooPLa: Reconfiguration Language

ReCooPLa is a language for designing coordination-based reconfigurations. As a DSLs tailored to the area of architectural reconfigurations, it makes possible to abstract away from specific details, such as the effect of each primitive operation and their actual application (whether in sequence or in parallel), as well as to hide their actual computation under a processor.

4.1 Overview

In ReCooPLa, a *reconfiguration* is a first class citizen, as much as functions are in some programming languages. In fact, these two concepts share characteristics: both have a signature (identifier and arguments) and a body which designates a specific behaviour. However, a reconfiguration is always applied to, and always returns, a coordination pattern. Additionally, reconfigurations accept arguments of the following data types: *Name*, *Node*, *Set*, *Pair*, *Triple*, *Pattern* and *Channel*.

The reconfiguration body is a list of different sorts of instructions. The main one concerns *application* of (primitive, or previously defined) reconfigurations, since this is the only way of modifying a coordination pattern. As auxiliar operations, ReCooPLa resorts to other constructs that mainly manipulate the parameters of a reconfiguration. In particular, they provide ways to declare, assign and manipulate local variables, for example, field selectors, the usual set connectives (union, intersection and subtraction), and an iterative control structure to iterate over the elements of a set.

In brief, ReCooPLa is a small language borrowing most of its constructs from imperative programming languages. Actually, reconfigurations are better expressed in a procedural/algorithmic way, which justifies the choice of an imperative style.

4.2 The Language

In the sequel, we introduce ReCooPLa by presenting (the most important) fragments of the underlying grammar. A number of constructs are defined for further reference in the paper. Formally, a sentence in ReCooPLa specifies one or more reconfigurations.

Reconfiguation

A reconfiguration (see Listing 6) is expressed similarly to a function. The header is composed of a reserved word `reconfiguration` followed by a unique identifier (the reconfiguration name) and a list of arguments, which may be empty. The body is a list of instructions as explained below. Arguments are aggregated by data type, differently from what happens in conventional languages where data types are replicated for every different argument.

Listing 6 Extended Backus–Naur Form (EBNF) notation for the *reconfiguration* production.

```
reconfiguration
        : 'reconfiguration' ID '(' args* ')' '{' instruction+ '}'
args    : arg (';' arg)*
arg     : datatype ID (',' ID)*
```

The constructor for a reconfiguration is given by: $rcfg(n, t_1, a_1, \ldots, t_n, a_n, b)$, where n is the name of the reconfiguration; each a_i is an argument of type t_i; and b is the body of the reconfiguration.

Data types

ReCooPLa builds on a small set of data types: primitive (*Name* and *Node*), generic (*Set*, *Pair* and *Triple*) and structured (*Pattern* and *Channel*). *Name* is a string and represents a channel identifier or a channel end. *Node*, although considered as a primitive data type, is internally seen as a set of names, to maintain compatibility with its definition in Section 3. The generic data types (based on the Java generics) specify a type by its contents, as seen in Listing 7.

Listing 7 EBNF notation for the *datatype* production.

```
datatype: ...
        | ('Set' | 'Pair' | 'Triple') '<' datatype '>'
```

Structured data types have an internal state, matching their definition in Section 3. Each instance of these types is endowed with attributes and operations, which can be accessed using selectors (later in this section).

The construct of a data type is either given as $T()$ or $T_G(t)$, where T is a ReCooPLa data type and t is a subtype of a generic data type T_G.

Reconfiguration body

The reconfiguration body is a list of instructions, where each instruction can be a declaration, an assignment, an iterative control structure, or an application of a reconfiguration. A declaration is expressed as usual: a data type followed by an identifier or an assignment.

In its turn, an assignment associates an expression, or an application of a reconfiguration, to an identifier. The respective constructs are, then, $decl(t, v)$ and either $assign(t, v, e)$ or $assign(v, e)$, where t is a data type, v a variable name; and e an expression.

The control structure `forall` is used to iterate over a set of elements. Again, a list of instructions defines the behaviour of this structure. In Listing 8 it can be seen the corresponding production rule.

Listing 8 EBNF notation for the *forall* production.

```
forall  : 'forall' '(' datatype ID ':' ID ')' '{' instruction+ '}'
```

The constructor for this iterative control structure is given as $forall(t, v_1, v_2, b)$, where t is a data type, v_1, v_2 are variables and b is a set of instructions.

The application of a reconfiguration, (*c.f.*, `reconfiguration_apply` production in Listing 9), is expressed by an identifier followed by the `'@'` operator and a reconfiguration name. The latter may be a primitive reconfiguration or any other previously declared. The `'@'` operator stands for *application*. A reconfiguration is applied to a variable of type *Pattern*. In particular, this variable may be omitted (optional identifier in the production rule `reconfiguration_apply`); when this is the case, the reconfiguration called is applied to the original pattern. This typical usage can be seen in Listing 13

Listing 9 EBNF notation for the *reconfiguration_apply* production.

```
reconfiguration_apply
        : ID? '@' reconfiguration_call
reconfiguration_call
        : ('join'|'split'|'par'|'remove'|'const'|'id'|ID) op_args
```

Application is called either as $@(c)$ or $@(p, c)$, where p is a *Pattern* and c a reconfiguration call. Each reconfiguration call also has its own constructor: $r(a_1, \ldots, a_n)$, for r a reconfiguration name, and each a_i one of its arguments.

Operations

An expression is composed of one or more operations. They can be specific constructors for generic data types, including nodes, or operations over generic or structured data types. Listing 10 shows examples of these types of operation. Each constructor is defined as a reserved word (`S` stands for *Set*, `P` for *Pair*, `T` for *Triple* and `N` for *Node*); and a list of values which is expected to comply to the data type involved. The corresponding production rule is given in Listing 10 and exemplified in Listing 11.

Listing 10 EBNF notation for the constructor production.

```
constructor
        : 'P' '(' expression ',' expression ')'
        | 'T' '(' expression ',' expression ',' expression ')'
        | 'S' '(' ( expression (',' expression)*)? ')'
        | 'N' '(' ID (',' ID)* ')'
```

For the Set data type, ReCooPLa provides the usual binary set operators: '+' for union, '−' for subtraction and '&' for intersection. For the remaining data types (except *Node* and *Name*), selectors are used to apply the operation, as shown in Listing 12 (production rule `operation`). Symbol `#` is used to access a specific channel from the internal structure of a pattern.

Listing 11 *Constructors* input example.

```
Pair<Node> a = P(n1, n2);
Triple<Pair<Node> b = T(a, P(n1,n2), P(n3,n4));
Set<Node> c = S(n1, n2, n3, n4, n5, n6);
Node d = N(e1, e2);
```

Listing 12 EBNF notation for the `operation` and `attribute_call` productions.

```
operation
        : ID ('#' ID)? '.' attribute_call
attribute_call
        : 'in' ( '(' INT ')' )?
        | 'out' ( '(' INT ')' )?
        | 'ends' '(' ID ')'
        | 'name' | 'nodes' | 'names' | 'channels'
        | 'fst' | 'snd' | 'trd'
```

An `attribute_call` corresponds to an attribute or an operation associated to the last identifier, which must correspond to a variable of type *Channel*, *Pattern*, *Pair* or *Triple*. The list of attributes/operations in the language is presented in Listing 12 and described below:

- `in`: returns the input ports from the *Pattern* and *Channel* variables. It is possible to obtain a specific port refered by an optional integer parameter indexing a specific entry from the set (seen as an array).
- `out`: returns the output ports from the *Pattern* and *Channel* variables. The optional parameter can be used as explained for the `in` *attribute call*.
- `name`: returns the name of a *Channel* variable, i.e., a channel identifier.
- `ends`: returns the ends of a *Channel* variable in the context of a *Pattern* given as parameter.
- `nodes`: returns all input and output ports plus all the internal nodes of a *Pattern* variable.
- `names`: returns all channel identifiers associated to a *Pattern* variable.
- `channels`: returns a set of channels associated to a *Pattern* variable.
- `fst`, `snd`, `trd`: act, respectively, as the first, second and third projection from a tuple (*Pair* and *Triple* variables).

All these operations give rise to their own language constructors. For example, the constructor of a *Pair* data type is $P(e_1, e_2)$, where e_1, e_2 are expressions; for field selection $.(v, c)$ is used, where v is a variable and c a call to an operation; for set union we write $+(s_1, s_2)$, with s_1, s_2 variables of type *Set*. The remaining constructors are defined similarly.

Listing 13 shows an example of valid ReCooPLa sentences which declare two reconfigurations: `removeP` and `overlapP`. The former removes from a coordination pattern an entire set of channels by applying the `remove` primitive repeatedly. The latter sets a coordination pattern in parallel with the original one, using the `par` primitive, and performs connections between the two patterns by applying the `join` primitive with suitable arguments.

5 ReCooPLa: Language Compilation

This section introduces the reconfiguration engine, which executes reconfigurations specified in ReCooPLa, and the correspnding translation schema into Java code.

Listing 13 ReCooPLa input example.

```
reconfiguration removeP (Set<Name> Cs ) {
    forall ( Name n : Cs) {
        @ remove(n);
    }
}
reconfiguration overlapP(Pattern p; Set<Pair<Node>> X) {
    @ par (p);
    forall(Pair<Node> n : X) {
        Node n1, n2;
        n1 = n.fst;
        n2 = n.snd;
        Set<Node> E = S(n1, n2);
        @ join(E);
    }
}
```

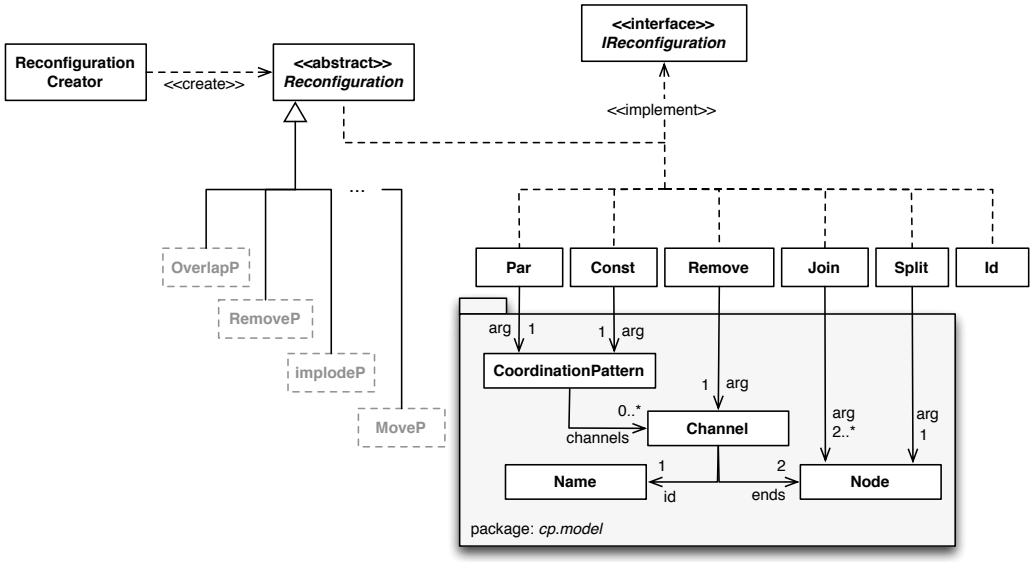


Figure 1 The Reconfiguration Engine model.

5.1 Reconfiguration Engine

As it often happens with domain specific languages, ReCooPLa is translated into a subset of Java, which is then recognised and executed by an engine. This engine, referred to as the *Reconfiguration Engine*, is developed in Java to execute reconfigurations specified in ReCooPLa over coordination patterns, which are defined in CooPLa [20], a lightweight language to define the graph-like structure of coordination patterns. The model of the engine is as simple as it can be, taking into account only a few entities. Figure 1 presents the corresponding Unified Modelling Language (UML) class diagram.

Package *cp.model*, represented as a shaded diagram, concerns the model of a coordination pattern. This is actually, the implementation of the formal model presented in Section 3. Both CoordinationPattern and Channel classes provide attributes and methods that match

the attributes and operations of the *Pattern* and *Channel* types in ReCooPLa. For instance, the attribute `nodes` of the *Pattern* type has its corresponding method `getNodes()` in the CoordinationPattern class.

The remaining entities of the diagram are concerned with reconfigurations themselves, and assumed to belong to a *cp.reconfiguration* package. Clearly, classes Par, Const, Remove, Join, Split and Id are the implementation of the corresponding primitive reconfigurations also introduced in Section 3. The relationships with the elements of the *cp.model* package define their arguments. Moreover, these classes have a common implicit method (given by the interface IReconfiguration): apply(CoordinationPattern p), where the behaviour of these primitives is defined as the combined effect of their application to the coordination pattern p given as an argument.

The Reconfiguration class represents a generic reconfiguration that requires its concrete classes to implement the apply(CoordinationPattern p) method. The careful reader may have noticed that the concrete classes of Reconfiguration are greyed-out, and also that they are not all presented. This is where the most interesting part of the engine comes into play. In fact, there are no such concrete classes at design time. All of them are created dynamically, at run time, by the ReconfigurationCreator class, taking advantage of reflection in Java Virtual Machine (JVM) and working packages like *Javassist*[2]. This implementation follows a similar approach to the well-known Factory design pattern, but instead of creating instances, it creates concrete classes of Reconfiguration. The idea is that each reconfiguration definition within a ReCooPLa specification gives rise to a newly created class with an apply(CoordinationPattern p) method. Then, the content of such method is derived from the content of the ReCooPLa reconfiguration and added dynamically, via reflection, to the created class. Once the classes are loaded into the running JVM, the application of reconfigurations becomes as simple as calling the apply method from instances of such classes.

However, for this to be possible, it is first necessary to correctly translate ReCooPLa constructs into the code accepted by the Reconfiguration Engine. Section 5.2 goes through the details of such a translation.

The application of reconfigurations is also specified in ReCooPLa, taking into consideration the coordination patterns defined in CooPLa (which may be imported to ReCooPLa, a detail omitted in this paper). A script-based structure is assumed to define how reconfigurations are concretely applied to coordination patterns. A glimpse of how this can be achieved is unveiled in Listing 14.

Listing 14 Sketch of a reconfiguration script.

```
import "patterns.cpl", "reconfigurations.rcpl"
reconfigure (UserUpdate sq1)
    UserUpdate sq2 ;
    sq1 @ OverlapP(sq2, S(P(sq1#f2.out[0],sq2#s1.in[0])));
```

Its meaning is straightforward. First, the necessary definitions (reconfigurations and patterns) are imported. Then, the `reconfigure` reserved word marks the beginning of the reconfiguration script. Parameter `UserUpdate sq1` defines a *UserUpdate* coordination pattern (*c.f.* Figure 2) in some configuration. This is not limited to one pattern and in the future it may be a pointer to some ADL specification, where coordination patterns play the role of connectors. The declaration `UserUpdate sq2` defines a fresh instance of this coordination pattern. Finally, an OverlapP reconfiguration is applied on `sq1` with appropriated arguments.

[2] `http://www.javassist.org`

5.2 ReCooPLa Translation

Throughout this subsection, it is assumed the existence of Java classes to match the types in ReCooPLa. This means that, besides the classes already mentioned in Figure 1, the following ones are also assumed: Pair, with a getFst() and a getSnd() methods to access its fst and snd attributes; Triple, extending Pair with an attribute trd and method getTrd(); and the LinkedHashSet from the *java.util* package, which is abbreviated to LHSet for increased readability. Moreover, keep exposition simple, details about reflection will be ignored or abstracted. For instance, method mkClass(cl, t_1, a_1, ..., t_n, a_n, b) abstracts the dynamic creation of a Reconfiguration class with name cl; attributes a_1, ..., a_n of type t_1, ..., t_n, respectively; and method apply with body b, which always ends with a return p instruction, where p is the argument of apply.

This said, the translation of ReCooPLa constructors into the Reconfiguration Engine is given by the rule-based function $\mathcal{T}$(C), where C is a constructor of ReCooPLa as presented in Section 4 and defined as shown in Table 1.

Table 1 Translation rules for ReCooPLa constructs.[3]

$\mathcal{T}(rcfg(n, t_1, a_1, ...t_n, a_n, b))$	$\rightarrow$	mkClass(n, $\mathcal{T}(t_1)$, a_1, ... $\mathcal{T}(t_n)$, a_n, $\mathcal{T}(b)$)
$\mathcal{T}(T())$	$\rightarrow$	T
$\mathcal{T}(T_G(t))$	$\rightarrow$	T_G<$\mathcal{T}(t)$>
$\mathcal{T}(Set(t))$	$\rightarrow$	LHSet<$\mathcal{T}(t)$>
$\mathcal{T}(decl(t, v))$	$\rightarrow$	$\mathcal{T}(t)$ v
$\mathcal{T}(assign(t, v, e))$	$\rightarrow$	$\mathcal{T}(decl(t, v)) = \mathcal{T}(e)$
$\mathcal{T}(assign(v, e))$	$\rightarrow$	v = $\mathcal{T}(e)$
$\mathcal{T}(forall(t, v_1, v_2, b))$	$\rightarrow$	for($\mathcal{T}(t)$ v_1 : v_2){$\mathcal{T}(b)$}
$\mathcal{T}(@(r(e_1, \ldots, e_n)))$	$\rightarrow$	r rec = new r($\mathcal{T}(e_1)$, ..., $\mathcal{T}(e_n)$); rec.apply(p)
$\mathcal{T}(@(r(p, e_1, \ldots, e_n)))$	$\rightarrow$	r rec = new r($\mathcal{T}(e_1)$, ..., $\mathcal{T}(e_n)$); rec.apply(p)
$\mathcal{T}(P(e_1, e_2))$	$\rightarrow$	new Pair($\mathcal{T}(e_1)$, $\mathcal{T}(e_2)$)
$\mathcal{T}(T(e_1, e_2, e_3))$	$\rightarrow$	new Triple($\mathcal{T}(e_1)$, $\mathcal{T}(e_2)$, $\mathcal{T}(e_3)$)
$\mathcal{T}(S(e_1, \ldots, e_n))$	$\rightarrow$	new LHSet<T>(){{add($\mathcal{T}(e_1)$); ...; add($\mathcal{T}(e_n)$); }} [4]
$\mathcal{T}(N(n_1, \ldots, n_n))$	$\rightarrow$	new Node(new LHSet<String>(){{add(n_1); ...; add(n_n); }})
$\mathcal{T}(+(s_1, s_2))$	$\rightarrow$	(new LHSet(s_1)).addAll(s_2)
$\mathcal{T}(-(s_1, s_2))$	$\rightarrow$	(new LHSet(s_1)).removeAll(s_2)
$\mathcal{T}(\&(s_1, s_2))$	$\rightarrow$	(new LHSet(s_1)).retainAll(s_2)
$\mathcal{T}(\#(p, c))$	$\rightarrow$	p.getChannel(c)
$\mathcal{T}(.(v, c))$	$\rightarrow$	v.$\mathcal{T}(c)$
$\mathcal{T}(in(i))$	$\rightarrow$	getIn(i)
$\mathcal{T}(out(i))$	$\rightarrow$	getOut(i)
$\mathcal{T}(ends(p))$	$\rightarrow$	getEnds(p)
$\mathcal{T}(oper())$	$\rightarrow$	getOper()

[3] By convention n is used for identifiers; t, t_i for data types; a_i for arguments; b for set of instructions; T for non-generic data type; T_G for generic data type, except *Set*; v, v_i for local variables; e, e_i for expressions; p for patterns; s_i for sets; c for channel names; i for numbers; and finally *oper* for the operations enumerated in Section 4.2.

[4] T comes from the context where the construct appears or the type of the composing expressions e_i.

Listing 15 Example of a ReCooPLa reconfiguration translated.

```
public class OverlapP extends Reconfiguration {
    private CoordinationPattern p;
    private LHSet<Pair<Node, Node>> X;
    public OverlapP(CoordinationPattern arg1,
        LHSet<Pair<Node, Node>> arg2) {
        this.p = arg1;
        this.X = arg2;
    }
    public CoordinationPattern apply(CoordinationPattern pat) {
        Par par;
        Join join;
        par = new Par(this.p);
        par.apply(pat);
        for(Pair<Node> n : this.X) {
            Node n1, n2;
            n1 = n.getFst();
            n2 = n.getSnd();
            LHSet<Node> E = new LHSet<Node>() {{
                add(n1); add(n2);
            }};
            join = new Join(E);
            join.apply(pat);
        }
        return pat;
    }
```

It goes without saying that a translation can only occur when the ReCooPLa specification is syntactically and semantically correct. The ReCooPLa parser ensures syntactic correctness; on the other hand, a semantic analyser is defined to report errors concerning structure, behaviour and data types. Its definition is out of the scope of this paper.

Listing 15 shows the result of applying the translation rules to the *OverlapP* ReCooPLa reconfiguration documented in Listing 13.

6 Example

Consider a company that sells training courses on line and whose software system originally relied on the following four components: Enterprise Resource Planner (ERP), Customer Relationship Management (CRM), Training Server (TS) and Document Management System (DMS). In seeking an expedite expansion of the company and its information systems, a major software refactoring project was launched adopting a SOA solution. This entailed the need to change from the original structure of monolithic components into several services and their integration and coordination with respect to the different business activities.

One of the most important activities for the company concerns the updating of user information, which is accomplished taking into account the corresponding new user update services derived from the original ERP, CRM and TS components. Originally such an update was designed to be performed sequentially as shown in the coordination pattern of Figure 2.

However, other configurations were considered and studied taking advantage of the ReCooPLa language and the underlying reconfiguration reasoning framework. For instance,

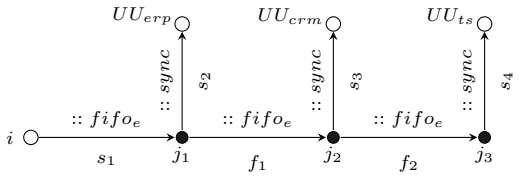


Figure 2 The User update coordination pattern. Each channel is identified with a unique name and a type (::t notation). It defines an instance of a sequencing pattern, where UU_{erp} executes first, then UU_{crm} and finally UU_{ts} with data entering in port i. Graphically, white circles represent input and output nodes while black ones represent mixed nodes.

Listing 16 `implodeP` reconfiguration pattern.

```
reconfiguration implodeP(Set<Node> X; Set<Name> Cs) {
        @ removeP(Cs);
        @ join (X);
}
```

another configuration for the user update activity may be given by the coordination pattern in Figure 3. This can be obtained from the initial pattern by application of a reconfiguration that collapses nodes and channels into a single node. In ReCooPLa, this is easy to define, as shown in Listing 16, resorting to removeP already defined in Listing 13.

This reconfiguration pattern takes as parameters the set of nodes and channels realtive to the strucutre one pretends to implode. Channels are removed and the nodes are joined. The translation mechanism of ReCooPLa specifications produces a Java class similar to the one presented in Listing 17.

Listing 17 ImplodeP class generated.

```
public class ImplodeP extends Reconfiguration {
    private LHSet<Node> X;
    private LHSet<Name> Cs;
    public OverlapP(LHSet<Node> arg1,LHSet<Name> arg2) {
        this.X = arg1;
        this.Cs = arg2;
    }
    public CoordinationPattern apply(CoordinationPattern pat) {
        RemoveP removeP;
        Join join;
        removeP = new RemoveP(this.Cs);
        removeP.apply(pat);
        join = new Join(this.X);
        join.apply(pat);

        return pat;
    }
}
```

In this example, applying $implodeP(\{j_1, j_2, j_3\}, \{f_1, f_2\})$ to the original coordination pattern would result in the one depicted in Figure 3, where (for reading purposes) node k is used to represent the union of j_1 and j_2.

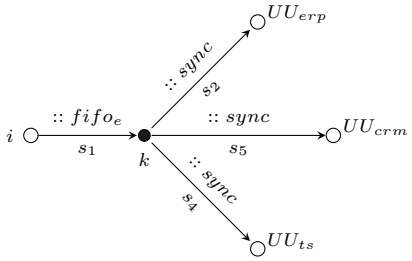


Figure 3 The User update coordination pattern reconfigured. It defines an instance of a parallel pattern, where UU_{erp}, UU_{crm} and UU_{ts} execute in parallel with data entering in port i.

7 Conclusions and Future Work

The paper introduces ReCooPLa, a DSL for the design of coordination-based reconfigurations. These reconfigurations act, through the application of primitive atomic operations, over a graph-based structure, which is an abstract representation of the coordination layer of a SOA-based system. ReCooPLa resorts to a Reconfiguration Engine that, via reflection, processes and applies such reconfigurations.

ReCooPLa differs from other architecture-oriented languages in the sense that it focus on reconfigurations rather than on the definition of architectural elements like components, connectors and their interconnections. Moreover, the language and the underlying approach is intended to target the early stages of software development; i.e., the design of reconfigurations and their analysis against requirements of the system. However, this approach may be lifted to the dynamic setting by mapping the code of each reconfiguration and coordination pattern to the actual coordination layer of a system. This would allow to reconfigure deployed systems offering an abstract, but effective way of planning such reconfigurations.

As future work, it is planned the full integration of ReCooPLa with the framework for reconfiguration analysis conceptualised in [19, 20]. In particular, it is intended to extend the language to cope with the probabilistic coordination model introduced in [22].

Acknowledgements. This work is partly funded by ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT, the Portuguese Foundation for Science and Technology, within project `FCOMP-01-0124-FEDER-028923`. Nuno Oliveira is supported by an Individual Doctoral Grant from FCT, with reference `SFRH/BD/71475/2010`.

References

1 B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: a reconfiguration language for distributed systems. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems, 1994*, pages 15–22, 1994.

2 R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

3 G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computer Surveys*, 23(1):49–90, March 1991.

4 F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical. Structures in Comp. Sci.*, 14(3):329–366, June 2004.

5 R. Balzer. Enforcing architecture constraints. In *Proceedings of ISAW'96*, pages 80–82, NY, USA, 1996. ACM.

6 E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.

7 R. Bruni, A. Lluch-Lafuente, U. Montanari, and E. Tuosto. Style-based architectural reconfigurations. *Bulletin of the European association for theoretical computer science*, 94:161–180, February 2008.

8 P.-C. David, T. Ledoux, M. Léger, and T. Coupaye. Fpath and fscript: Language support for navigation and reliable reconfiguration of fractal architectures. *Annals of Telecommunications - Annales des Télécommunications*, 64(1-2):45–63, 2009.

9 T. Erl. *SOA Design Patterns*. Prentice Hall PTR, NJ, USA, 1st edition, 2009.

10 D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of the CASCON'97*, pages 7–. IBM Press, 1997.

11 P. Hnětynka and F. Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In I. Gorton, G. T. Heineman, I. Crnković, H. W. Schmidt, J. A. Stafford, C. Szyperski, and K. Wallnau, editors, *Component-Based Software Engineering*, volume 4063 of *LNCS*, chapter 27, pages 352–359. Springer, 2006.

12 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.

13 C. Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, Amsterdam, The Netherlands, 2011.

14 C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming*, 76(1):23–36, 2011.

15 D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, September 1995.

16 J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of SIGSOFT'96*, page 3–14, NY, USA, 1996. ACM.

17 N. Medvidovic. Adls and dynamic architecture changes. In *Proceedings of ISAW'96*, pages 24–27, NY, USA, 1996. ACM.

18 M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys.*, 37(4):316–344, December 2005.

19 N. Oliveira and L. S. Barbosa. On the reconfiguration of software connectors. In *Proceedings of SAC'13*, pages 1885–1892, NY, USA, 2013. ACM.

20 N. Oliveira and L. S. Barbosa. Reconfiguration mechanisms for service coordination. In M. H. Beek and N. Lohmann, editors, *Web Services and Formal Methods*, volume 7843 of *LNCS*, pages 134–149. Springer, 2013.

21 N. Oliveira, M. J. V. Pereira, P. R. Henriques, and D. da Cruz. Domain-specific languages: a theoretical survey. In *INForum'09*, pages 35–46, Lisbon, Portugal, September 2009.

22 N. Oliveira, A. Silva, and L. S. Barbosa. Quantitative analysis of Reo-based service coordination. In *Proceedings of SAC'14*, volume 2, pages 1247–1254. ACM, March 2014.

23 P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of ICSE'98*, pages 177–186, WA, USA, 1998. IEEE Computer Society.

24 D. E. Perry. An overview of the state of the art in software architecture. In *Proceedings of ICSE'97*, pages 590–591, NY, USA, 1997. ACM.

25 D. Plump. The graph programming language GP. In S. Bozapalidis and G. Rahonis, editors, *Algebraic Informatics*, volume 5725 of *LNCS*, chapter 6, pages 99–122. Springer, Berlin, Heidelberg, 2009.

26 A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *Proceedings of SEAMS'10*, pages 49–58, NY, USA, 2010. ACM.

27 A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.
28 A. L. Wolf. Succeedings of the isaw-2. *SIGSOFT Softw. Eng. Notes*, 22(1):42–56, January 1997.

A Workflow Description Language to Orchestrate Multi-Lingual Resources

Rui Brito and José João Almeida

University of Minho
Department of Informatics
{ruibrito,jj}@di.uminho.pt

Abstract

Texts aligned alongside their translation, or Parallel Corpora, are a very widely used resource in Computational Linguistics. Processing these resources, however, is a very intensive, time consuming task, which makes it a suitable case study for High Performance Computing (HPC).

HPC underwent several recent changes, with the evolution of Heterogeneous Platforms, where multiple devices with different architectures are able to share workload to increase performance.

Several frameworks/toolkits have been under development, in various fields, to aid the programmer in extracting more performance from these platforms. Either by dynamically scheduling the workload across the available resources or by exploring the opportunities for parallelism. However, there is no toolkit targeted at Computational Linguistics, more specifically, Parallel Corpora processing. Parallel Corpora processing can be a very time consuming task, and the field could definitely use a toolkit which aids the programmer in achieving not only better performance, but also a convenient and expressive way of specifying tasks and their dependencies.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases workflow, orchestration, parallelism, domain specific languages, corpora

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.77

1 Introduction

It is widely accepted that tasks can be very difficult to manage, mainly when they have many steps, often with time consuming tools, interdependent tasks and huge datasets. As the complexity of applications increases, so does the difficulty to efficiently manage the different available resources. Users have to adopt frameworks/toolkits, as a way to improve their path to achieve the desired goals. The aim of this paper is to provide a toolkit targeted at Computational Linguistics, which may also benefit other fields that require workflow management. Texts aligned alongside their translation, or parallel corpora, are considered very rich resources for machine translation, whose formats usually follow Translation Memory eXchange (TMX) or Probabilistic Translation Dictionaries (PTD). Tools to aid machine translation based on parallel corpora include text aligners (e.g. Moses [6]), morphological taggers (e.g. Apertium [4] and Freeling [8]) and dictionary generators (e.g. NATools [10]). Some of these tools are computationally intensive, taking too long to display useful results. Most data is the result of a complex task chain and ever-changing data, which makes the results difficult to predict and errors difficult to detect. This type of work is currently done either by hand or by very crude scripts, but could be significantly improved if the linguistics community could access a toolkit to create and manage the interrelated parallel corpora processing tasks and take advantage of current heterogeneous computing systems. The lack of such a tool is the main motivation to pursue this work, developing a Workflow Description Language (WDL), which is a DSL to orchestrate Multi-Lingual resources.


© Rui Brito and José João Almeida;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 77–83

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Previous work of the authors on the Per-Fide [1] project gained insight on the requirements to specify and develop a toolkit to chain tasks and to handle large data sets with different data formats. The task chain usually contains many steps, from different tools, which may change frequently, making it difficult to keep track of the results being produced. The tasks involved are mostly irregular, where memory access patterns may not be predictable, with consequent penalties on execution time. This brings forth the need to develop a toolkit to describe tasks, their chaining, their dependencies, their types and with the need to provide environment details to better take advantage of current heterogeneous computing systems.

Given a set of tools, the user should be able to specify tasks to be executed and visually follow the workflow execution phases and results. This can be achieved with a flow-graph. Some work has been done in this area, particularly with the Makefile::Parallel [9] tool, which, provides a way of describing tasks and workflows to be performed in either a cluster environment or a single machine.

The expected contribution of the work described in this paper is a toolkit to better organize, manage and modify on the fly. The toolkit will also make an efficient use of the available computational resources, by dynamically scheduling the workload with high-level directives that abstract efficient implementations.

The Section 2 begins by describing the Workflow Description Language and its aims. Section 3 presents some examples of the language, as well as some use case scenarios. Finally, section 4 presents our conclusions on the work already done.

2 Workflow Description Language

The main goal of this paper is to describe an efficient toolkit to orchestrate computational linguistics tasks. The user specifies through a WDL how individual tasks are performed. The main features of the toolkit are as follows:

- The toolkit will be environment-aware, and thus, take advantage as best as possible of the computing units and systems available;
- The toolkit should also be able to revert back to its most recent stable state (if the task is aborted);
- Primitives to generate reports, diagrams and type signature should be provided;
- There should be a mode in which the user provides heuristics on how to best use the available computing units;
- The user should be able to specify mapReduce [3, 7, 2] functions to achieve certain tasks;
  - A cleaner and more elegant way of specifying tasks;
  - Efficient low-level implementations are hidden from the end user;
- The WDL should be Algebra based, providing:
  - Familiar operations such as composition, which makes the language clearer;
- The toolkit should contemplate not only complex cluster systems, but also single user systems.

Some of the generic goals of the toolkit to be developed include:

- A declarative approach;
- Separation of elemental/independent tools;
- Separation of resources/machines;
- Reuse of existing research;
- Expressive power.

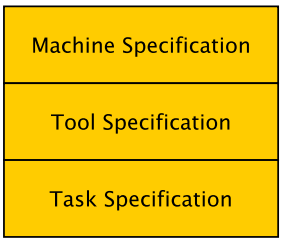


Figure 1 WDL Description.

The general organization of the WDL is described in the Figure 1. There are, essentially, three distinct parts to the language. A Machine Specification part, where all the computing systems available are described. In its simplest form, this means specifying hosts and users for remote machines. In the case of a cluster environment, there are primitives to find out which nodes are free, to ask for a full node (or nodes), since most cluster schedulers (e.g. PBS or Maui [5]) never supply the user with a full node, unless asked to. These are called execution attributes. There are also primitives to ascertain, for example, if a certain tool is available in a specific machine. Primitives of this kind are called existence attributes.

Then, there is a Tool Specification part, where all the tool to be used are described, along with any eventual execution attributes. Finally, there is an Task Specification part, where the Tasks are organized in a workflow.

This paper focus mostly on parts 2 and 3.

This WDL grammar is being developed using Perl YAPP, a Perl extension for generating and using LALR parsers. This decision is mainly because of Perl's expressive power and Regular Expressions, which make it easier to develop the kind of language we're trying to achieve.

As it can be seen from the grammar presented in Table 1, the user begins by specifying tools, which, in their most distilled form, are a collection of actions paired with an in and out type definition. The *action* production provides a way to specify how a certain tool is run. As it can be seen from the grammar, it can be either a bash script, or Perl code which is preprocessed before execution. The *execAttr* production's goal is to provide execution details for the tool, such has maximum desired walltime, whether it can run on a GPU or in Intel's MIC micro-architecture. In the case of a cluster environment, how many cores and how many nodes are needed for a specific tool. Eventually, more attributes will be added, or even changed. Once the tools are specified, the user begins specifying tasks. The production shown here (*taskSpec*), while not complete, hints at the direction we're aiming.

3 WDL Examples

Listing 1 presents how the syntax expressed in the grammar can be used to specify tools and tasks. First, the tool's type signature is specified, then a set of execution attributes followed by how to execute the tool. A couple of conventions were adopted, for example, for a given resource named corpus.pt.en.tmx:

- $B1 is the name of the resource (corpus);
- $1 is the name of the resource coupled with its extension (corpus.pt.en.tmx).

It is also possible to specify variable filenames, or type extensions, as seen in the *tmx2tmxa* tool definition in listing 1. This way, the user can make variable parts of a filename part of the type definition (e.g. language pairs).

Table 1 Workflow Description Language Grammar.

wdl	→	'%tools' toolDefs '%tasks' taskSpecs
		;
toolDefs	→	toolName ':' signature tool
		| toolDefs toolName ':' signature tool
		;
signature	→	type_id '→' type_id
		;
tool	→	'{' action execAttrs split join '}'
		;
action	→	'@BASH' '{' STRING '}'
		| '@PERL' '{' STRING '}'
		;
execAttrs	→	execAttrs execAttr
		| ε
execAttr	→	'wall='wall
		| 'gpu=' bool
		| 'mic=' bool
		| 'ncores=' INT
		| 'nnodes=' INT
		| ...
		;
split	→	'split=' action
		| ε
		;
join	→	'join=' action
		| ε
		;
taskSpecs	→	taskSpecs task
		| ε
		;
task	→	toolName '(' args ')'
		| toolName '.' toolName '(' args ')'
		| '#make' '(' args ')'
		| '#span' '(' args ')'
		;
args	→	ID
		| args ',' ID
		;

Listing 1 WDL Example.

```
1
2  tmx2ptd : tmx -> ptd {
3      @BASH {
4          nat-create -id $B1.ptd -tmx $1
5      }
6  }
7
8  tmx2tmxa : $l1.$l2.tmx -> tmxa {
9      @BASH {
10         tmx2tmxa -l $l1-$l2 -f $1 -o $B1.$l1.$l2.tmxa
11     }
12 }
13
14 tmx2ptda : tmxa -> ptda {
15     @BASH {
16         tmxa-lemmatizer -i $1 -o $B1.tmxpl
17         nat-create -id $B1.ptda -tmx $B1.tmxpl
18     }
19 }
20
21 tmx2tmxa (corpus.pt.en.tmx)
```

Figure 2 Typical serial workflow.

An alternative to specifying tasks in a function call fashion is to write something like the following using function composition:

```
1  tmxa2ptda.tmx2tmxa (corpus.pt.en.tmx)
```

The resulting workflow can be seen in Figure 2.

For example, the following definition adds mapReduce primitives to the tmxa2ptda tool[1].

The resulting workflow can be seen in Figure 3.

More complex examples, given there are no dependencies, would allow tasks to execute in parallel in a Cluster environment, spawning several processes in a single machine, or even across multiple machines. This would allow for a more efficient use of the available computational resources. Communication is done preferably by rsync, since it is a safer and faster alternative to scp.

There is also ongoing work in modes in which the user writes:

```
1  #span(resource.src_type)
2  #make(resource.target_type)
```

The *#span* construct will traverse the graph generated from the type signatures of the tools and generate all the resources possible from the achievable nodes. The *#make* construct

[1] *tmxsplit* is a script that cuts a tmx when possible after a given number of lines. *ptdcat* is a script that concatenates the various parts of a ptd file.

Listing 2 WDL with MapReduce Example.

```
tmxa2ptda : tmxa -> ptda {
    @BASH {
        tmxa-lemmatizer -i $1 -o $B1.tmxpl
        nat-create -id $B1.ptda -tmx $B1.tmxpl
        }
        split=@BASH {
            tmxsplit -l 10000 $1
        }
        join=@BASH {
            ptdcat *.ptda > result.ptda
        }
}
```

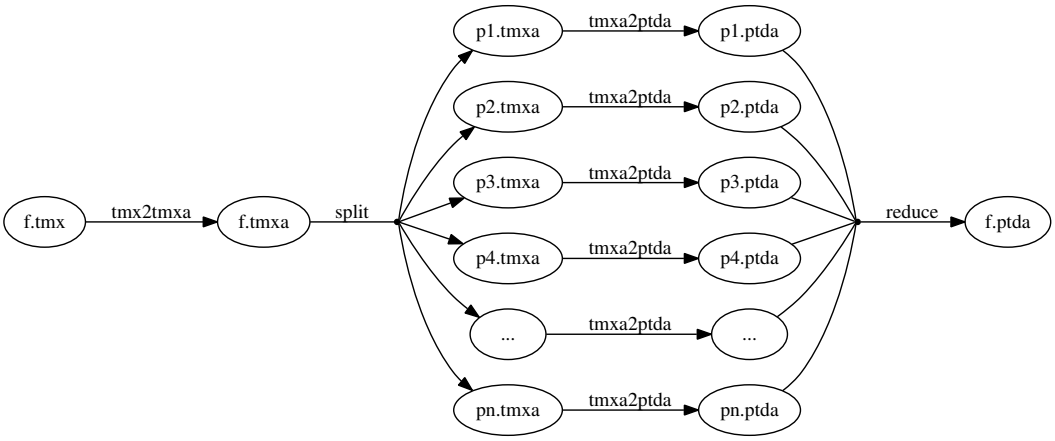


Figure 3 Workflow with MapReduce.

will take a *resource.target_type*, and, by inference, figure out the correct path to generate it from an existing *resource.src_type*. There will also be a built-in Perl preprocessor, so that the user can build more complex tasks.

4 Conclusions

Workflow orchestration and management is something very present in today's computing world and specially in fields that deal with intensive tools and very large data-sets. The lack of a tool to achieve this end, makes it even more difficult. In this short paper we introduced a new language to orchestrate workflows and to better manage existing computational resources. High level primitives and the direct integration of a scripting language (Perl) means we achieved a versatile format to express complex workflows.

Acknowledgments. This work is funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014

References

1. José João Almeida, Sílvia Araújo, Nuno Carvalho, Idalete Dias, Ana Oliveira, André Santos, and Alberto Simões. The Per-Fide corpus: A new resource for corpus-based terminology, contrastive linguistics and translation studies. In Tony Berber Sardinha and Telma São-Bento Ferreira, editors, *Working with Portuguese Corpora*, chapter 9, pages 177–200. Bloomsbury Publishing, April 2014.
2. M. Bhandarkar. MapReduce programming with Apache Hadoop. *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010.
3. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):1–13, 2008.
4. Mikel L. Forcada, Mireia Ginestí-Rosell, Jacob Nordfalk, Jim O'Regan, Sergio Ortiz-Rojas, Juan Antonio Pérez-Ortiz, Felipe Sánchez-Martínez, Gema Ramírez-Sánchez, and Francis M. Tyers. Apertium: a free/open-source platform for rule-based machine translation. *Machine Translation*, 25(2):127–144, 2011.
5. David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui Scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer-Verlag, 2001.
6. Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL'07, pages 177–180, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.
7. K. H. Lee, Y. J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with MapReduce: a survey. *ACM SIGMOD Record*, 40(4):11–20, 2012.
8. Lluís Padró and Evgeny Stanilovsky. FreeLing 3.0: Towards Wider Multilinguality. *LREC*, pages 2473–2479, 2012.
9. Alberto Simões, Rúben Fonseca, and José João Almeida. Makefile::Parallel Dependency Specification Language. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par 2007*, pages 33–41, Rennes, France, August 2007. Springer-Verlag.
10. Alberto M. Simões and J. J. Almeida. NATools – a statistical word aligner workbench. *Procesamiento del Lenguaje Natural*, 31:217–224, Sep. 2003.

Converting Ontologies into DSLs

João M. Sousa Fonseca[1], Maria João Varanda Pereira[2], and Pedro Rangel Henriques[1]

1 **Centro de Ciência e Tecnologia da Computação (CCTC)**
Departamento de Informática, Universidade do Minho
Braga, Portugal
{jprophet89,pedrorangelhenriques}@gmail.com

2 **Centro de Ciência e Tecnologia da Computação (CCTC)**
Departamento de Informática e Comunicações,
Instituto Politécnico de Bragança
Bragança, Portugal
mjoao@ipb.pt

Abstract

This paper presents a project whose main objective is to explore the Ontological-based development of Domain Specific Languages (DSL), more precisely, of their underlying Grammar.

After reviewing the basic concepts characterizing Ontologies and Domain-Specific Languages, we introduce a tool, Onto2Gra, that takes profit of the knowledge described by the ontology and automatically generates a grammar for a DSL that allows to discourse about the domain described by that ontology.

This approach represents a rigorous method to create, in a secure and effective way, a grammar for a new specialized language restricted to a concrete domain. The usual process of creating a grammar from the scratch is, as every creative action, difficult, slow and error prone; so this proposal is, from a Grammar Engineering point of view, of uttermost importance.

After the grammar generation phase, the Grammar Engineer can manipulate it to add syntactic sugar to improve the final language quality or even to add semantic actions.

The Onto2Gra project is composed of three engines. The main one is OWL2DSL, the component that converts an OWL ontology into an attribute grammar. The two additional modules are Onto2OWL, converts ontologies written in OntoDL (a light-weight DSL to describe ontologies) into standard OWL, and DDesc2OWL, converts domain instances written in the DSL generated by OWL2DSL into the initial OWL ontology.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases ontology, OWL, RDF, languages, DSL, grammar

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.85

1 Introduction

The use of domain-specific languages (DSL) enables a quick interaction with different domains, thereby taking a greater impact on productivity because there is no need for special or deep programming skills to use that language [3]. However, to create a domain-specific languages is a thankless task, which requires the participation of language engineers, which are (usually) not experts in the domain for which the language is targeted [4].

The work hereby presented takes advantage of the processable nature of OWL[1] ontologies

[1] Web Ontology Language as defined at http://www.w3.org/TR/owl-features/

© João Manuel Sousa Fonseca, Maria João Varanda Pereira, and Pedro Rangel Henriques;
licensed under Creative Commons License CC-BY

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 85–92

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to generate DSLs from the enclosed domain knowledge. This is expected to automatize, at a certain extent, the language engineer task of bringing program and problem domains together.

Ontologies are usually created as only a scheme of the domain knowledge. But this is far from being a complete ontology. Ontologies also support instances of the concepts and their relations, but populating such *database* is a tremendous manual and time consuming routine.

An additional outcome of the proposed work is the possibility of populating ontologies from text files written in the new and automatically generated DSL. This would combine the best of both worlds. In practice, it is desired to take advantage of modelling the domain as an ontology from where a DSL (and its processor) can be extracted. The DSL processor can be specialized to convert its input into new OWL files containing the concept instances described in the input.

In this paper we will demonstrate that, given an abstract ontology describing a knowledge domain in terms of the concepts and the relations holding among them, it is possible to derive automatically a grammar, more precisely an attribute grammar, to define a DSL for that same domain.

The rest of the paper is organized as follows. Section 2 is used to discuss work by other authors similar to ours. Section 3 is devoted to an overview of Onto2Gra, discussing its functionality and introducing its architecture. The two modules already implemented, Onto2OWL and OWL2DSL, are presented in Section 4 and Section 5, respectively. The paper ends in Section 6 with some concluding remarks and guidelines for future work.

2 Related Work

In [2], a DSL is defined as a language designed to provide a notation tailored to one aplication domain and is based on the relevant concepts and features belonging to that domain. By providing notations tailored to the application domain, a DSL offers substancial gains in productivity and turns easier the task of the end-user. The main disadvantage of DSL is the cost of their development, requiring both domain and language development expertise.

Tairas et al. in [4], the authors explore the use of ontologies to perform domain analysis. A domain model is created by defining the scope of the domain, the terminology, concepts description and features model describing commonalities and variabilities. An ontology can be used to define the domain model. The information contained in the ontology will influence the language shape contributing for the early development stages. With this approach there is no need to start from scratch the DSL development. So, this work uses ontologies to validate the domain analysis and to get the domain terminology for the DSL creation.

Ceh et al. presented in [1] a concrete tool for ontology-based domain analysis and its incorporation on the early design phase of the DSL development. In this work, the authors identified several phases that a DSL development need. The most important are the decision, domain analysis, design, implementation and deployment.

The domain analysis is a process that uses several methodologies that differ on the level of formality and on the information extraction approach. The objective is to select and define the domain focus and collect the important information and integrate within a domain model. However, little attention is being paid to the analysis and design phase comparing with the efforts done in the implementation phase for instance. The methodologies have proven to be too complex (too much work) and they do not provide guidelines about how to use that information in the design phase. Subsequently the domain analysis is often performed informally.

The main idea of this work is to use ontologies to define the domain model. OWL classes, which define basic concepts, may be organized into a hierarchy. OWL defines two kinds of classes: simple named and predefined (the "Thing" and "Nothing"). The second component, the properties, is a binary relation, which associates values with individuals. The two main kinds of properties in OWL are object properties and datatype properties. Object properties relate objects to other objects. Datatype properties relate objects to datatype values. The third component, the individuals, are members of the user-defined ontology classes.

The authors of [1] also created a framework, called Ontology2OWL, to enable the automatic generation of a grammar from a target ontology. This framework accepts OWL files as input and parses them in order to generate and fill internal data structures. Then following transformation patterns, execution rules are applied over those data structures. The result is a grammar, acquired automatically, that is inspected for a DSL engineer in order to verify and find any irregularities. The engineer can either correct the constructed language grammar or change the transformation pattern or the source ontology. If the change was done on the ontology or transformation pattern, then a new transformation run is required. The framework can then use the old transformation pattern on the new ontology, the new transformation pattern on the old ontology, or the new transformation pattern on the new ontology. The DSL engineer can edit the source ontology with the use of a preexisting tool, such as Protégé. The final grammar can later be used for the development of DSL tools that are developed with the use of language development tools (e.g., LISA, VisualLISA).

This framework has the same objective as the one presented in this paper but some features were improved: reduction of the generation process steps, reduction of the user dependency, validation of the resulting grammar, transformation of the developed DSL to a form that is compatible with compiler generators, and generation of semantic actions associated with the grammar.

3 Onto2Gra: General Overview and Architecture

The purpose of Onto2Gra project here reported is to create automatically a Domain Specific Language based on an ontological description of that domain.

Figure 1 – the block diagram that depicts the architecture of Onto2Gra system – represents how, given an abstract ontology describing a knowledge domain in terms of its concepts and the relations among them, it is possible to derive automatically a grammar to define a new DSL for that same domain.

In a first stage the objective was to create a tool that allows to upload into the Protégé System an ontology in an acceptable OWL format. The original ontology shall be described in a special purpose DSL, a kind of natural language specifically tailored for that purpose, called OntoDL – Ontology light-weight Description Language. This tool was named Onto2OWL and what it does is to take an OntoDL file, that is an ontology specification file, and to convert it into OWL, that is the standard format for ontology descriptions. The aim of this tool is to offer an easy way to build a knowledge base to support the next phase. However, it is important to notice that this phase is not mandatory – this step can be skipped if the source ontology is already available in OWL format (or even in RDF/XML format). In that case, the user of Onto2Gra system can go directly to the second stage.

The second block is the most important on this proposal, and also the core of Onto2Gra. It is composed of a tool, OWL2DSL that makes the conversion of an OWL file into a grammar. This grammar is created systematically from a set of rules that will be explained in Section 5. From the OWL ontology description, OWL2DSL is able to infer: the non-terminal and

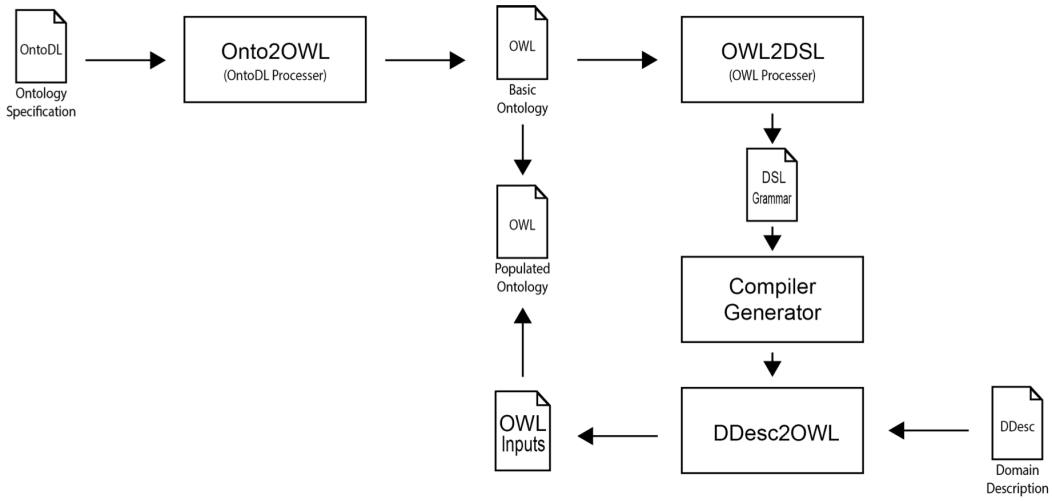


Figure 1 Onto2Gra Architecture.

terminal symbols; the grammar production rules; the symbol attributes and their evaluation rules[2]. Besides that, OWL2DSL generates a set of Java classes that are necessary to create an Internal Representation of the concrete ontology[3] extracted from each sentence of the target language[4]

The Grammar generated by OWL2DSL is written in such a format that can be compiled by a Compiler Generator (specifically in our case we are using the AnTLR compiler generator) in order to immediately create a processor for the sentences of the new DSL. AnTLR builds a Java program to process the target language; we call that processor DDesc2OWL and it is precisely the engine in the center of the third block.

Finally in the last block of the Onto2Gra architecture, the above referred tool DDesc2OWL, will read an input file, with a concrete description of the Domain specified by the initial ontology, and will generate an OWL file that, when merged into the original OWL file, will originate a specification that populates the original ontology creating a network of knowledge.

4 Onto2OWL Module

Onto2OWL is the first module of Onto2Gram system. The objective is to offer the possibility creating a specification file in the standard notation for ontologies specification that is OWL/XML from a file with a different and simple ontology specification language.

This module is compose of two parts. The first is a parser for the input files written in a DSL, called OntoDL, we have specially designed to describe ontologies. The second part is a transalator (a Java class) that manipulates the information gathered by the parser in order to generate the OWL file. In the next subsections it will be explained how these two parts work together.

[2] In the future we will also be able to derive the contextual conditions.
[3] An ontology with instances.
[4] The new Domain Specific Language defined by the generated Grammar.

4.1 The Parser for OntoDL Files

This parser is generated from a grammar that was created to specify OntoDL language. It recognizes all the basic components of an ontology described in OntoDL language.

After analyzing the problem, we found that only four parts are essential for a basic description of an ontology: Concepts, Hierarchies, Relations and Links. This supported the definition of OntoDL language as schematized below.

Listing 1 Template for a OntoDL File.

```
Ontology{
        Concepts[List of concepts]
        (,Hierarchies[List of hierarchies])?
        (,Relations[List of relations])?
        (,Links[List of links])?
}
```

An ontology is a specification of a certain domain. In order to describe the domain objects the ontology uses Concepts, or Classes. A Concept has a name, a description and a list of attributes. An attribute has a name and a type that can be a 'string', 'int', 'boolean' or 'float'.

After the Concepts specification, it is possible to define the hierarchy between two Concepts, the first is the father Concept(the super-class) and second is the son Concept(the sub-class). If one of the Concepts is not previously specified, the program ignores the Hierarchy that is being specified and issues a warning message.

After defining the hierarchical relations holding among Concepts, it is necessary to define the non-hierarchical Relations that will be used to connect Concepts. A Relation is a bridge between Concepts and it brings semantic value to the domain graph. With that in mind it was added a production rule to describe a Relation.

At last, we need to define the Links to identify the Concepts and the Relation that connect them. A Link is composed of two Concepts and one Relation. If one of the three items is not previously specified, the Link is ignored and a warning is displayed on the console. Listing 2 is an example of an OntoDL file to describe a small ontology called BookIndex (we will use this as a running example along the paper).

Listing 2 OntoDL File for the BookIndex example.

```
Ontology{
        Concepts[{Book},{Page},{Title},{SpecialTerm}]
        Hierarchies[]
        Relations[{has},{contains}]
        Links[{Book has Page},{Book has Title}, {Page contains
            SpecialTerm}]
}
```

From OntoDL grammar it is possible to generate a parser to process OntoDL sentences (ontology descriptions). This parser will store the information extracted from the input file in a set of java classes that was designed to accommodate the needs of the translator, as explained in the next subsection.

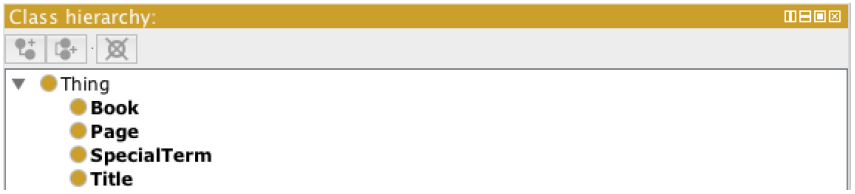


Figure 2 The generated OWL BookIndex ontology opened in Protégé.

4.2 The OWL File Generator

The OWL file generation has the objective of taking the information that was retrieved by the parser and stored in the java classes in order to generate the final product that is an OWL/XML file.

The parser returns an object of the class "Ontologies". Listing 3 shows how this class is organized.

Listing 3 Ontologies Class.

```
public class Ontologies{
    public ArrayList<Concepts> concept;
    public ArrayList<Hierarchies> hierarchy;
    public ArrayList<Relations> relation;
    public ArrayList<Triples> triple;
    public void gerarowl(String input){...}
}
```

As the listing above shows, the object that is returned by the parser already has all the information required to generate the OWL file. This information is stored in four important Array List. The first is the array of Concepts. This array stores all the Concepts that are processed by the parser. The second list stores the Hierarchies between concepts. This object is very simple, it only saves the name of the super-class and of the sub-class. The next ArrayList is also important because it stores all the non-hierarchical connections between Concepts. If a relation is not present in this list its name can not be used to create Links. As was referred above, the final array is the one that stores the Links or Triples. The links state what Concepts are connected and with what Relations.

To generate the final OWL/XML file it is enough to run the method with the name "gerarowl(String name)" that belongs to the Ontologies class. This method receives a string parameter. This parameter specifies the name of the OWL file; normally this name is the same as the OntoDL file, sent as input to the parser.

After the generation of the OWL/XML file we can work more on the ontology, to explore or to edit it. For that purpose we can load it into any tool that supports OWL/XML, like Protégé[5]. Then it is also possible to add information to enrich the ontology. Figure 2 represents the OWL version of the BookIndex ontology produced from the OntoDL description by the Onto2OWL module.

Concluding, this module was created with the objective of generating OWL files from simple descriptions of the domain. It can be used separately and independent of the other components, but it was important to generate input files for the second module of this project, OWL2DSL.

[5] `http://protege.stanford.edu/`

5 OWL2DSL Module

This section introduces the OWL2DSL module, that is presented in Figure 1 , and explains how it is possible to generate a grammar specification and generate a parser for a DSL to describe elements for the domain.

This module combines two phases the OWL or RDF parser and the CodeGenerator. This first phase, is a simple OWL or RDF parser that retrieves a Ontology Object (OO).This OO is crucial because it will enable the creation of the desired grammar.The CodeGenerator receives as input this OO and produces: the grammar and a set of Java classes necessary to implement the next module DDesc2OWL. The output grammar is composed of simple productions that obey a set of transformation rules. Those rules, look for OWL patterns to transform them systematically into grammar elements (symbols, and productions).

To start the generation process, we create a production with the axiom, `Thing` and we insert as many alternatives as the number of Concepts that are hierarchically connected to `Thing`. The listing below shows the production diagram generated by the application of that first transformation rule.

Listing 4 Thing Production.

```
thing: 'Thing['(Tproduction1|Tproductioni|TproductionN)(','(
    Tproduction1|Tproductioni|TproductionN))*']' ;
```

After the creation of this main production, all the Concepts that appeared on its RHS are iterated aiming at creating all the subsequent productions for those concepts. This task is accomplished by a recursive function `sub-production`. The function `sub-production` generates almost all the complementary Java classes that add dynamic semantics to the generated Grammar. As this `sub-production` function is recursive, it explores all the sub-Concepts and returns all the productions to the CodeGenerator Class.

The productions corresponding to the concepts also are generated according to a set of systematic transformation rules. These productions are composed of four parts. The first part is an ID that will be used to identify each instance of that concept. The attributes are specified using a non-terminal symbol preceded by a keyword (attribute name). Then, for each attribute, one new production is added to the grammar in order to return its value. After the list of attributes we have the third part, a set of sub-productions. Each sub-production contains on the RHS a sub-concept that is connected to the main Concept by a *is-a* relation. The last part of the production is used to specify the non-hierarchical relationships for the non-hierarchical relationships that connect these Concepts with other concepts.

The listing below shows some of the productions of the grammar generated from the BookIndex ontology displayed in Figure 2.

Listing 5 Example of a Generated Grammar: BookIndex.

```
thing: 'Thing['(book | page | title | specialterm) (','(book | page |
     title | specialterm))* ']';
book:'Book{' bookID(','''['(book_has)(','(book_has))*']')?'}';
bookID: STRING;
book_has:'{''has' STRING'}'; //Book has Page or Title use the STRING
    to link them
```

Summing up, it was demonstrated that a complete grammar for the new DSL can be generated from the ontology that describes that domain applying a small set of transformation rules. The upcoming grammar is completely human readable and can be adapted to any

special purpose. Adding to the grammar the Java Classes, also generated systematically and explained above, it is possible to obtain a processor to cope with the sentences of the new DSL. Listing 6 is a short example of a sentence written in the language BookIndex defined by the generated grammar shown in the previous listing.

Listing 6 Example of a DDesc input file.

```
Thing{
        Book{ "Ref_002" ,[{has "Game of Thrones"},{has "Page1"}, {has
             "Page2"}]},//the reasoner will infer the types on the
            has relations
        Page{ "Page1"},
        Page{ "Page2"},
        Title{ "Game of Thrones"}
}
```

6 Conclusion

As it was presented in this paper, the ontologies are a very useful formalism to specify domain models. Moreover a direct translation to a grammar avoids manual and informal methodologies in the design phase of a new language. This work is focused on the offline properties that an ontology can offer and define a set of transformation rules to convert the ontological information into an AntLR grammar. It was possible to automatize this process in such a way that the user has no need to interfere. Besides the grammar, associated attribute evaluation is also generated in order to fill internal data structures that can be used in a further step. At the end, it will be possible to describe a specific domain using the new created DSL and to use the information extracted from each description (written in that DLS) to populate the initial ontology.

Acknowledgements. We are indebted to Marjan Mernik and his team for the preliminary discussion on this topic. This work is funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

References

1 I. Čeh, M. Črepinšek, T. Kosar, and M Mernik. Ontology driven development of domain-specific languages. *Comput. Sci. Inf. Syst.*, 8(2):317–342, 2011.

2 Tomaz Kosar, Pablo Martinez Lopez, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, April 2008.

3 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.

4 Robert Tairas, Marjan Mernik, and Jeff Gray. Using ontologies in the domain analysis of domain-specific languages. In Michel R. Chaudron, editor, *Models in Software Engineering*, pages 332–342. Springer-Verlag, Berlin, Heidelberg, 2009.

JSON on Mobile: is there an Efficient Parser?

Ricardo Queirós

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

Abstract

The two largest causes for battery consumption on mobile devices are related with the display and network operations. Since most application need to share data and communicate with remote servers, communications should be as lightweight and efficient as possible. In network communication, serialization plays a central role as the process of converting an object into a stream of bytes. One of the most popular data-interchange format is JSON (JavaScript Object Notation). This paper presents a survey on JSON parsers in mobile scenarios. The aim of the survey is to find the most efficient JSON parser in mobile communications characterised by high transfer rate of small amounts of data. In the performance benchmark we compare the time required to read and write data with several popular JSON parser implementations such as Gson, Jackson, org.json and others. The results of this survey are important for others that need to select an efficient parser for mobile communication.

1998 ACM Subject Classification D.2.2 Design Tools and Techniques, D.4.4 Communications Management

Keywords and phrases serialization formats, mobile communication

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.93

1 Introduction

Mobile devices have become a necessity for many people around the world. The ability to keep in touch with family and business partners or to share data in real time are only a few of the reasons for the increasing importance of mobile devices. The flip side of this global trend is related with battery consumption. Smartphones are evolving from the past ten years with faster CPUs, cheaper and bigger storage, and higher-quality displays. However, battery technology did not improve at the same pace. The two biggest causes for battery consumption on mobile devices are related with the display and network traffic. The display is a major mobile phone energy hog, that can be softened by reducing its brightness and timeout.

Network operations are unavoidable in today's clouds world where everything is a service. Mobile devices need to communicate to achieve usefulness whether to transmit data over the Internet or to share data with another device. Therefore, developers followed best practices to reduce the amount of network operations in order to increase the battery's life. Basically they all resume to the following four best practices [3]:

- Consider first the need to perform a network call right now. Alternatives are pulling the service at regular intervals or allowing the server to push the data down to the client.
- Consider how much data you need to retrieve. It is possible to use different types of caches (e.g., response cache introduced for *HttpUrlConnection* in Android 4/ICS) and retrieving smaller pages of data from the service will greatly reduce your application's network traffic.
- Use transparent compressions (supported by *HttpUrlConnection* class) verifying that the data retrieved from the server is gzip-compressed.


© Ricardo Queirós;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 93–100

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- Choose a better data format, which usually involves a balance between size optimization and how dynamic the format is. If you can, choose a format that allows you to extend your data definition without losing backward-compatibility. There are several solutions such as XML, YAML, JSON, Protobuf, and others.

The last recommendation touches in a very important factor data transmission over the Internet. In a communication process it is necessary to transform data into a format that is suitable for transmission over the network and that allows the recipient to consume it without any problems. This technique is called serialization. In the context of data storage and transmission, serialization is the process of writing an object to a stream of bytes. That stream can then be sent through a socket, stored to a file and/or database or simply manipulated so that this exact same memory representation can be read later. This last process is called deserialization.

In the serialization realm, XML was used as the standard language for data representation. The most notable advantage regarding the use of XML is its heterogeneous facet. However, when encoding data in XML, the result is typically larger in size than other formats due to XML's well-known verbosity which also negatively affects the reading process. To overcome this disadvantage, JSON [1] is currently becoming a popular data representation. When data is encoded in JSON, the result is typically much smaller in size than an equivalent encoding in XML.

This paper presents a survey on JSON parsers in mobile scenarios. The aim of the survey is to find the most efficient JSON parser implementation in mobile communications. The types of communication are characterized by a high transfer rate of small amounts of data. Based on a performance benchmark we compare the time required to read and write data with several popular JSON parser implementations such as Gson, Jackson, org.json and others. The criteria used for the selection of the parser implementations were based on their popularity.

With this paper we do not intend to present an in depth description of the serialization mechanism. The results of this survey are important for others that need to select an efficient parser for mobile communication.

The remainder of this paper is organized as follows: Section 2 introduces several serialization formats. Then, we focus on the comparison of several JSON parser implementations to evaluate efficiency. Finally, we conclude with a summary of the main contributions of this work.

2 Serialization Formats

Serialization consists in the conversion of an object into a representation that can be transmitted. An application that is aware of the serialization format used can then recreate a serialized object by deserialization. The object is then restored to its original state.

In this process the serialization format plays a central role. There are two types of serialization: textual and binary. The following subsections enumerate various serialization format for both types.

2.1 Textual Serialization

One of the first standard data serialization formats was the External Data Representation (XDR) developed and published in 1987 at Sun Microsystems. XDR became an IETF standard in 1995.

Table 1 Textual serialization formats.

Name	Date	Creator	Based on	Schema/IDL	Human-Readable
CSV	1967	Yakov Shafranovich	n/a	partial	yes
XML	1998	W3C	SGML	yes	yes
XML-RPC	1998	Dave Winer	XML/SOAP	no	yes
JSON	2001	Douglas Crockford	JavaScript	partial	yes
YAML	2001	Clark Evans	C/Perl/XML	partial	yes
Candle	2005	Henry Luo	XML/JSON	yes	yes
OpenDDL	2013	Eric Lengyel	C/PHP	no	yes

In 1998, XML was introduced for asynchronous transfer of structured data between client and server in Ajax Web applications. In this context XML was defined as a human readable text-based encoding that can be used to persist objects and transmit them to other systems regardless of the platform or programming language used. Despite the format verbosity, the human readability and language independent features were very appreciated. In order to overcome the compactness issue, Binary XML has been proposed as an alternative to the regular XML.

To overcome XML's disadvantages, JavaScript Object Notation (JSON) is currently becoming a popular data representation. When data is encoded in JSON, the result is typically smaller in size than an equivalent encoding in XML. JSON is defined as a low-overhead alternative to XML and is commonly used for client-server communication in Web applications. JSON is based on JavaScript syntax, but is supported in other programming languages as well. There also exists binary encoding for JSON (e.g. BSON, Smile, UBJSON).

Another human-readable serialization format is YAML (a super-set of JSON). The main features of this format includes tagging data types, support for non-hierarchical data structures, data structures with indentation, and multiple forms of scalar data quoting.

Table 1 presents a comparison of textual data serialization formats [6].

2.2 Binary Serialization

In addition to textual formats, several binary data interchange formats have been proposed over the last decade in order to address the verbosity and the efficiency limitations of widely-accepted text-based formats such as XML and JSON [5, 4]. Among these formats we highlight the Apache Thrift, Apache Avro and the Google Protocol Buffers. Each of these protocols uses a custom Interface Description Language (IDL) to specify the structure of the exchanged data.

Google Protocol Buffer, or protobuf, is an extensible way (regardless of platform/language) for serializing structured data for use in communication protocols, data storage, among others. Protobuf is used at Google to encode structured data in binary format for implementing smaller and faster serialization. The implementation of a strategy using the protobuf format follows the sequence:

1. Definition of the schema file for the structured data;
2. Compilation of the file for generation of access classes;
3. Use of the programming language API for reading and writing messages.

After the schema definition is stored in a file (.proto), we use the protobuf compiler to generate the data access classes. These classes provide accessors for each field, methods to

serialize and deserialize data and special builder classes to encapsulate internal data structure. Listing 1 presents an example of a protobuf schema that defines the shopping item entity.

Listing 1 The protobuf schema.

```
package com.example.protobuf.model;
option optimize_for = LITE_RUNTIME;
option java_package = "com.example.protobuf.model";
option java_outer_classname = "Shopping";
message Item {
  required string name = 1;
  required string category = 2;
  optional int32 quantity = 3 [default = 1];
  enum status {
    BOUGHT = 1;
    CANCEL = 2;
  }
  message Provider {
    required string name = 1;
    required float price = 2;
  }
  repeated Provider providers = 4;
}
```

Listing 2 shows how to build a new protobuf object to an item. You start by creating a new *Builder* for the specific object you want to build and then sets up the desired values, and finally, we use the *Builder.build()* method to create an immutable protobuf object (object item). The *Item object* is then serialized to an *OutputStream*.

Listing 2 The protobuf serialization.

```
public void writeToStream(
String name, String cat, int qt, Shopping.Item.Status status,
List<Shopping.Item.Provider> providers,
OutputStream os) throws IOException {
  Shopping.Item.Builder builder = Shopping.Item.newBuilder();
  builder.setName(name);
  builder.setCategory(cat);
  builder.setQuantity(qt);
  builder.setStatus(status);
  if (providers != null)
    builder.addAllProviders(providers);
  Shopping.Item item = builder.build();
  item.writeTo(os);
}
```

Listing 3 shows how to deserialize a protobuf object from an *InputStream*.

Protobuf has a lite version for Java suitable for Android. Protobuf has more limited language reach compared to JSON or XML. Officially, Google only provides compilers for C++, Java and Python.

Thrift is a binary communication protocol. Although developed at Facebook, it is now an open source project of the Apache Software Foundation. The currently supported programming languages are C++, Java, Python, PHP, Ruby, Erlang, Perl, Go, Haskell, C#,

Listing 3 The protobuf deserialization.

```
public Shopping.Item readFromStream(InputStream is) {
 Shopping.Item item;
 item = Shopping.Item.newBuilder().mergeFrom(is).build();
 Log.d("ProtobufDemo", "Read item name: " + item.getName());
 return item;
}
```

Table 2 Binary serialization formats.

Name	Date	Creator	Based on	Schema/IDL	Human-Readable
Avro	2009	ASF	n/a	yes	no
BSON	2003	MongoDB	JSON	no	no
Cap'n Proto	2013	Kenton Varda	protobuf	yes	no
Protocol Buffers	2008	Google	n/a	yes	no
Thrift	2007	Facebook/Apache	yes	yes	no

Cocoa, JavaScript, Node.js, Smalltalk, and OCaml. Similarly to the protobuf format, we need to prepare a schema definition as input for the code generation tool generates source code for a specified programming language. A typical thrift schema representing a phone object is presented in Listing 4.

Listing 4 The thrift schema.

```
enum PhoneType {
 HOME,
 OTHER
}
struct Phone {
 1: i32 id,
 2: string number,
 3: PhoneType type
}
```

Apache Avro is a serialization framework. It uses JSON for defining data types and protocols, and serializes data in a compact binary format. Its primary use is in Apache Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services. It is similar to Thrift, but does not require running a code-generation program when a schema changes. The currently supported programming languages are Java, Scala, C, C++, C#, Python and Ruby.

Table 2 presents a comparison of binary serialization formats [6].

Choosing textual or binary data formats often depends on the context in which thay are used. Text-based formats (XML, JSON) are parsed character by character, thus imposing a limit on deserialization speed. On the other hand, binary formats make use of positional binding which allows storing the"name part of the name-value pairs in a separate file (e.g., '.proto' for ProtoBuf). These files do not need to be sent over the Web, which decrease the size of the data to be communicated. However, since these files have to be compiled before being included in a program, there are restrictions based on what languages each protocol supports.

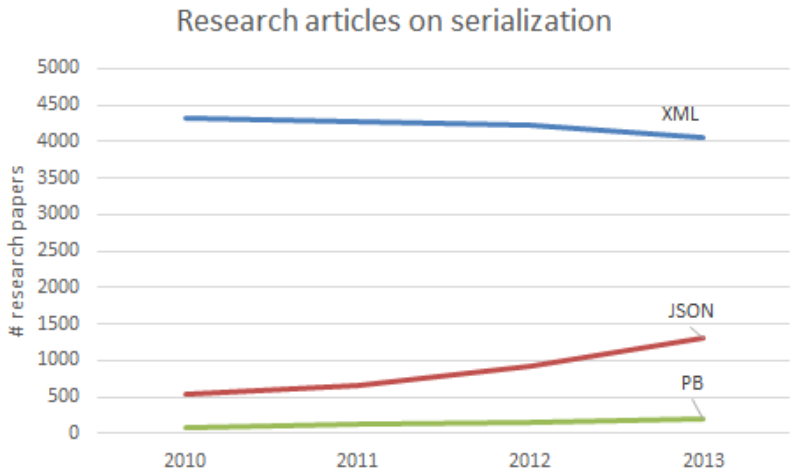


Figure 1 Research articles about serialization on Google Scholar.

2.3 Selection of a Serialization Format

In this subsection, we present the criteria used for the selection of the serialization format that will be used in the benchmark. Several criteria could be used to select a serialization format: the most popular, the most used among existent Web services, the one used in most popular applications, and others. In this case we decide that the selection will be based on the research papers found in the freely accessible Web search engine Google Scholar. This search engine indexes the full text of scholarly literature.

Figure 1 shows a comparison of the three most cited serialization formats on the Google Scholar website.

Based on the values presented in Figure 1 and, despite the XML format being the most cited in research articles, is the JSON format that has the highest growth in recent years. For this reason we decided to use JSON for the benchmark tests in the next section.

3 Comparison and Benchmark of JSON Libraries

In this section we compare the performance of several JSON parser implementations. The purpose of this benchmark is only to ensure a reasonable reading and writing performance compared to other parsers. It is obvious that the performance depends on several factors such as the used operating system, the programming language and network signal. All this just to say that the benchmark results may be misleading – if you want to infer results for a concrete case it is better to produce your own tests, with your custom data on your own hardware.

3.1 Setup and Methodology

To examine the performance of serializing and deserializing structured data, an experiment was designed using the following hardware and software:

- Hardware: ASUS Padfone with 1.5 GHz dual-core Qualcomm Krait and 1 GB memory
- Operating System: Android version 4.1.1
- Java: version 1.6.0

The test object used for this experiment is a JSON object obtained from a weather service called OpenWeatherMap. This service is often used in order to present a description of the weather of a given city. A request to the REST service returns OpenWeatherMap meteorological data of a certain city (set in the request) in JSON format. For instance, this is a typical URL request: `http://api.openweathermap.org/data/2.5/weather?q=porto`. The service returns the output in JSON format presented in listing 5.

Listing 5 OpenWeatherMap meteorological data.

```
{"id":88319,"dt":1345284000,"name":"Porto",
    "coord":{"lat":41.15,"lon":-8.61},
    "main":{"temp":306.15,"pressure":1013,"humidity":44,
    "temp_min":306,"temp_max":306},
    "wind":{"speed":1,"deg":-7},
    "weather":[
                {"id":520,"main":"Rain",
                   "description":"light intensity shower rain",
                   "icon":"09d"},
                {"id":500,"main":"Rain",
                   "description":"light rain","icon":"10d"},
                {"id":701,"main":"Mist",
                   "description":"mist","icon":"50d"}
            ],
    "clouds":{"all":90},
    "rain":{"3h":3}}
```

The JSON libraries are selected based on their popularity. Tested libraries and their versions are the following: Gson (2.2.4), Jackson (2.2.1), Minimal-json (0.9.1) and org.json (n/a).

The experiment was designed as follows:

1. 100 iterations were executed for warming-up, and then 100 iterations were executed for measuring.
2. The execution time was measured using *System.currentTimeMillis()*.
3. Finally, the average execution time is taken for each operation and library.

3.2 Performance Benchmark

While mobile devices are becoming more powerful, they still lack the processing speed of desktop PCs. Despite this fact, it is essential that the chosen data serialization format allows fast serialization and deserialization of an object. For the performance comparison of the JSON libraries previously enumerated, we compared the time required to read and write a typical weather message with the parser implementations. The results are presented in Figure 2.

Our conclusion is that when you need to serialize/deserialize Java POJOs without sacrificing performance you should choose Jackson [2]. Although minimal-json cannot outperform Jackson's writing performance, it offers a very good reading and writing performance.

4 Conclusions

This paper presented a comparison on the use of a set of JSON libraries within a mobile application. When comparing serialization libraries on a mobile platform, it is necessary to

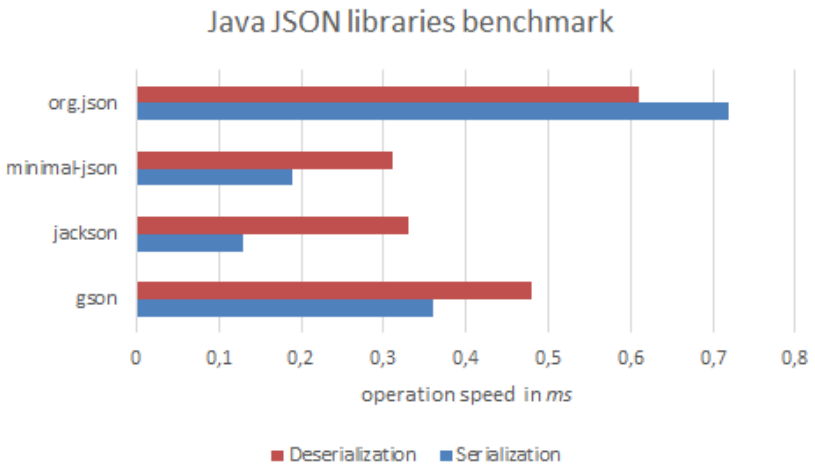


Figure 2 Java JSON libraries benchmark.

consider the most important aspects for this environment, such as data size and serialization speed. In this paper we focus on the performance facet.

The main contribution of this paper is two-fold: a survey on serialization formats organized by types: textual and binary; a performance benchmark that could be important for others that need to select an efficient parser for mobile communication.

Based on the benchmark results one can conclude that Jackson showed the best combined results. However, if your mobile app will only deserialize data, minimal-json offers the best performance in the experiment.

References

1 T. Bray. RFC 7159 – the javascript object notation (json) data interchange format. `http://tools.ietf.org/html/rfc7159`, 2014. [Online; accessed 06-May-2014].

2 Codehaus. High-performance json processor. `http://jackson.codehaus.org/`, 2013. [Online; accessed 06-May-2014].

3 Erik Hellman. *Android Programming – Pushing the limits*. Wiley, 2013.

4 K. Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *Digital Information and Communication Technology and its Applications (DICTAP), 2012 Second International Conference on*, pages 177–182, May 2012.

5 Audie Sumaray and S. Kami Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC'12, pages 48:1–48:6, New York, NY, USA, 2012. ACM.

6 Wikipedia. Comparison of data serialization formats. `http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats`, 2014. [Online; accessed 15-April-2014].

Unfuzzying Fuzzy Parsing

Pedro Carvalho, Nuno Oliveira, and Pedro Rangel Henriques

Departamento de Informática, Universidade do Minho, Braga, Portugal
{pedrocarvalho,nunooliveira,prh}@di.uminho.pt

Abstract

Traditional parsing has always been a focus of discussion among the computer science community. Numerous techniques and algorithms have been proposed along these years, but they require that input texts are correct according to a specific grammar. However, in some cases it's necessary to cope with incorrect or unpredicted inputs that raise ambiguities, making traditional parsing unsuitable. These situations led to the emergence of robust parsing theories, where fuzzy parsing gains relevance. Robust parsing comes with a price by losing precision and decaying performance, as multiple parses of the input may be necessary while looking for an optimal one.

In this short paper we briefly describe the main robust parsing techniques and end up proposing a different solution to deal with fuzziness of input texts. It is based on automata where states represent contexts and edges represent potential matches (of constructs of interest) inside those contexts. It is expected that such an approach reduces recognition time and ambiguity as contexts reduce the search space by defining a smaller domain for constructs of interest. Such benefits may be a great addition to the robust parsing area with application on program comprehension, among other research fields.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases robust parsing, fuzzy parsing, automata

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.101

1 Introduction

Recognizing sentences of a language has always been an interesting topic in computer science. This need can be easily transposed to our common life, as in our daily routine we are constantly in need to interpret all kind of inputs like images, texts or sounds. A good example of *natural parsing* is our ability to listen to human's speech and collect useful information from those sounds (sometimes imperceptible because of surrounding noise). Computer Science builds on this, as computers must understand each other or (primarily) the commands given by humans under the shape of programming language sentences.

To transpose this abstract notion of parsing to a formal domain, like computer science, rules and methodologies had to be engineered. Concerning classical parsing theory [1], the recognition process is done in accordance to rules of a formal grammar. This process is divided into three different types of analysis: Lexical, Syntactic and Semantic. Lexical analysis consists in reading and grouping the characters of an input text into tokens. In syntactic analysis a tree-like intermediate representation (the parsing tree) is built, based on the tokens and the rules of the underlying grammar. Finally, in Semantic analysis, the actual values of symbols are computed to decorate that tree and a check for consistency between the source and the language semantics definition is carried out.

In classical parsing theory sentences either belong or not to the language; and when belonging they must be derived from the grammar, following a structured set of rules without uncertainty. This property is related with the basic membership concept found on set


© Pedro Carvalho, Nuno Oliveira, and Pedro Rangel Henriques;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 101–108

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

theory [11]. However, in the recognition of unpredictable input (incomplete sentences or handwritten expressions [9, 13, 8]), it is impossible to create a grammar that covers all possible cases. If in some cases classical parsing falls short to solve the problem at hand, in other cases it goes beyond the needs. For instance, generating high level models from source code (in the context of program comprehension) does not require its complete recognition. By limiting the recognition process only to the essential portions, smaller parsing trees will be generated and unnecessary work will be spared. These scenarios where classical parsing proved to be inadequate, led to the emergence of robust parsing theories. In this context, a sentence belongs to a language with some degree of uncertainty, differently from the *do or die* situation seen before.

To cope with the robust parsing idea, two generic approaches have stood out: Island and Fuzzy grammars. But still in these approaches, parsing tends to be time consuming as multiple attempts may be required while looking for an optimal one. Performance decays and recognition precision may be partially or completely lost, which is undesirable when the objective is to extract reliable information from the input sentences.

In this context, this short paper overviews an approach for fuzzy parsing that is targeted for reliably extracting information from partially correct or incomplete input texts. We claim that the recognition in our approach is precise while tolerant to the input's fuzziness. The underlying grammar is partial w.r.t. the original one. Therefore, our approach defines a minimum degree m of certainty (in the interval [0,1]), with which each recognized sentence belongs to the original language. The approach is based on deterministic finite automata where states define precise contexts within the sentences and edges represent potential matches of constructs of interest[1] inside each context. By using this notion of contexts, the search space becomes smaller, reducing both the time to recognize the input and the ambiguity conflicts.

Since this is a starting project, this paper only scratches the surface of the approach.

Outline. The reminder of the paper provides a running example in Section 2 to make the discussion of the paper clear. Related work on the two mentioned approaches, island and fuzzy grammars, is presented in Section 3. Then, in Section 4 it is introduced our approach for fuzzy parsing, highlighting its main aspects and features. Finally, in Section 5 we conclude the paper, with a discussion of our approach in comparison with others, and also, presenting the steps for future work.

2 Running Example

Program comprehension [17, 18] is a research field aiming at studying and providing means to allow software engineers to understand legacy code, which is intended to be evolved. Several techniques have been proposed [16, 6, 5, 4, 10] for the elaboration of high-level models that abstract aspects of the programs, known to be relevant for their comprehension. One such model is the class diagram of a system, where relations between the several entities implied in the system are shown along with their internal attributes and methods. Usually, to obtain such diagrams it is necessary to process the whole system and to construct the complete parsing tree, from where only a tiny part of it is of interest.

In this example, the intention is to extract relevant information from Java programs in order to create simplified class diagrams that only show relations of classes and ignores

[1] A construct of interest is any portion of the original language that is desired to be recognized.

methods and attributes. Clearly, only the definition of the headers of classes are relevant to analyse. Everything else is to be ignored. To add complexity, amounts the fact that there are several ways of implementing classes as can be seen a few in Listing 1.

Listing 1 Examples of class headers in the Java language.

```
public class A { . . . }
public class E extends Exception { . . . }
public interface I { . . . }
public class B extends A implements I { . . . }
```

3 Related Work

In order to fix notations for a better understanding of this section, a brief recall of the formal definition of Context-Free Grammar [7] is given. Let G be such a context free grammar. Formally it is defined by the 4-tuple $G = (V, \Sigma, R, S)$ where:

- V is a set of nonterminals, which are syntactic variables denoting sets of sentences that can be derived by successive applications of the production rules.
- Σ is the alphabet of the language, and each symbol σ in Σ is called a terminal (or a token). Terminals are the basic immutable symbols of the grammar (i.e., they are never rewritten by production rules), and together form the sentences of the language.
- R is a set of production rules that specify the concrete way in which terminals and nonterminals may be combined to form sentences. A production rule p is formally given as a function of the form $V \to (V \cup \Sigma)^*$, meaning that the left-hand side (LHS) symbol (a nonterminal) derives into a sub-language given by the symbols in the right hand side (RHS). The latter may be the empty string or any combination of symbols from V or Σ.
- $S \in V$ is the start symbol (or axiom) of G.

Informally, a context-free grammar, is a set of production rules that syntactically derive sentences of a formal language. These rules describe how to form sentences from the language's alphabet that are valid according to the language's syntax. A grammar is considered *context-free* when its production rules can be applied regardless of the context of a nonterminal. A context-free grammar G defines a language, given by $\mathcal{L}(G)$, corresponding to all the sentences accepted by G.

3.1 Island Grammars

Island Grammars have been described in [15, 14]. They consist in grammars that are conceptually divided in two core sections: (i) productions where we specify constructs of interest – referred to as Islands – and (ii) productions designed to match the rest of the input – referred to as Water.

From a formal point of view, given a context-free grammar $G = (V, \Sigma, R, S)$, such and a set of constructs of interest $I \subset \Sigma^*$ such that $\forall_{i \in I}\ \exists_{s_1, s_2 \in \Sigma^*}\ .\ s_1\ i\ s_2 \in \mathcal{L}(G)$. An *island grammar* $G_I = (V_I, \Sigma_I, R_I, S_I)$ for $\mathcal{L}(G)$ defines a new language $\mathcal{L}(G_I)$, and has the following properties:

1. $\mathcal{L}(G) \subset \mathcal{L}(G_I)$, this is, G_I generates an extension of $\mathcal{L}(G)$;
2. $\forall_{i \in I}\ \exists_{v \in V_I}\ .\ v \to i \wedge \exists_{s_3, s_4 \in \Sigma^*}\ .\ s_3\ i\ s_4 \notin \mathcal{L}(G) \wedge s_3\ i\ s_4 \in \mathcal{L}(G_I)$, this is, G_I can recognize constructs of interest from I in at least one sentence that is not recognized by G.

Let's consider the running example introduced in Section 2. Listing 2 shows the base specification for an Island Grammar that recognizes the headers of classes in Java. First it is defined the start symbol input that derives in a (possibly empty) sequence of chunks. Then, all chunks derive in either water or island. In this particular case islands will match the constructs of interest for analysing class headers.

Listing 2 Island grammar intended to recognize classes in Java.

```
input   : chunk*

chunk   : island | water

island : 'class' ID extend? impls?
       | 'interface' ID
extend : 'extends' ID
impls  : 'implements' ID (',' ID)*
       ...

water   : .*

ID      : [A-Z][A-Z0-9]*
```

Although simple at first glance, this grammar is able to recognize all class headers in Java source code. More complex problems may imply the implementation of new islands, producing an almost tailored made parser.

3.2 Fuzzy Grammar

A fuzzy parser [12] is able to recognize only parts of a language according to some kind of specification, or set of rules, that are established by the programmer. It ignores the input that is being consumed until it reaches a mark that symbolizes the start of a substring that is meant to be recognized. These special marks are commonly referred to as anchors.

From a more formal stand, considering a context-free grammar $G = (V, \Sigma, R, S)$, a fuzzy parser $F(G)$ is defined as $F(G) = (V', \Sigma, R', A, S)$, where V' is a set of anchor nonterminals, Σ remains the alphabet of the language, R' is the provided set of rules of type $V' \to A \times (\Sigma \cup V')*$, $A \subseteq \Sigma$ is the set of anchor symbols and S is the start symbol.

An anchor flags, thus, the beginning of a substring that is meant to be recognized by $F(G)$; this means that each $s \in \mathcal{L}(G)$ contains at least one anchor that is accepted by $F(G)$. For each anchor $a \in A$ there is a rule $r_a \in R'$ that specifies the substring of s that is meant to be recognized. In conclusion, the language $\mathcal{L}(F(G))$ that is partially accepted by $F(G)$ can be described as follows: $\mathcal{L}(F(G)) = \{s \in \mathcal{L}(G) | s = \omega_1 a \omega_2 \wedge \omega_1 \in \Sigma^* \wedge \omega_2 \in \Sigma^* \wedge a \in A\}$, where $S \to^* s$ (s derives from S by multiple applications of rules in R).

Considering again the running example, it can be easily re-engineered to conform to this notion of Fuzzy grammars. The obtained grammar is showed in 3.

The main difference to the island grammars approach is the implicit existence of a water section that consumes characters not matching anchors and associated rules.

4 Our Approach

Our approach springs from fuzzy grammars and it is intended to *precisely* recognize constructs of interest of some language, while dealing with the fuzziness (incompleteness, incorrectness, etc.) of the inputs. We do this by adding the notion of contexts that *unfuzzy* the parsing

Listing 3 Fuzzy grammar able to recognize classes in Java.

```
 input   : anchor*

 anchor  : 'class' ID ext? impls?
         | 'interface' ID
 ext     : 'extends' ID
 impls   : 'implements' ID (',' ID)*
         ...

ID       : [A-Z][A-Z0-9]*
```

and reduce the search space for constructs of interest. Since this is a fresh idea, by the time of writing of this paper, no formal definition was defined for this approach. Therefore, the following presentation is rather informal, and kept at a high level of abstraction following the running example of Section 2.

The approach is based on deterministic finite automata notions. The rationale is to recognize constructs of interest defined within specific contexts of a language. Anchors and associated rules define the syntax of such constructs of interest and their recognition may or may not define transitions between contexts. To be precise, in a context, any sequence of characters is consumed and ignored unless a rule for such a sequence is defined. The normal behavior after matching a rule is to remain in the same context (defining a loop transition in the automaton); the exception is to transit to a different context.

Contexts are (final) states in an automaton with particular notion of hierarchy. This means that a context is inside other contexts, facilitating transitions from a child context to its parent, when explicit transitions are not suitable. In a sense, the overall behaviour of the context space can be regarded as a non-linear stack machine, where pushing contexts is as usual, but popping contexts may depend on the existence of a transition (defined in the respective automaton) to a deeper context in the stack. Bare in mind that this is different to backtracking in parsing strategies: it is intended behaviour jumping from contexts to contexts.

Because our approach requires the generation of a deterministic automaton, ambiguity will not be an issue (as in many fuzzy parsing approaches). Consequently, a unique parsing tree is produced that only contemplates the recognized constructs of interest (it is not a full parsing tree). Considering these claims, performance gains seem to be evident.

To make things clear, let's build on the running example by adding a new level of complexity: recognizing methods and their arguments inside each class. In the following, notation @X and $>> X$, is considered meaning respectively, the definition of a new context X and the definition of a transition to the concrete context X. In the latter, X may be substituted by $\wedge$ to express a transition to the parent context. Optionally, $>> X[A]$ may be used to specify that after transiting to context X, the input text must be read from before the previously consumed anchor A, and not from the current position.

The grammar begins with a default context (*@default*). In this context, when an anchor `class` is recognized (along with the associated construct of interest) a transition is made to the *@cls_hdr* context:

```
@default
 class   : 'class' ID >> cls_hdr
 iface   : 'interface' ID
```

Inside the *@cls_hdr* context, two constructs of interest are defined that detail the relationships of the class (*Extends* and *Implements*) with other classes. This means that the search space in this context is limited to these two constructs. However, since the body of the class is also required to be analysed, an extra construct '{' shall be recognized that defines a transition to the new context *@cls_bdy*:

```
@cls_hdr
ext      : 'extends' ID
impl     : 'implements' ID (',' ID)*
cls_in   : '{' >> cls_bdy
```

To unburden notation and recognition complexity, the `(',' ID)*` construction on impl rule could be defined within a new context as follows:

```
@cls_hdr
...
impl     : 'implements' >> ids
...

@ids
id       : ID
```

Besides alleviating the notation, this version modularizes the grammar as *@ids* context may be reused from another context or construct of interest. Moreover, details like `','` symbols are automatically ignored and all the `ID` symbols are processed as a unit and not as a list, which may be beneficial when extracting and transforming data. However, extra rules would be needed to define when to transit to a different context, since *@ids* may not be desired to be final, implying more parsing time. Also, the notion of concrete *context* is lost (contradictory to the philosophy of the approach), being only possible to recover it by taking advantage of semantic actions (not in the scope of this paper).

When in the context of the body of a class (*@cls_bdy*), for this example, only the constructs for public methods and private instance attributes are of interest. Notice that methods also include class constructors and that, in Java, they have different syntax:

```
@cls_bdy
cons     : 'public' ID '(' >> args
mthd     : 'public' TYPE ID '(' >> args
mthd_in  : '{' >> mth_bdy
attr     : 'private' TYPE ID ';'
```

However, both constructors and methods define their arguments following the same syntax. Therefore, and following the approach exemplified above, a new context (*@args*) is defined to parse the arguments. This is an example of reusing a context from different constructs of interest:

```
@args
arg      : TYPE ID
arg_ext  : ')' >> ∧

@mth_bdy
mth_ext  : '}' 'public' >> ∧ ['public']
```

As explained above, an extra rule is added to the *@args* context in order to pop it from the stack and returning to the parent context (*@cls_bdy*, in this example). Such a *pop* occurs when `')'` anchor is matched.

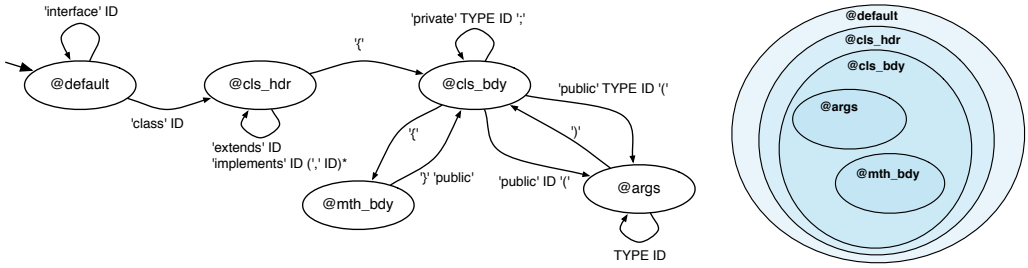


Figure 1 Automata and stack representation of the *unfuzzied* fuzzy grammar for the Java classes recognition.

Still in the *@cls_bdy*, when `'{'` occurs, the parsing transits to *@mth_bdy*, *i.e.*, the body of a method. Since for the purpose of the running example, nothing is required from the interior of a method, the grammar only pops the context when `'}'` is recognized. Everything else is ignored. The recognition of `'public'` after the `'}'` is necessary, so that other `'{'` and `'}'` (representing blocks inside the method) will be ignored. But notice that recognition of `'public'` is discarded once going to the parent context, in order to be recognized again as part of the *@cls_body* context. The careful reader may be asking what happens when there is no `'public'` anchor to leave *@mth_body* context. This occurs, for instance, on the definition of the last method of a class. In these cases the parsing will finish in the *@mth_bdy* context, as there is nothing else of interest to recognize at that point. Since every context is a final state in the automaton, then the recognition correctly accepts the input.

As expected, this last context as well as the mthd_in rule could be left out of the grammar. This is only included to show that from a context, it is possible to go to more than one different contexts.

The main difference of this approach to the ones presented in Section 3 resides in the usage of contexts while processing the input. This allows for a finer assessment of data, because the constructs of interest are limited to a smaller domain. This, in turn, allows for the reduction of uncertainty that is inherent to robust parsing, and to fuzzy parsing in particular (thus the title of this paper).

5 Conclusion

In this paper we informally presented an approach to fuzzy parsing that reduces the uncertainty of what is being recognized. The introduction of a notion of contexts (as states of an automaton) provide such disambiguation and goes beyond diminishing the search space for constructs of interest. The reduction of uncertainty is more valuable when the objective is to extract reliable information from the parsed input, by means of semantic actions. Semantic actions were not covered in this paper due to space limitations and lack of completeness by the time of writing. However it seems easy to understand the role of contexts to this end: at each context, it is known exactly what each syntactic construction means; for instance, `TYPE ID` can represent in a context a variable declaration whereas in another one it may be the declaration of a class attribute.

A key point of this approach is that only a part of a parsing tree will be generated. Although not covered by this paper, several approaches decided to recognize the entire source, sometimes even making multiple parses [2, 3, 9]. Again, the precise definition of the processing steps for our approach is left for future work. Nevertheless, the automata- and stack-based operational behaviours seem to provide basic intuition about it.

References

1 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

2 Peter R. J. Asveld. A fuzzy approach to erroneous inputs in context-free language recognition. In *Proceedings of the Fourth International Workshop on Parsing Technologies IWPT'95*, pages 14–25, Prague, Czech Republic, 1995. Institute of Formal and Applied Linguistics, Charles University.

3 Peter R. J. Asveld. Fuzzy context-free languages – Part 2: Recognition and parsing algorithms. *Theoretical computer science*, 347(1):191–213, 2005.

4 Mario Berón, Pedro R. Henriques, Maria J. Pereira, and Roberto Uzal. Program inspection to incerconnect behavioral and operational view for program comprehension. In *York Doctoral Symposium, 2007*. University of York, UK, 2007.

5 Mario Berón, Pedro R. Henriques, Maria J. V. Pereira, and Roberto Uzal. Static and dynamic strategies to understand c programs by code annotation. In *OpenCert'07, 1st Int. Workshop on Fondations and Techniques for Open Source Software Certification (co-located with ETAPS'07)*, 2007.

6 Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1):13–17, 1990.

7 N. Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, 1956.

8 John A. Fitzgerald, Franz Geiselbrechtinger, and Mohand Tahar Kechadi. Application of fuzzy logic to online recognition of handwritten symbols. In *IWFHR*, pages 395–400, 2004.

9 John A. Fitzgerald, Franz Geiselbrechtinger, and Mohand Tahar Kechadi. Structural analysis of handwritten mathematical expressions through fuzzy parsing. *ACST*, 6:151–156, 2006.

10 Yann-Gaël Guéhéneuc. A theory of program comprehension: Joining vision science and program comprehension. *International Journal of Software Science and Computational Intelligence*, 1(2):54–72, 2009.

11 Alexander S. Kechris. *Classical descriptive set theory*, volume 156. Springer-Verlag New York, 1995.

12 Rainer Koppler. A systematic approach to fuzzy parsing. *Software Practice and Experience*, 27:649, 1996.

13 Scott MacLean and George Labahn. Recognizing handwritten mathematics via fuzzy parsing. Technical report, Tech. Rep. CS-2010-13, School of Computer Science, University of Waterloo, 2010. 3, 2010.

14 Leon Moonen. Generating robust parsers using island grammars. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 13–22, 2001.

15 Leon Moonen. Lightweight impact analysis using island grammars. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 219–228. IEEE, 2002.

16 Václav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *IWPC'02: Proceedings of the 10th International Workshop on Program Comprehension*, pages 271–278, Washington, DC, USA, 2002. IEEE Computer Society.

17 Scott Tilley. 15 years of program comprehension. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 279–280, 2007.

18 A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

Part IV

Programming Languages and Compilers

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Contract-Java: Design by Contract in Java with Safe Error Handling

Miguel Oliveira e Silva[1] and Pedro G. Francisco[2]

1 **University of Aveiro, IEETA, DETI**
Campus Universitário de Santiago, Aveiro, Portugal
mos@ua.pt

2 **University of Aveiro, IEETA**
Campus Universitário de Santiago, Aveiro, Portugal
goucha@ua.pt

Abstract

Design by Contract (DbC) is a programming methodology in which the meaning of program entities, such as methods and classes, is made explicit by the use of programming predicates named assertions. A false assertion is always a manifestation of an incorrect program.

This simple founding idea, when properly applied, give programmers a tool able to specify, test, debug, document programs, as well as a mechanism to construct a simple, safe and sane error handling mechanism. Nevertheless, although well adapted to object-oriented programming (and other popular techniques such as unit testing), DbC still has a very low practical acceptance and application. We believe that one of the main reasons for such is the lack of a proper support for it in many programming languages currently in use (such as Java). A complete support for DbC requires not only the ability to specify assertions; but also the necessity to distinguish different kinds of assertions, depending of what is being asserted; a proper integration in object-oriented programming; and, finally, a coherent connection with error handling mechanisms.

It is in this last requirement that existing tools that extend Java with DbC mechanisms completely fail to properly, and coherently, integrate DbC within Java programming. The dominant practices for systematically handling failures in programming languages are not DbC based, using instead a defensive programming approach, either by using normal languages mechanisms (as in programming language `C`) or by the use of typed exceptions in `try/catch` based exception mechanisms.

In this article, we will present and justify the requirements posed on programming languages for a complete support for DbC; On the context of the last presented requirement – error handling – defensive programming will be discussed and criticized; It will be showed that, unlike Eiffel's original DbC error handling, existing typed exceptions in `try/catch` based exception mechanisms are not well adapted to algorithmic abstraction provided by methods; Finally, a new DbC Java extension named *Contract-Java* will be presented and it will be showed that it is coherently integrated both with Java existing mechanisms and DbC. It will be presented an innovative *Contract-Java* extension to DbC that automatically generates debugging information for (non-rescued) contract failures, that we believe further enhances the DbC debugging capabilities.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, D.3.4 Processors

Keywords and phrases design by contract, defensive programming, exceptions, Java, contract-Java

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.111

© Miguel Oliveira e Silva and Pedro G. Francisco;
licensed under Creative Commons License CC-BY

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 111–126

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Design-by-Contract programming, or officially [21] *Design by Contract*^{TM} (DbC)[1] is a development methodology inspired both by studies on formal programming and also, by the way contracts work in the "real world", in particular, in a clear distribution of responsibilities whenever a failure occurs in a program. It aims for a substantial improvement of a program correctness and robustness. It was born in 1986 [19, 21, 18] and first implemented within the Eiffel language (1988) [20].

The concepts in which Design by Contract is based are present in the works of Turing [27], Floyd [5], Hoare [9], Dijkstra [4], Gries [8], Jones [11, 10] and also Goguen [7].

Several approaches exist to extend Java with DbC: Jass [2], Modern Jass [26], JML[2] [13], Cofoja [12], ezContract [3], and DbC4J [1]. However, all have achieved just a portion of the features required by the methodology. All but one (Jass) fail to implement fault tolerance with a disciplined exception mechanism and automatic documentation generation; furthermore, all treat the DbC approach as an optional add-on to the language (using annotations or aspects), failing to fulfill the requirements for the full implementation of DbC. A full integration of contracts as core language syntactical constructs enhances the possibility to fulfill all the requirements including a disciplined exception mechanism (in which the knowledge of contracts is essential). Although in this approach it is not possible to directly use of native Java compilers when the new syntax is involved (as happens with other approaches) it not only makes contracts non-optional language constructs (impossible to ignore), but is also allows the direct integration and use of native Java code (allowing the direct reuse of existing Java code and libraries). Table 1 summarizes the support for DbC of all these approaches.

Regarding error handling, defensive programming [14] remains the dominant approach to systematically handle internal program's failures. This dominance has also "contaminated" some DbC approaches (e.g. JML's `exceptional_behavior`) in which the launch of exceptions can also be made part of a contract specification blurring the simple DbC view of methods (that either succeed, meeting its postcondition and the object's invariant, or fail with a contract failure). If a method terminates launching a contractualized exception, did it really fail, or is it simply meeting its contract?

The way DbC handles errors is much simpler, coherent and safe. It even gives the ability for a program to unambiguous know when it is (or it is not) failing, without the necessity for an external error arbitration, or a deep knowledge of the way programmer implement their code. Current use of exceptions disallow such possibility, because there are many types of exceptions, and some of them are, sometimes, used as normal program's flux control, to the point of such usage is promoted as a good programming practice [14].

It should be clarified that we are following a very pragmatic view of DbC in the line of original Eiffel's proposal and implementation, and not aiming a more formal assurance of contracts. Also, we are not stating that in practice contracts express the full semantics of modules (e.g. a formally complete postcondition), but simply assert *some* of those semantics.

This article is structured as follows. In Section 2 we present our contributions. In Section 3 we identify an justify the requirements posed for a complete DbC language implementation. Section 4 the problems and solutions for systematic error handling are discussed. Section 5 presents Contract-Java approach. Finally, Section 6 present some concluding remarks and future evolutions of the language.

[1] Trademarked by Eiffel Software in the United States
[2] Java Modeling Language.

2 Contributions

The major contributions of this article are the following:

- The presentation and justification of the necessary requirements posed to a programming language for a complete pragmatic support for DbC;
- A critical comparison between defensive programming and DbC approaches to error handling, and, in particular, the algorithm abstraction problems posed by typed exceptions and `try/catch` based exceptions mechanisms;
- A new complete DbC extension for Java named *Contract-Java* (able to accept and compile existing Java code);
- A complete support for a disciplined exception mechanism in *Contract-Java* without negative and undesirable interferences with native's Java exception mechanism;
- An innovative, and safe, integration of Java's native exception mechanism within DbC (allowing the application of the powerful DbC disciplined exception mechanism to any Java exception);
- A DbC enhanced debugging mechanism, by automatic generation of semantic information in the presence of an assertion failure (freeing the programmer from that burden).

3 Requirements

To achieve a complete pragmatic support for DbC within a programming language, we must clearly identify the necessary requirements to be fulfilled and justify the rationale behind them. That is the goal of this section.

3.1 Different Assertions

▶ **Requirement 1** (different assertions). *Different kinds of contracts (preconditions, postconditions, invariants and others) should be represented by different assertions. These assertions assume different roles depending on their kind, carefully assigning responsibility to different parts of the program.*

Although one can attach the (total or partial) meaning of a software element by an assertion, different responsibility chains are involved depending on where the assertion resides. A clear identification of such responsibility is required for a proper software element understanding.

In general one may identify assertions (preconditions) that must be observed before a subprogram execution (method or block), and the ones that must be ensured afterwards (postconditions). The former, are the responsibility of the client of the subprogram execution (caller or the code before the block), and the latter are the subprogram's responsibility. Hence, in the presence of an error (false assertion), depending on the type of the assertion, different program parts are to be blamed (this distinction is essential not only for debugging but also for the implementation of an appropriate error handling mechanism).

Structured programming gives a special abstraction role to methods, from which a separation of method assertions and internal algorithm assertions is desirable. Hence the terms *precondition* and *postcondition* are usually applied to methods, and other internal assertions are named *assert* (or *check* in Eiffel).

From object-oriented programming also results the necessity for a new type of assertion: *invariant*. It is needed to properly attach meaning to abstract data types implementations (based on classes and objects). In particular, it asserts the conditions that are required to be

true whenever objects are in an observable state (stable time) [21]. Invariants also clearly identifies who's responsible for it (the object).

A DbC realization that does not syntactically differentiates all these different assertions may lead to a wrongly attributed responsibility chain, compromising its proper specification and rectification.

3.2 Locality of Contracts

▶ **Requirement 2** (locality). *Contracts should be defined near to the entities they specify. The meaning (specification) of a software entity should be defined near the classes they contractualize; they are integral part of the code.*

This requirement simply states that the programmer should not be mislead to assume a different contract of a software entity than the one that was really defined. Also, no doubt should ever exist on the complete contract that applies to it.

Such possibility arises when one allows that the definition of a contract to reside elsewhere in the program text other than near to the software element if contractualizes. By definition, AOP approaches to DbC suffer from this problem.

3.3 Contracts are Part of the Interface

▶ **Requirement 3** (interface). *Contracts are part of the interface*[3] *(not implementation): contracts are expected to be readily available to anyone, with or without access to the source code of a contracted program: they are part of a program's interface with the rest of the program.*

An *Abstract Data Type* (ADT) [15, 21] defines a class of abstract objects which is completely characterized by the (public) operations available to those objects. A *class* is a (possible partial) implementation of an ADT [22]. An object-oriented program is a structured collection of ADT implementations [22]. Hence, ADTs are the most important abstraction blocks within object-oriented programming.

However, ADTs without explicit semantics (as provided by non DbC languages such as Java) suffer from the same serious problems as methods without contracts, increased by a scale factor because an ADT exports multiple methods (and not just one) and contains a (possible abstract) data representation.

Since a class is much more than the sum of its public methods, a new contract is required to express such semantics. That is the role of *invariants*, which express assertions that must be always true when the class's instances (objects) are in an observable state (named stable time [22]).

The set of contracts (preconditions and postconditions) of all the class's public methods together with the class invariant form the class contract. This kind of contract is the most important contract in object-oriented programming.

If we take a broader view of these concepts – ADTs, contracts, methods and classes – we can recognize that they all fit perfectly together. ADTs define the class interface. The class contract implements the ADT's semantics. The class is defined as a set of public methods glued by a common invariant, and method contracts implement the method semantics.

Contracts must, as such, be integral to the class interface, as is the name of the methods and its arguments. They define the ADT by means of a specification and, as such, contracts

[3] in terms of defining an ADT, not in terms of the Java's interface mechanism

are independent of the implementation. Furthermore, when we extend the class, using subtype polymorphism, the ADT must be kept consistent. The only way this is possible is if contracts belong to the class interface and not to its implementation.

3.4 Contracts are Inherited

▶ **Requirement 4** (inheritance). *Contracts are inherited: a descendant class must fulfill at least all contracts of its parent class, as well as its method's postconditions; preconditions can, but don't have to, be loosened.*

Liskov's substitution principle states that, on object-oriented programming, any property which is verified on a supertype also holds for its subtypes.

> Let ϕ(x) be a property provable about objects x of type T . Then ϕ(y) should be true for objects y of type S where S is a subtype of T [16].

In the context of DbC, this implies that class contracts must be inherited. As Meyer states [21] it is possible to redefine contracts on descendant classes as long as certain conditions are met. The precondition of the descendant class must be equal or weaker than that of the parent class and, in the case of invariants and postconditions, the descendant class must abide at least by the parent class, meaning it can further restrict its invariant and/or its output (postconditions), but never to weaken them.

Since contracts must be taken in consideration by the descendant classes in order to not change the ADT associated with the parent class, we further strengthen the need for requirement `R3`: the contracts must be part of the class interface and not implementation – otherwise, the semantic meaning of the class would be partially hidden from the outside view, stripping the added value which contracts bring on defining ADT.

3.5 Documentation

▶ **Requirement 5** (documentation). *The documentation must not only be included in the code but also validated with the code as much as possible and, thus, be at least partially extracted from the defined contracts, forming the class and method specification. In order to properly support contracts, the documentation must support inheritance. In addition, to completely document the method/class, the documentation should feature a flat view of the documented class. Full documentation support for contracts is not only desirable but a requirement to implement Design by Contract. Contracts (and thus, the documentation they provide) are validated at every program run.*

Design by Contract in its full form allows for the "single product principle": the product is the software. All specification and documentation is in, or extracted automatically from, the software [23].

In order to be useful, the documentation of a class must present the full overview of this class. This means that the documentation must be presented in flat form: it must contain not only the contracts that were defined on that class and its methods but as well all contracts inherited from the implemented interfaces and from its superclasses. The flat documentation allows for a complete documentation of the class expressed semantics in one place.

3.6 DbC Exceptions

▶ **Requirement 6** (DbC exceptions). *An error handling mechanism should be provided in order to ensure that a method can only succeed, by observing all of its attached assertions, or*

Table 1 Support for DbC in some of existing Java extensions.

DbC Java	R1-different assertions	R2-locality	R3-interface	R4-inheritance	R5-documentation	R6-DbC exceptions
Native Java	no	yes	no	no	no	no
jass	yes	yes	no	no	partial	partial
Modern Jass	yes	yes	yes	yes	no	no
JML	yes	yes	yes	yes	no	no
Cofoja	yes	yes	yes	yes	no	no
ezContract	yes	yes	no	no	no	no
DbC4J	yes	yes	yes	no	no	no

fail signaling its failure to the caller with an exception; no other outcome should be allowed. Also, a disciplined exception mechanism should be provide to support a clean termination or for fault tolerance purposes, without ever compromising the method execution semantics.

This last requirement will be discussed in the next section.

4 Systematic Approaches for Error Handling

The dominant practice for handling errors in use today, is based on *defensive programming* [14]. In this methodology, partial procedures are considered a bad idea, and should be replaced with methods that, accepting everything, protect themselves by using normal language constructs – such as conditionals, results, error variables or exceptions – to identify the error and notify the caller. Lacking an exception mechanism, programming language `C`, uses function results (given two different meanings to the result), or global variables (as `errno`). On the other hand, Java and other more recent languages, use also the exception mechanism for the same purpose. As an example, consider the following code excerpt:

```
static double sqrt(double x) {
  if (x < 0)
    throw new IllegalArgumentException();
  ...
}
```

It is assumed that all clients of `sqrt` should track `IllegalArgumentException` to ensure complete robustness. Even if a client it confident that the argument will never be negative – by simply performing the logical test *before* the call – one cannot deactivate internal method's defensive code (because it is a total procedure).

Hence, since methods are implemented as total procedures, in defensive programming no special burden is putted on a method's client *before* its execution. It is assumed that the client will take the proper measures to handle possible failures *after* the method's execution. Such practice, however, is not only seriously flawed when exceptions are not used (because it is easy to forget such post-execution verifications), but it may also be flawed when they are used. Not only it is easy in a `try/catch` instruction to ignore the exception (all that is required is an empty `catch` block), but also, in this example, it completely misses the real source of the error: *a precondition failure*. To support this argument, it is of little importance to verify that such precondition is not explicit in the code. No sane programmer

implements a real number `sqrt` method for negative arguments, only for non-negative ones. The precondition is simply implicit in the code, implemented defensively with a conditional and an `IllegalArgumentException`, but it is, nevertheless, there. Hence, the guilty part for this failure is not the `sqrt` method (as its post-execution error handling might suggests it was), but its caller. The fault *precedes* the call. Also, it should be absolutely clear that a call to `sqrt` with a negative argument should be considered a program error (and not some ambiguous execution state of the program).

DbC takes the opposite approach to this problem: methods should be specified for what they are meant to do. If such goal only makes sense for some of the possible states of its arguments (or its object's state) so be it. A partial method implementation should be the choice, expressing clearly the necessary preconditions for its use (and the postcondition for its effect and result). It is important to note, that nothing is lost in terms of detecting errors and protecting methods and classes. To that goal, in DbC, all that is required is active executable assertions. In this respect, the difference to defensive programming is that we may deactivate particular assertions if we are confident the asserted code is correct.

However, taking now a broader perspective on systematic error handling methodology, DbC gives us much more than simply a way to detect and act on errors.

First, as already stated, in DbC a program *knows* when and if it has failed: simply when a false assertion (any one) was executed. In defensive programming, no such simple criteria exists to assert the program correctness (while executing). Not even the existence of an active exception is a similar criteria because, due to defensive programming practices, exceptions are used also as a simple flux control instruction, and sometimes promoted as perfectly acceptable practices [14].

Secondly, DbC clearly and unambiguously distributes the responsibility of the fault: a precondition failure is the responsibility of the method's caller, any other false assertion is of the responsibility of the method and/or class it belongs to.

Finally, it is perfectly adapted to method's algorithmic abstraction, the meaning of a failure adapts itself automatically as we climb up the method execution stack. For example, a precondition failure is the responsibility of the method's direct caller, but if this failure propagates in the execution stack, it no longer is a precondition failure to the caller of the caller (which would not make sense, because the precondition failure was in another method).

To better understand this last point we must take a more detail view of the dominant exception mechanism in use today.

4.1 Typed Exceptions and `try/catch` Instruction

In modern programming languages, faults are handled through exception handling mechanisms. The rationale is to attach a particular exception type to each fault source, and then allow the programmer to explicitly handle them elsewhere in the program. However, methods and classes can be very powerful abstraction mechanisms. A particular type of exception might be meaningful near to where it is generated, but soon enough it loses its meaning as one travels up in the method call stack. Such problem has been identified more than 30 years ago [17], and to cope with it an "explicit propagation of exceptions" requirement was defined for an alleged ideal exception mechanism [6].

Take, as an example, a possible function to solve the quadratic equation (using the previous listed `sqrt` method).

```
/**
* ax^2 + bx + c = 0
* the 2 root returned as a 4 length array ({r1.re, r1.im, r2.re, r2.im}.
*/
static double[] quadraticEquationSolver(double a, double b, double c) {
  if (a == 0)
    throw new IllegalArgumentException();
  ... sqrt(delta) ...
}
```

For the sake of the argument, lets suppose that this method is incorrectly implemented and the programmer did not take enough care to ensure a call to `sqrt` with a non-negative argument. Obviously, a `IllegalArgumentException` will be throwed by `sqrt`. However, if automatically propagated to `quadraticEquationSolver` client, as the exception mechanism does by default, it will fool it to believe that it as passed a wrong argument ($a = 0$).

To cope with this serious problem, some authors [14] suggest that the exception should be handled in places in which its meaning is correct, and eventually a different type of exception should be launched. To put this wise advise in practice, however, not only it could be an overwhelming amount of extra work on the part of programmers (and easy to miss, creating hard to track errors), but also questions one of the main goals of the exception mechanism: to separate normal code from exceptional one. Our methods would be "contaminated" with lots of `try/catch/throw` instructions. A new defensive exception mechanism could be devised, in which the automatic propagation of exceptions in the call stack, would be always wrapped in new `MethodFailure` alike exceptions. But such a mechanism seems clumsy, inefficient, and most of all, not necessary because DbC provides a much simpler solution.

In Java, it is also suggested that the method signature should always list the exceptions it may throw (and its meaning), which would also "contaminate" our code with lots of `throws` declarations, making it a possible maintenance nightmare.

But even if we assume that such a practice was the only way to handle exceptions (which is is not), it would still break algorithmic abstraction of methods: When a method is constructed, an abstraction barrier is created between the client (caller) of the method, and its implementation. The client care only for the meaning of the method (its postcondition). The implementer job, is to use an algorithm than fulfills such meaning. However, the possible exceptions that might be launched *depends* on the algorithm chosen by the implementer. Take, for example, the case of a "days of a month" method:

```
public static int daysOfMonth(int month, int year) {
  final int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
  int result = days[month-1]; // Possible ArrayIndexOutOfBoundsException
  if (month == 2 && leapYear(year))
    result++;
  return result;
}
```

Should `ArrayIndexOutOfBoundsException` exception be part of the method's signature? It might, in this particular implementation, but different algorithms can be devised in which no such exception will ever be launched:

```
public static int daysOfMonth(int month, int year) {
  int result = 0;
  switch(month) {
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
      result = 31;
      break;
    ...
  }
  return result;
}
```

Please note that, as already mentioned, the alternative usage of other exception types, such as `IllegalArgumentException`, would still be problematic because its scope (meaning) only reached the method's direct caller (hence a `try/catch` would be required to handle the exception).

This clearly shows that attaching such exceptions to the method's signature would be an overspecification, hence: a break in the algorithm abstraction of methods. Thus, any exception type linked only to an algorithm's implementation, should never be part of the method's specification.

4.2 DbC Error Handling

In a DbC approach, running a method can only result in two possible outcomes (no compromise here): either the method succeeds, observing all its attached assertions (postcondition, possible object invariants, and other internal assertions); or it fails raising a DbC exception. Furthermore, the meaning of the exception signaled by methods is not immutable (forever binded to its original fault). As the exception propagates in the method invocation stack, its abstract meaning changes, going from the original assertion failure to the failure of each method in which it is propagated. What this means is that if the programmer desires to build a fault tolerant program, he can adapt it (automatically) to the abstraction level of the redundant method, regardless of the primeval fault origin. The rationale is as simple as it is powerful: a method need only to ensure its attached assertions, hence in a DbC fault tolerant program we only need a disciplined exception mechanism [20] containing a rescue execution block, in which either the method execution is retried (possibly selecting a different execution path within the method), or it fails (with an exception propagation). The possible alternative method execution, need only to be concerned with ensuring the method postconditions (and object invariant), not with any possible internal assertion failure that might have occurred in a previous failed execution (hence, for fault tolerance goals, the specific type of the original exception loses much of its importance, as long as the objects involved were properly cleaned-up).

In a disciplined exception mechanism it is structurally very hard to ignore an exception either by distraction, laziness, or bad coding practices: A DbC exception is only recovered iff due to the interaction of the rescue code and the methods body, the method is able to fulfill its postcondition (and invariant). To the clients of such a method, the program proceeds as if nothing wrong had happened, thus achieving a simple, safe, and powerful fault tolerance mechanism.

To stress even further the importance of requirement 1, the rescue code should only be applied to faults of the responsibility of the method. Hence, if this method's precondition has failed, its eventual rescue clause should not be executed (it is impossible to ensure postconditions, in the presence of a false precondition).

Serving as an introduction to the next section, *Contract-Java*'s implementation of the quadratic equation solver methods follows:

```
static double sqrt(double x)
  requires x >= 0;
{ ... }
  ensures Math.abs(result*result - x) <= NEAR_ZERO;

static double[] quadraticEquationSolver(double a, double b, double c)
  requires a != 0;
{ ... }
  ensures areRoots(a, b, c, result);
```

Listing 1 Example of a Contract-Java array implementation.

```
public class Array<T>
{
  public invariant (isEmpty() && size() == 0) || // object's ADT
                   (!isEmpty() && size() > 0);   // invariant

  public Array(int size)
    requires size >= 0;                          // Constructor
  {                                              // precondition
    array = (T[]) new Object[size];
  }

  public int size()
  {
    return array.length;
  }
    ensures result >= 0;                         // size postcondition

  public T get(int idx)
    requires idx >= 0 && idx < size();           // get precondition
  {
    return array[idx];
  }

  public void set(int idx, T elem)
    requires idx >= 0 && idx < size();           // set precondition
  {
    array[idx] = elem;
  }
    ensures get(idx) == elem;                    // set postcondition

  protected T[] array;

  protected invariant array != null;             // internal representation
}                                                // invariant
```

5 *Contract-Java*

Contract-Java language was developed to ease and to take full advantage of DbC programming within Java, while attempting to retain usual syntactical and semantic choices in the language. A special care was taken both to disallow undesirable side-effects and to promote synergic behaviors with existing mechanisms (exceptions, for instance).

Unlike most existing approaches, *Contract-Java* makes DbC constructs normal language entities and fully implements all six requirements for DbC support as specified in section 3. As such, *Contract-Java* is defined as a *superset* of the Java language aiming the support of DbC[4]. All Java code is a valid *Contract-Java* code (obviously, the reverse is not true).

To achieve a full implementation of all six requirements, *Contract-Java* extends Java with nine new keywords: `invariant`, `requires`, `ensures`, `rescue`, `retry`, `check`, `local`, `old` and `result`.

Listing 1 shows an example of an array module's ADT in *Contract-Java*.

The syntax diagrams of the *Contract-Java* extensions to Java are showed in appendix 6.1.

5.1 Method Contracts

Contract-Java allows methods to be attached with a *precondition* and a *postcondition*. Such assertions must be defined near to the method declaration, and as close as possible as to where

[4] Minor incompatibilities may arise due to the new language keywords, although a proper compiler implementation might eliminate or reduce them to a minimum (`check`).

(when) they apply: precondition before the methods body, and postcondition afterwards (following Eiffel's approach). A contract might be applied to abstract methods (and even to an interface method declaration). All method interface contracts are inherited as described by requirement 4.

In the method's postcondition two new keyword are recognized in order to allow to express assertions using the method's result (if any), and the values of expressions when the method started its execution. Respectively: `result` and `old`.

No support exists yet for frame rules within method assertions to express what should *not change* during its execution.

5.2 Class Contracts

One or more invariant declarations might be declared in a *Contract-Java* class. Syntactically they are similar to method's preconditions and postconditions, except that its scope is within the class and its visibility can be specified (as happens with other class members).

The visibility definition of an invariant in *Contract-Java* is an interesting feature of the language because it allows the definition of different invariants applied to different abstraction levels (public, package, protected and private). Public invariants are the ADT's invariants. One the other hand, protected (or other) invariants are useful to express less abstract representation invariants[5]. These differences should be taken into consideration by *Contract-Java*'s automatic documentation tools. Listing 1 exemplifies the usage of both an ADT and a representation invariant.

When a class is a descendant of another class (or one or more interfaces) it inherits its invariants (as explained in requirement 4).

5.3 Java Interfaces

Interfaces specify an ADT without providing its implementation. As such, to completely specify the ADT, contracts need to be supported on interface classes. In the same way Native Java's throws are considered part of the class interface, contracts also belong to it. By implementing an interface, a class inherits the interface contracts. The same rules apply to inheritance on interfaces as to normal classes: interfaces are also ADT definitions and as such have the same treatment.

5.4 DbC Exceptions

The sixth requirement of our DbC requirements (section 3) is support for DbC exceptions.

To that goal, *Contract-Java* implements a set of DbC exceptions (descendant of `Error` type), for each type of assertion (although, as showed, such detail in not very important in DbC error handling, in which all that matter is if the method succeeds or has failed, and, if so, whose to blame for that). So, as part of the language specification, assertion errors will in fact be handled by Java's exception mechanism. However, it is ensure by the language semantics that it is impossible for a `try/catch/throw/throws` to use such exceptions[6]. These special exceptions can only be handled by the language's disciplined exception mechanism.

[5] Consistently, a private invariant will be hidden from descendant classes.
[6] Even catching `Throwable` does not catch a *Contract-Java* exception.

Listing 2 Example of real code of how to define a rescue clause on a class.

```
public class AFaultTolerantMethod
  // Does a path exist in labyrinth from src to dst?
  public boolean findPath(Labyrinth labyrinth, Location src, Locality dst)
    requires labyrinth != null; // this precondition is not rescued by
             src != null;       // this method's rescue clause.
             dst != null;
  local
    int attempt = 1;
  {
    boolean result = false;
    switch(attempt)
    {
      case 1:
        result=findPathAlg1(labyrinth,src,dst); // a method that tries
        break;                                  // to find the path
      case 2:
        result=findPathAlg2(labyrinth,src,dst); // another method that tries
        break;                                  // to find the path
    }
    return result;
  }
  rescue(RuntimeException e) // rescues both DbC exceptions and
  {                          // runtime exceptions.
    if (attempt < 2)
    {
      attempt++;
      retry;
    }
    // exception propagated to caller!
  }
```

This mechanism, whose syntax is showed in appendix 6.1, works in a similar way as Eiffel's original mechanism [21], but fully adapted to Java's mechanisms, in particular, exception handling.

Syntactically, methods were extended with an optional *rescue* clause able catch and handle contract failures within the execution of the method, and also (if desired) an optional *rescue* clause in which variables can be declared whose scope includes method's *body* and *rescue* clauses. This *local* clause allows the construction of rescue code which may depend on previous retried executions of the method.

A disciplined exception mechanism works as follows. With the important exception of the method's preconditions, all contract failures that occur during the method execution (including its postcondition, the invariant, and contract failures of called methods) are cached by the method's rescue clause. However, unlike catch blocks in usual exceptions mechanisms, rescue clauses are only allowed to retry the methods execution (command: `retry`), or re-propagate the failure to the caller method (if the rescue clause finishes without a retry command). Hence, they serve the purpose of an eventual object's cleanup, or to support a fault tolerant method.

Since some contract failures are sometimes implicit, and rely on the native exception mechanism (e.g. `NullPointerException`), *Contract-Java* extends rescue clause with an optional argument, quite similar to Java's 7 *catch* syntax, enabling the possibility to rescue non-DbC exceptions. This functionality could be extremely useful as it also eases the integration of legacy code within *Contract-Java*.

All these semantics is possible, because *Contract-Java* compiler is able to unambiguously distinguish *Contract-Java* classes and native Java's classes (both classes and object can coexist peacefully in a *Contract-Java* program).

Listing 2 exemplifies a fault tolerant method (allegedly it tries two algorithms for searching a path within a labyrinth).

The rescue clause must be associated to a non-abstract method. If the failure is of the method's responsibility (i.e., not on a precondition) then the execution jumps to the rescue clause where the failure is attempted to be dealt with. If the rescue clause reaches its end without a retry, or there is no rescue clause, the *Contract-Java* exception is rethrown, in order for the upper level of execution to be able to decide what to do with the error: either recover for it, or throw it again to its caller. In case a retry is done, the execution restarts on the beginning of the method; only local variables will keep its state.

5.5 Enhanced Debugging in *Contract-Java*

When checking for an assertion the programmer can define an appropriate error message to be used when that assertion fails. For example, the program:

```
public class TestAssert {
  static boolean boolFunc01() { return false; }
  static boolean boolFunc02() { return true; }
  static boolean boolFunc03() { return false; }

  public static void main(String[] args) {
    assert (boolFunc01() && boolFunc02()) || boolFunc03() : "message";
  }
}
```

would yield the result:

```
$ java -ea TestAssert
Exception in thread "main"
    java.lang.AssertionError: message
    at TestAssert.main(TestAssert.java:16)
```

However, in *Contract-Java* the error messages associated with the assertion failures are automatically enhanced with relevant debugging information, such as the boolean expression that failed, and an expansion of the various expression values, thus reducing the need of manual definition of the assertions' associated text and providing a clearer view of why the assertion failed. If the following example program is executed:

```
public class TestBooleanExpansion {

  boolean boolFunc01() { return false; }
  boolean boolFunc02() { return true; }
  boolean boolFunc03() { return false; }

  public void doSomething()
  requires (boolFunc01() && boolFunc02()) ||
           boolFunc03();
  { ... }
}
```

it could be created the following output:

```
$ java TestBooleanExpansion
Exception in thread "main"
  Contract_JavaPreconditionFailure
    at TestBooleanExpansion.main
  TestBooleanExpansion.java:16)
Precondition failed: boolFunc01() &&
  boolFunc02() || boolFunc03()
   boolFunc01() && boolFunc02() ||
  boolFunc03() => false;
   boolFunc01() && boolFunc02() => false;
   boolFunc01() => false;
   boolFunc02() => true;
   boolFunc03() => false;
```

5.5.1 Fine-tuning

Contract-Java allows the possibility to fine-tune the activation and deactivation of assertions (by assertion kind, to the whole program, to packages, or class by class). However, unlike Java's native `assert`, contracts are defined at compile time.

5.6 Other Assertions

We support the equivalent to the `assert` keyword from Java, namely `check`. This instruction allows for the verification of a boolean expression triggering a failure when it is not true, of type `checkFailure`.

5.7 *Contract-Java* Native Library

Contract-Java does not support contracting preexisting class files. The alternative is to create wrapper classes in order to encapsulate access to a preexisting class file, adding the desired contracts on the wrapper class.

Since Java uses defensive programming, *Contract-Java* would probably benefit from an effort to contractualize Java libraries providing new libraries which wrap and contractualize native libraries, which would lead to new classes[7] being defined.

5.8 Documentation

The support for full automatic documentation generation is the fifth requirement of our DbC requirements (Section 3). All contracts (with the exception of non-public invariants) must be extracted from the definition and automatically added to the documentation. In the cases where inheritance is involved, each class documentation should allow a full listing of the ADT definition, namely make documentation available in "flat" form (which includes all available public methods, their contracts and the class public invariant). Such documentation would be extracted using another tool, which would generate javadoc-like documentation with the addition of contract information.

6 Conclusion

We have presented and justified the requirements needed to completely implement Design by Contract in an Object-Oriented programming language. Systematic error handling approaches, such as defensive programming, were critically analyzed and compared with DbC alternative. It was showed that typed exceptions and `try/catch` instructions break algorithmic abstraction of methods; hence becoming a questionable alternative to handle errors in a program. A better approach, based on DbC was presented and justified.

A new DbC language extension to Java – named *Contract-Java* – was presented. This new language accepts all existing Java code, and was implemented in order to avoid undesirable side-effects with Java's native mechanisms. Is was given a special care to error handling, thus, *Contract-Java* implements a disciplined exception mechanism, preventing exceptions from being ignored, and easing the development of fault-tolerant programs. To better integrate with existing Java code, this new exception mechanism is able to integrate any desired

[7] With a different ADT, since Java native classes follow defensive programming and have no regard for command-query separation.

non-DbC exceptions within the DbC error handling mechanism. Finally, a new enhanced debugging mechanism was implemented in which relevant information is automatically generated and printed in the presence of a contract failure.

6.1 Future Developments

We expect to enhance *Contract-Java* with mechanisms for pure query detection (assertions should use only expressions without side-effects to the program's state).

Some work on frame rules (predicates expressing what did not change), is also one of our objectives. Also, the implementation of other kinds of assertions like loop invariants

Finally, work in on the way for a concurrent versions of *Contract-Java* (*Concurrent Contract-Java*) adapting one of the authors PhD work [24, 25] to Java.

Acknowledgment. This work was partially supported by IEETA (Instituto de Engenharia Electrónica e Telemática de Aveiro), funded by FCT's project PEst-OE/EEI/UI0127/2014.

Contract-Java new Syntax

```
classDeclaration: modifiers "class" name ( typeParameters )?
   "{" ( ";" | staticBlock | interfaceDeclaration |
       classDeclaration | field | method | invariant )+ "}"

method: modifiers type name "(" ( arguments )? ")"
    ( precondition )?
    ( ( local )? "{" body "}" )?
    ( postcondition )?
    ( rescue )?

invariant: ("public" | "protected" | "" | "private" )
    "invariant" ( assertionClause )+

precondition: "requires" ( assertionClause )+

postcondition: "ensures" ( assertionClause )+

assertionClause: conditionalExpression
    ( ":" expression )? ";"

rescue: "rescue" ( excpDecl )? "{" blockStatement "}"

excpDecl: "(" excpTypeList name ")"

excpTypeList: excpType ( "|" excpType )*
```

References

1 Sérgio Agostinho. An aspect-oriented infrastructure for design by contract in java. Master's thesis, Universidade Nova de Lisboa, 2008.

2 D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, October 2001.

3 Chien-Tsun Chen, Yu Chin Cheng, and Chin-Yun Hsieh. Contract specification in java: Classification, characterization, and a new marker method. *IEICE – Trans. Inf. Syst.*, E91-D(11):2685–2692, November 2008.

4 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976.

5 Robert Floyd and J. T. Schwartz. Assigning meanings to programs. In *Proceedings of a Symposium on Applied Mathematics*, volume 19, pages 19–31, 1967.

6 Alessandro F. Garcia, Cecília M. F. Rubira, Alexander Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, November 2001.

7 J. A. Goguen, J. W. Thatcher, E. G. Wagner, and R. Yeh. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology: Data Structuring*, volume 4, pages 80–149. Prentice–Hall, 1978.

8 David Gries. *The Science of Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1987.

9 C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

10 C. B. Jones. *Systematic software development using VDM.* Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1986.

11 Cliff B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1980.

12 Nhat Minh Lê. Contracts for java: A practical framework for contract programming. Technical report, Google Switzerland GmbH, 2011.

13 Gary T. Leavens. The java modeling language (jml) online page. `http://www.eecs.ucf.edu/~leavens/JML/download.shtml`, August 2013.

14 Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

15 Barbara Liskov and Stephen Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, March 1974.

16 Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

17 B. H. Liskov and A. Snyder. Exception handling in clu. *IEEE Transactions on Software Engineering*, 5(6):546–558, 1979.

18 B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, October 1992.

19 Bertrand Meyer. Technical report tr-ei-12/co. Technical report, Interactive Software Engineering Inc., 1986.

20 Bertrand Meyer. Eiffel: A language and environment for software engineering. *The Journal of Systems and Software*, 1988.

21 Bertrand Meyer. *Object-oriented software construction.* Prentice-Hall, New York, 1988.

22 Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, 2nd edition, 1997.

23 Bertrand Meyer. Software architecture: Lecture 4: Design by contract. `http://se.inf.ethz.ch/old/teaching/ss2007/0050/slides/04_softarch_contract_6up.pdf`, ETHZ, March-July 2007.

24 Miguel Oliveira e Silva. Concurrent object-oriented programming: The mp-eiffel approach. *Journal of Object Technology*, 3(4):97–124, April 2004. Proceedings of the TOOLS USA 2003 Conference, September 30 to October 1, 2003 – Santa Monica, CA.

25 Miguel Oliveira e Silva. Automatic realizations of statically safe intra-object synchronization schemes in MP-Eiffel. In *Proceedings of the first Symposium on concurrency, Real-Time, and Distribution in Eiffel-Like Languages, CORDIE'06*, pages 91–118. University of York – Department of Computer Science, July 2006. Available at `http://www.ieeta.pt/~mos/pubs`.

26 J. Rieken. Design by contract for java-revised. *Master's thesis, Department für Informatik, Universität Oldenburg*, 2007.

27 A. Turing. Checking a large routine. In Martin Campbell-Kelly, editor, *The Early British Computer Conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

Implementing Python for DrRacket

Pedro Palma Ramos and António Menezes Leitão

INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol 9, Lisboa, Portugal
{pedropramos,antonio.menezes.leitao}@tecnico.ulisboa.pt

Abstract

The Python programming language is becoming increasingly popular in a variety of areas, most notably among novice programmers. On the other hand, Racket and other Scheme dialects are considered excellent vehicles for introducing Computer Science concepts. This paper presents an implementation of Python for Racket and the DrRacket IDE. This allows Python programmers to use Racket libraries and vice versa, as well as using DrRacket's pedagogic features. In particular, it allows architects and designers to use Python as a front-end programming language for Rosetta, an IDE for computer-aided design, whose modelling primitives are defined in Racket.

Our proposed solution involves compiling Python code into equivalent Racket source code. For the runtime implementation, we present two different strategies: (1) using a foreign function interface to borrow the data types and primitives from Python's virtual machine or (2) implementing Python's data model over Racket data types.

While the first strategy is easily implemented and provides immediate support for Python's standard library and existing third-party libraries, it suffers from performance issues: it runs, at least, one order of magnitude slower when compared to Python's reference implementation.

The second strategy requires us to implement Python's data model in Racket and port all of Python's standard library, but it succeeds in solving the former's performance issues. Furthermore, it makes interoperability between Python and Racket code easier to implement and simpler to use.

1998 ACM Subject Classification D.3.4 Programming Languages: Processors

Keywords and phrases Python, Racket, language implementations, compilers

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.127

1 Introduction

Architects who use computer-aided design (CAD) applications are beginning to shift from a traditional approach to an algorithmic approach. This is leading to an increasing need for the CAD community to master generative design, a design method based on a programming approach which allows them to build complex three-dimensional structures that can then be effortlessly modified through simple changes in a program's code or parameters. It is, thus, increasingly important for architects to master programming techniques.

Although most CAD applications provide programming languages for generative design, programs written in these languages have very limited portability, as each CAD application provides its own specific language and functionality. Therefore, a program written for one CAD application cannot be used on other CAD applications. In addition to this, these programming languages are rarely pedagogical and most of them are poorly designed or obsolete.

With this in mind, we are developing Rosetta, an extensible integrated development environment (IDE) for generative design [11]. Rosetta seeks to answer the portability problem by allowing the development of programs in different programming languages


© Pedro Ramos and António Menezes Leitão;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 127–141

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

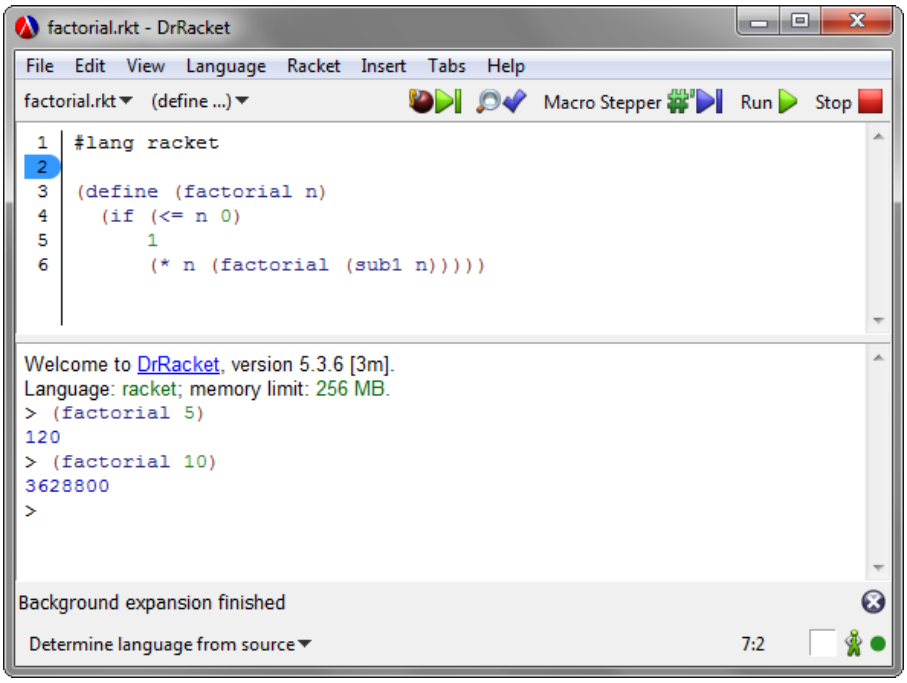


Figure 1 DrRacket's graphical user interface.

which are then portable across different CAD applications. A program written in one of the supported programming languages is compiled to an intermediate language, where the primitives essential to 3D modelling are defined. These primitives are then translated to commands of the selected CAD application's interface and invoked through interprocess communication. This design allows Rosetta to support new front-end programming languages and new back-end CAD applications, independently from each other.

Currently, Rosetta is based on the DrRacket IDE and uses Racket as its intermediate language. DrRacket (formerly known as DrScheme) is a pedagogic IDE for the Racket programming language, a dialect of LISP and a descendant of Scheme [3, 4].

Unlike IDEs such as Eclipse or Microsoft Visual Studio, DrRacket provides a simple and straightforward interface (Figure 1). It also provides a set of tools, including a syntax highlighter, a syntax checker, a macro stepper and a debugger, aimed at inexperienced programmers, such as the target audience of Rosetta.

Additionally, Racket and DrRacket support the development and extension of other programming languages [20], which is crucial for Rosetta's front-end extensibility. Currently, Rosetta supports front-ends for Racket, AutoLISP, JavaScript and RosettaFlow (a graphical language inspired by Grasshopper). AutoLISP and JavaScript were chosen precisely because they have been used for generative design. More recently, the Python language has emerged as a good candidate for this area of application.

Python is a high-level, interpreted, dynamically typed programming language [23, p. 3]. It supports the functional, imperative and object-oriented programming paradigms and features automatic memory management. It is mostly used for scripting, but it can also be used to build large scale applications. Its reference implementation, CPython, is written in C and it is maintained by the Python Software Foundation. There are also other third-party implementations such as Jython (written in Java), IronPython (written in C#) and PyPy (written in Python).

Due to its large standard library, expressive syntax and focus on code readability, Python is becoming an increasingly popular programming language in many areas, including architecture. Python has been receiving a lot of attention in the CAD community, particularly after it has been made available as scripting language for CAD applications such as Rhino or Blender. This justifies the need for implementing Python as another front-end language of Rosetta, i.e. implementing Python in Racket.

Therefore, our goal is to develop a correct and efficient implementation of the Python language for Racket, which is capable of interfacing Python and Racket code. This will allow Rosetta users to use Python as a front-end programming language for generative design. Additionally, we want to support some of DrRacket's features for Python development, namely syntax highlighting, syntax checking and debugging.

In the next sections, we will briefly examine the strengths and weaknesses of other Python implementations, describe the approaches we took for our own implementation and showcase the results we have obtained so far.

2 Related Work

There are a number of Python implementations that are good sources of ideas for our own implementation. In this section we describe the most relevant ones.

2.1 CPython

CPython, maintained by the Python Software Foundation, is written in the C programming language and has been the reference implementation of Python since its first release. It parses Python source code (from `.py` files or interactive mode) and compiles it to bytecode, which is then interpreted on a virtual machine.

The Python standard library is implemented both in Python and C. In fact, CPython makes it easy to write third-party module extension in C to be used in Python code. The inverse is also possible: one can embed Python functionality in C code, using the Python/C API [22].

2.1.1 Object Representation

CPython's virtual machine is a simple stack machine, where the byte codes operate on a stack of `PyObject` pointers [21].

At runtime, every Python object has a corresponding `PyObject` instance. A `PyObject` contains a reference counter, used for garbage collecting, and a pointer to a `PyTypeObject`, which is another `PyObject` that indicates the object's type. In order for every value to be treated as a `PyObject`, each built-in type is declared as a structure containing these two fields, plus any additional fields specific to that type.

This means that everything is allocated on the heap, even basic types. To avoid relying too much on expensive dynamic memory allocation, CPython enforces two strategies:

- Only requests larger than 256 bytes are handled by `malloc` (the C standard allocator), while smaller ones are handled by pre-allocated memory pools.
- There is a pool for commonly used immutable objects (such as the integers from -5 to 256). These are allocated only once, when the virtual machine is initialized. Each new reference to one of these integers will point to the instance on the pool instead of allocating a new one.

2.1.2 Garbage Collection and Threading

Garbage collection in CPython is performed through reference counting. Whenever a new Python object is allocated or whenever a new reference to it is made, its reference counter is incremented. When a reference is no longer needed, the reference counter is decremented. When the reference counter reaches zero, the object's finalizer is called and the space is reclaimed.

Reference counting, however, does not work well with reference cycles [24, ch. 3.1]. Consider the example of a list containing a reference to itself. When its last reference goes out of scope, its counter is decremented, however the circular reference inside the list is still present, so the reference counter will never reach zero and the list will not be garbage collected, even though it is already unreachable.

Furthermore, these reference counters are not thread-safe [25]. If two threads would attempt to increment an object's reference counter simultaneously, it would be possible for this counter to be erroneously incremented only once. To avoid this from happening, CPython enforces a global interpreter lock (GIL), which prevents more than one thread running interpreted code at the same time.

This is a severe limitation to the performance of threads on CPU-intensive tasks. In fact, using threads will often yield a worse performance than using a sequential approach, even on a multiple processor environment [1]. Therefore, the use of threads is only recommended for I/O tasks [2, p. 444].

Note that the GIL is a feature of CPython and not of the Python language. This feature is not present in other implementations such as Jython or IronPython, which will be described in the following section.

2.2 Jython

Jython is another Python implementation, written in Java and first released in 2000. Similarly to how CPython compiles Python source-code to bytecode that can be run on its virtual machine, Jython compiles Python source-code to Java bytecode, which can then be run on the Java Virtual Machine (JVM).

2.2.1 Implementation Differences

There are some aspects of Python's semantics in which Jython's implementation differs from CPython's [10]. Some of these are due to limitations imposed by the JVM, while others are considered bugs in CPython and, thus, were implemented differently in Jython.

The standard library in Jython also suffers from minor differences from the one implemented in CPython, as some of the C-based modules have been rewritten in Java.

2.2.2 Java Integration

Jython programs cannot use module extensions written for CPython, but they can import Java classes, using the same syntax for importing Python modules.

There is work being done by a third-party [18] to integrate CPython module extensions with Jython, through the use of the Python/C API. This would allow using NumPy and SciPy with Jython, two very popular Python libraries which rely on CPython's ability to run module extensions written in C.

2.2.3 Performance

It is worth noting that garbage collection is performed by the JVM and does not suffer from the issues with reference cycles that plague CPython [9, p. 57]. Furthermore, there is no global interpreter lock, so threads can take advantage of multi-processor architectures for CPU-intensive tasks [9, p. 417].

Performance-wise, Jython claims to be approximately as fast as CPython. Some libraries are known to be slower because they are currently implemented in Python instead of Java (in CPython these are written in C). Jython's performance is also deeply tied to performance gains in the Java Virtual Machine.

2.3 IronPython

IronPython, developed as a follow-up to Jython, is an implementation of Python for the Common Language Infrastructure (CLI). It is written in C# and was first released in 2006. It compiles Python source-code to CLI bytecode, which can be run on Microsoft's .NET framework or Mono (an open-source alternative implementation of the CLI).

IronPython provides support for importing .NET libraries and using them with Python code [13]. As it happened with Jython, there is work being done by a third-party in order to integrate CPython module extensions with IronPython [8].

As far as performance goes, IronPython claims to be 1.8 times faster than CPython on `pystone`, a Python benchmark for showcasing Python's features. Additionally, further benchmarks demonstrate that IronPython is slower at allocating and garbage collecting objects and running code with `eval`. On the other hand, it is faster at setting global variables and calling functions [6].

2.4 PyPy

PyPy is yet another Python implementation, written in a restricted subset of Python, RPython[1]. It was first released in 2007 and currently its main focus is on speed, claiming to be 6.2 times faster than CPython in a geometric average of a comprehensive set of benchmarks [17].

It supports all of the core language, most of the standard library and even some third party libraries. Additionally, it features incomplete support for the Python/C API [16].

PyPy includes two very distinct modules: a Python interpreter and the RPython translation toolchain [15]. Like the implementations mentioned before, the interpreter converts user's Python source code into bytecode.

However, what distinguishes it from those other implementations is that this interpreter, written in RPython, is in turn compiled by the RPython translation toolchain, effectively converting Python code to a lower level platform (typically C, but the Java Virtual Machine and Common Language Infrastructure are also supported).

The translation toolchain consists of a pipeline of transformations (flow analysis, annotator, backend optimizations, among others), but what truly makes PyPy stand out as currently the fastest Python implementation is its just-in-time compiler (JIT), which detects common codepaths at runtime and compiles them to machine code, optimizing their speed.

[1] RPython (Restricted Python) is a heavily restricted subset of Python, in order to allow static inference of types. For instance, it does not allow altering the contents of a module, creating functions at runtime, nor having a variable holding incompatible types.

Table 1 Comparison between implementations.

	Language(s) written	Platform(s) targetted	Speedup (vs CPython)	Std. library support
CPython	C	CPython's VM	1×	Full
Jython	Java	JVM	∼ 1×	Most
IronPython	C#	CLI	∼ 1.8×	Most
PyPy	RPython	C, JVM, CLI	∼ 6×	Most
PLT Spy	Scheme, C	Scheme	∼ 0.001×	Full

The JIT keeps a counter for every loop that is executed. When it exceeds a certain threshold, that codepath is recorded and compiled to machine code. This means that the JIT works better for programs without frequent changes in loop conditions.

2.5 PLT Spy

PLT Spy is an experimental Python implementation written in PLT Scheme and C, first released in 2003. It parses and compiles Python source-code into equivalent PLT Scheme code [12].

PLT Spy's runtime library is written in C and interfaces with Scheme via the PLT Scheme C API. It implements Python's built-in types and operations by mapping them to the CPython virtual machine, through the use of the Python/C API. This allows PLT Spy to support every library that CPython supports (including NumPy and SciPy).

This extended support has a big trade-off in portability, though, as it led to a strong dependence on the 2.3 version of the Python/C API library and does not seem to work out-of-the-box with newer versions. More importantly, the repetitive use of Python/C API calls and conversions between Python and Scheme types severely limited PLT Spy's performance. PLT Spy's authors use anecdotal evidence to claim that it is around three orders of magnitude slower than CPython.

2.6 Comparison

Table 1 displays a rough comparison between the implementations discussed above.

To sum up, PLT Spy can interface Python code with Scheme code and is the only alternative implementation which can effortlessly support all of CPython's standard library and third-party modules extensions, through its use of the Python/C API. However, the performance cost that results from the repeated conversion of data from Scheme's internal representation to CPython's internal representation is unacceptable.

Furthermore, our implementation will require using Racket's bytecode and tools in order to support Rosetta's modelling primitives (defined in Racket), therefore PyPy's performance strategy is not feasible for our problem.

On the other hand, Jython and IronPython show us that it is possible to implement Python's semantics over high-level languages, with very acceptable performances and still provide means for importing that language's functionality into Python programs. However, Python's standard library needs to be manually ported.

With this in mind, we will be presenting our proposed solution in the next section.

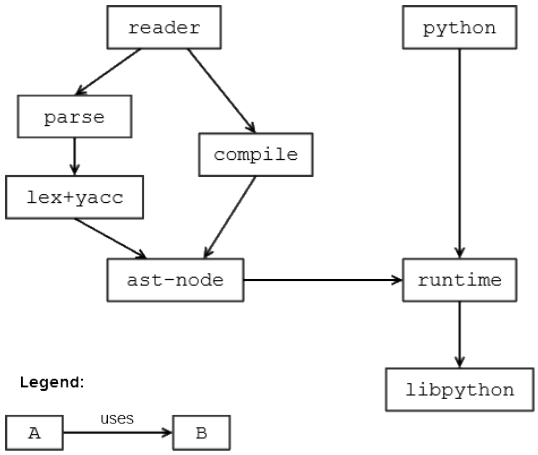


Figure 2 Dependencies between modules. The arrows indicate that a module uses functionality that is defined on the module it points to.

3 Solution

Our proposed solution consists of two compilation phases:

1. Python source-code is compiled to Racket source-code;
2. Racket source-code is compiled to Racket bytecode.

In phase 1, the Python source code is parsed into a list of abstract syntax trees, which are then expanded into a list of syntax objects containing equivalent Racket code.

In phase 2, the Racket source-code generated above is fed to a bytecode compiler which performs a series of optimizations (including constant propagation, constant folding, in-lining, and dead-code removal). This bytecode is interpreted on the Racket VM, where it may be further optimized by a JIT compiler.

Note that phase 2 is automatically performed by the Racket implementation, therefore our implementation effort relies only on a source-to-source compiler from Python to Racket.

3.1 General Architecture

Figure 2 summarises the dependencies between the different Racket modules of the proposed solution. The next paragraphs provide a more detailed explanation of these modules.

3.1.1 Racket Interfacing

A Racket file usually starts with the line `#lang <language>` to indicate which language is being used (in our case, it will be `#lang python`).

To define a language for the Racket platform, Racket requires only two files: one which defines how source code compiles to Racket code, and another which defines the functions and macros that make up the compiled code. These correspond, in our solution, to the `reader` and `python` modules, respectively.

The `python` module simply provides all the bindings from the Racket language and the bindings defined at the `runtime` module (which will be described at 3.1.3). The `reader` module must provide:

- The `read` function, which takes a Racket input-port as an argument (this could be the standard input, a file, a string, etc.) and return a list of s-expressions, which correspond to the Racket code compiled from the input port;
- The `read-syntax` function, which also takes an input-port as argument and returns a list of syntax-objects, which correspond to the compiled Racket code.

Syntax-objects are Racket's native representation for code. They abstract over s-expressions, but they may also keep source location information (file, line number, column number and span) and lexical binding information about said code. By keeping track of the original position of each token during the parsing process and copying it to the compiled syntax-objects, each compiled s-expression can be mapped back to the original Python expression it corresponds to. This way, our implementation fully integrates with DrRacket's features such as signalling the line number where an error has occurred or tracking the location of a debugging session on the source code.

3.1.2 Parse and Compile Modules

The `lex+yacc` module defines a set of Lex and Yacc rules for parsing Python code, using the Lex/Yacc implementation provided by Racket's `parser-tools` library. This outputs a list of abstract syntax trees (ASTs), which are defined in the `ast-node` module. These nodes are implemented as Racket objects. Each subclass of an AST node defines its own `to-racket` method, responsible for the code generation. A call to `to-racket` works in a top-down recursive manner, as each node will eventually call `to-racket` on its children.

The `parse` module simply defines a practical interface of functions for converting the Python code from an input port into a list of ASTs, using the functionality from the `lex+yacc` module.

In a similar way, the `compile` module defines a practical interface of functions for converting lists of ASTs into syntax objects with the compiled code, by calling the `to-racket` method on each AST.

3.1.3 Runtime Modules

The `libpython` module defines a foreign function interface to the functions provided by the Python/C API. Its use will be explained in detail on the next section.

Compiled code contains references to Racket functions and macros, as well as some additional functions which implement Python's primitives. For instance, we define `py-add` as the function which implements the semantics of Python's `+` operator. These primitive functions are defined in the `runtime` module.

3.2 Runtime Implementation using Racket's Foreign Function Interface

For the runtime, we started by following a similar approach to PLT Spy, by mapping Python's data types and primitive functions to the Python/C API. The way we interact with this API, however, is radically different.

On PLT Spy, this was done via the PLT Scheme C API, and therefore the runtime is implemented in C. This entails converting Scheme values into Python objects and vice-versa for each runtime call. Besides the performance issue (described on the Related Work section), this method is cumbersome and lacks portability, since it requires compiling the runtime with a platform-specific C compiler, and to do so each time the runtime is modified.

Instead, we used the Racket Foreign Function Interface (FFI) to directly interact with the foreign data types created by the Python/C API, therefore our runtime is implemented in Racket. These foreign functions are defined on the `libpython` modules, according to their C signatures, and are called by the functions and macros defined on the `runtime` module.

The values passed around correspond to pointers to objects in CPython's virtual machine, but there is sometimes the need to convert them back to Racket data types, so they that can be used in control flow forms like `if`s and `cond`s.

As with PLT Spy, this approach only requires implementing the Python language constructs, because the standard library and other libraries installed on CPython's implementation are readily accessible.

Unfortunately, as we will show in the Performance section, the repetitive use of these foreign functions introduces a huge overhead on our primitive operators, resulting in a very slow implementation.

Additionally, objects allocated with the Python/C API must have their reference counters explicitly decremented, or they will not be garbage collected. This can be solved by attaching a finalizer to each FFI function that allocates a new Python object. This finalizer is responsible for decrementing the object's reference counter when Racket's GC proves that there are no more live references to the Python object. While this solves the garbage collection issue, it entails having another layer of expansive FFI calls, which degrade the runtime performance.

For these reasons, we've tried a second approach, which is described in the following section.

3.3 Runtime Implementation using Racket

Our second approach consists in implementing Python's data model purely in Racket.

In Python, every object has an associated type-object (where every type-object's type is the `type` type-object). A type-object contains a hash table which maps operation names (strings) to the respective functions that type supports (function pointers, in CPython).

As a practical example, in the expression `obj1 + obj2`, the behaviour of the plus operator is determined at runtime, by computing `obj1`'s type-object and looking up the string `__add__` in its hash table. This dynamism in Python allows objects to change behaviour during the execution of a program, simply by adding, modifying of deleting entries from these hash tables, but it also forces an interpreter to constantly lookup these behaviours, contributing to Python's slow performance when compared to other languages.

In CPython, an object's type is stored at the `PyObject` structure. We can use the same strategy in Racket, but there is a nicer alternative. In Racket, one can recognize a value's type through its predicate (`number?`, `string?`, etc.). A Python object's type never changes, so we can directly map basic Racket types into Python's basic types and make their types available through a pattern matching function, which returns the most appropriate type-object, according to the predicates that value satisfies.

This way, we avoid the overhead from constantly wrapping and unwrapping frequently used values from the structures that hold them and it also leads to a cleaner interoperability between Racket and Python code, from the user's perspective. Complex built-in types, such as type-objects, are still implemented through Racket structures.

Comparing it to the FFI approach, this one entails implementing all of Python's standard library in Racket, but, on the other hand, it is much faster and provides reliable memory management of Python's objects, since it does not need to coordinate with another virtual machine.

3.4 Examples

In this section we provide some examples of the current state of the translation between Python and Racket. Note that this is still a work in progress and, therefore, the compilation results of these examples are likely to change in the future.

3.4.1 Fibonacci

Consider the following program in Racket which implements a naive algorithm for computing the Fibonacci function:

```
1 (define (fib n)
2   (cond
3     [(= n 0) 0]
4     [(= n 1) 1]
5     [else (+ (fib (- n 1))
6              (fib (- n 2)))]))
7
8 (fib 30)
```

Its equivalent in Python would be:

```
1 def fib(n):
2   if n == 0: return 0
3   elif n == 1: return 1
4   else: return fib(n-1) + fib(n-2)
5
6 print fib(30)
```

Currently, this code is compiled to:

```
1 (define (:fib :n)
2   (cond
3    ((py-truth (py-eq :n 0)) 0)
4    ((py-truth (py-eq :n 1)) 1)
5    (else (py-add (py-call :fib (py-sub :n 1))
6                  (py-call :fib (py-sub :n 2))))))
7
8 (py-print (py-call :fib 30))
```

Starting with line 1, the first thing one might notice is the colon prefixing the identifiers `fib` and `n`. This has no syntactic meaning in Racket; it is simply a name mangling technique to avoid replacing Racket's bindings with bindings defined in Python. For example, one might set a variable `cond` in Python, which would then be compiled to `:cond` and therefore would not interfere with Racket's built-in `cond`. This also prevents Racket bindings from leaking into Python user code. For instance, the functions `car` and `cdr` will only be available in Python if explicitly imported from Racket.

The functions and macros starting with the `py-` prefix are all defined on the `runtime` module. The functions `py-eq`, `py-add`, and `py-sub` implement the semantics of the Python operators `==`, `+`, and `-`, respectively. The function `py-truth` takes a Python object as argument and returns a Racket Boolean value, `#t` or `#f`, according to Python's semantics for Boolean values. This conversion is necessary because, in Racket, only `#f` is treated as false, while, in Python, the Boolean value false, zero, the empty list and the empty dictionary, among others, are all treated as false when used on the condition of an `if`, `for` or `while`

statement. Finally, `py-call` and `py-print` implement the semantics of function calling and the print statement, respectively.

Implementing a runtime function such as `py-add`, with the FFI strategy, is literally as simple as calling two functions from the foreign interface: one for getting the `__add__` attribute from the first operand's type-object and another to call it with the correct arguments. With the Racket runtime strategy, this entails implementing the attribute referencing and method calling semantics. Additionally, to fully cover the semantics of the plus operator, we'll have to implement the `__add__` method for every built-in type that supports it.

The compilation process is independent of the runtime strategy used. Since literals values are pre-computed at compile-time, the only difference in the compilation results are the literals (using the FFI approach, the literals `0`, `1`, `2` and `30` would be foreign pointers to corresponding CPython integer objects). This means that moving from the FFI to the Racket strategy does not entail changing the compiler.

As a final remark for this example, notice that except for the added verboseness, the original Racket code and the compiled code are essentially the same. This is relevant to ensure that DrRacket/Rosetta users that program in Python get a coherent behaviour from DrRacket's step-by-step debugger.

3.4.2 Sieve of Eratosthenes

Consider now a Racket program which implements a variation of the sieve of Eratosthenes, that counts the number of primes below a given number, as shown on listing 1. Its Python equivalent could be implemented as presented on listing 2.

This program presents some other compilation challenges, in order to preserve Python's semantics for binding scopes and control flow.

First of all, in Python we can assign new local variables anywhere, as shown in line 2, while in Racket they have to be declared, e.g., with a `let` form. In Python, only modules, class definitions, function definitions and lambda define a new binding scope. Unlike such languages as C and Java, compound statements (i.e. "blocks") in Python do not define their own scope. Therefore, a variable which is first assigned in a compound statement is also visible outside of it.

This can be implemented by enclosing the body of a function definition inside a `let` form containing all the referenced local variables. This way, Python assignments can be mapped to `set!` forms. Global assignments follow a similar strategy: the variable is first "declared" with a `define` form and then it can be `set!`.

Listing 1 Racket implementation for Sieve of Eratosthenes.

```
1  (define (sieve n)
2    (let ([primes (make-vector n #t)]
3          [counter 0])
4      (for ([i (in-range 2 n)])
5        (when (vector-ref primes i)
6          (set! counter (add1 counter))
7          (for ([j (in-range (* i i) n i)])
8            (vector-set! primes j #f))))
9      counter))
10
11 (sieve 10000000)
```

Listing 2 Python implementation for Sieve of Eratosthenes.

```
def sieve(n):
  primes = [True] * n
  counter = 0
  for i in range(2,n):
    if primes[i]:
      counter = counter + 1
      for j in range(i*i, n, i):
        primes[j] = False
  return counter

print sieve(10000000)
```

As for Python's `for` statements, Racket provides a `for` macro with similar semantics, however Python's `for` loop allows using `break` and `continue` statements to alter the control flow inside the loop.

We implement it with our own `py-for` macro. It expands to a named `let` which updates the control variables, evaluates the `for`'s body and recursively calls itself, repeating the cycle with the next iteration. A `continue` statement is implemented via calling the named `let` before the end of the body, thus starting a new iteration of the loop, while a `break` statement is handled with escape continuations. Listing 3 shows the final program.

Listing 3 Sieve of Eratosthenes Python version implemented in Racket.

```
(define (:sieve :n)
  (let ([:j (void)]
        [:i (void)]
        [:counter (void)]
        [:primes (void)])
    (begin
      (set! :primes (py-mul (make-py-list :True) :n))
      (set! :counter 0)
      (py-for continue43341
       [:i (py-call :range 2 :n)]
       (cond
        ((py-truth (py-index :primes :i))
         (begin
           (set! :counter (py-add :counter 1))
           (py-for continue50281
            [:j (py-call :range (py-mul :i :i) :n :i)]
            (py-set-index :primes :j :False))))
        (else py-None)))
      :counter)))

(py-print (py-call :sieve 10000000))
```

3.5 Performance

In this section we discuss the performance of our implementation and we compare it to the official Python implementation and to the semantically equivalent Racket code.

The charts on Figure 3 compare the running time of these examples for:

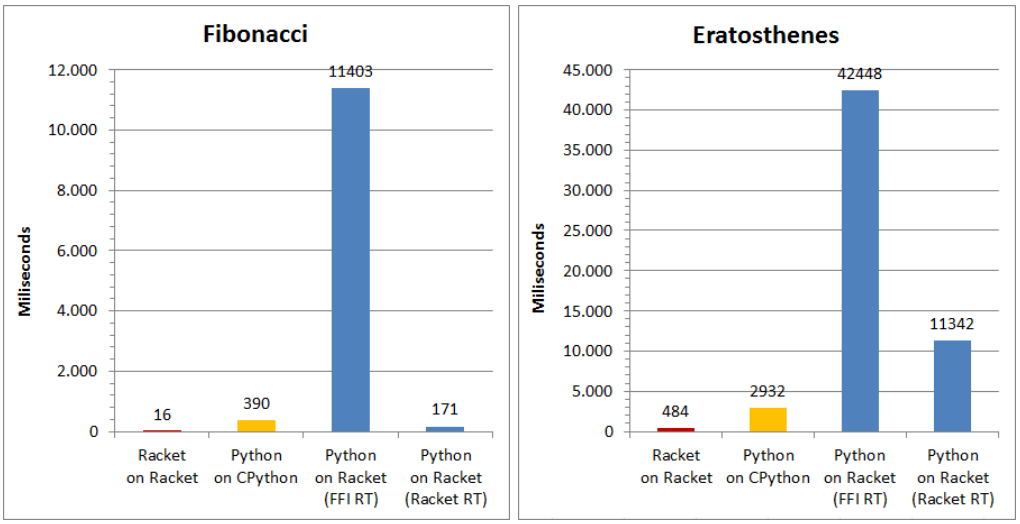


Figure 3 Benchmarks of the Fibonacci and Sieve of Eratosthenes examples.

- Racket code running on Racket;
- Python code running on CPython;
- Compiled Python code running on Racket with the FFI runtime approach;
- Compiled Python code running on Racket with the Racket runtime approach.

These benchmarks were performed on an Intel® Core™ i7 processor at 3.2GHz running under Windows 7. The times below represent the minimum out of 3 samples.

It can be seen that with our first approach (runtime implemented with foreign functions to CPython's virtual machine), Python code running on Racket is currently about 20-30 times slower than the same Python code running on CPython. This is mainly due to the overhead from the FFI calls, which is especially significant since simple operations like an addition entail several FFI calls.

The times presented for this approach do not include the overhead from the finalizers, which severely penalizes execution times. Benchmarks with these examples point to an increase in execution times by a factor between 100% and 150%.

With our second approach (runtime implemented purely on Racket), the Fibonacci example running on Racket is now faster than the same code running on CPython. This can be attributed to Racket's lighter function calls and more efficient primitive operators. We have optimized the use of operators on commonly used types. Since most uses of the `+` and `-` are for numbers, our implementation of these operators tests this case and dispatches the corresponding Racket function for number addition and subtraction, instead of invoking Python's heavier method dispatching mechanism. This is a valid approach, because it is not possible, in Python, to change the predefined semantics of the built-in types.

The Sieve of Eratosthenes example still runs slower than in CPython, but it's performance is quite acceptable for our current goals.

4 Conclusions

There is a need for an implementation of Python for the Rosetta community, and, more specifically, for Racket users. This implementation must be able to interact with Racket libraries and should be as close as possible to other state-of-the-art implementations in terms of performance.

Our solution follows a traditional compiler's approach, as a pipeline of scanner, parser and code generation. Python source-code is, thus, compiled to equivalent Racket source-code. This Racket source-code is then handled by Racket's bytecode compiler, JIT compiler and interpreter.

We have presented two approaches for the runtime implementation. The first one makes use of Racket's Foreign Interface and the Python/C API handle Python objects in CPython's virtual machine. This allows our implementation to effortlessly support all of Python's standard library and even third-party libraries written in C. On the other hand, it suffers from bad performance (at least one order of magnitude slower than CPython).

It is worth noting, however, that this approach can be used for quickly implementing a runtime for other interpreted languages, provided that they have an API which allows foreign function access to their functionality. Such languages include Ruby (using the Ruby C API [19]), Lua (using the Lua C API [7]) and SQL (using the MySQL C API [26] or PostgreSQL's `libpq` [14]).

Implementing a language this way would just require building a parser and a compiler which handles the program's control flow, since the language's data model and libraries would be handled by the API.

Our second approach consists in implementing Python's data model and functions in Racket. This leads to a greater effort in order to implement all of Python's standard library, but allows for a better integration with Racket code, and a better performance, currently standing at about 4 times slower than CPython.

We will be following our second approach, but we may offer support for accessing third-party libraries and unimplemented standard library modules using the features from our first approach.

Some of Python's common expressions and control flow statements have been already implemented, allowing for the successful compilation of two examples: the Fibonacci sequence and an implementation of the sieve of Eratosthenes.

In the future, we plan on implementing the remaining Python features and work on the integration between Python and Racket code.

Acknowledgements. This work was partially supported by Portuguese national funds through FCT under contract Pest-OE/EEI/LA0021/2013 and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

References

1 David Beazley. Understanding the Python GIL. In *PyCON Python Conference*, Atlanta, Georgia, 2010.

2 David M Beazley. *Python Essential Reference*. Addison-Wesley Professional, 2009.

3 Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of functional programming*, 12(2):159–182, 2002.

4 Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Programming Languages: Implementations, Logics, and Programs*, pages 369–388. Springer Berlin Heidelberg, 1997.

5 Matthew Flatt and Robert Bruce Findler. *The Racket Guide*, 2013.

6 Jim Hugunin. IronPython: A fast Python implementation for .NET and Mono. In *PyCON 2004 International Python Conference*, volume 8, 2004.

7 Roberto Ierusalimschy. *Programming in lua*, chapter An Overview of the C API. Roberto Ierusalimschy, 2006.

8 Ironclad – Resolver Systems. `http://www.resolversystems.com/products/ironclad/`. [Online; retrieved on January 2014].

9 Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Munoz Soto, and Victor Ng. *The definitive guide to Jython*. Springer, 2010.

10 Differences between CPython and Jython. `http://jython.sourceforge.net/archive/21/docs/differences.html`. [Online; retrieved on December 2013].

11 José Lopes and António Leitão. Portable generative design for CAD applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pages 196–203, 2011.

12 Philippe Meunier and Daniel Silva. From Python to PLT Scheme. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming*, pages 24–29, 2003.

13 Microsoft Corporation. *IronPython .NET Integration documentation*. `http://ironpython.net/documentation/`. [Online; retrieved on January 2014].

14 Bruce Momjian. *PostgreSQL: introduction and concepts*, chapter C Language Interface (LIBPQ).

15 Benjamin Peterson. Pypy. *The Architecture of Open Source Applications*, 2:279–290.

16 PyPy compatibility. `http://pypy.org/compat.html`. [Online; retrieved on December 2013].

17 PyPy speed center. `http://speed.pypy.org/`. [Online; retrieved on December 2013].

18 Stefan Richthofer. JyNI – using native CPython-extensions in Jython. In *EuroSciPi 2013*, Brussels, Belgium, 2013.

19 Muriel Salvan. The Ruby C API – basics. `http://blog.x-aeon.com/2012/12/13/the-ruby-c-api-basics/`. [Online; retrieved on March 2014].

20 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. *ACM SIGPLAN Notices*, 46(6):132–141, 2011.

21 Peter Tröger. Python 2.5 virtual machine. `http://www.troeger.eu/files/teaching/pythonvm08.pdf`, April 2008. [Lecture at Blekinge Institute of Technology].

22 Guido van Rossum and Fred L. Drake. *Extending and embedding the Python interpreter*. Centrum voor Wiskunde en Informatica, 1995.

23 Guido van Rossum and Fred L. Drake. *An introduction to Python*. Network Theory Ltd., 2003.

24 Guido van Rossum and Fred L. Drake. *The Python Language Reference*. Python Software Foundation, 2010.

25 Guido van Rossum and Fred L. Drake Jr. *Python/C API reference manual*, chapter Thread State and the Global Interpreter Lock. Python Software Foundation, 2002.

26 Michael Widenius and David Axmark. *MySQL reference manual: documentation from the source*, chapter MySQL C API. O'Reilly Media, Inc., 2002.

Plagiarism Detection: A Tool Survey and Comparison

Vítor T. Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz

Centro de Ciências e Tecnologias da Computação (CCTC)
Departamento de Informática, Universidade do Minho
Gualtar, Portugal
{vtiagovm,danielamoraisfonte,pedrorangelhenriques,danieladacruz}@gmail.com

Abstract

We illustrate the state of the art in software plagiarism detection tools by comparing their features and testing them against a wide range of source codes. The source codes were edited according to several types of plagiarism to show the tools accuracy at detecting each type.

The decision to focus our research on plagiarism of programming languages is two fold: on one hand, it is a challenging case-study since programming languages impose a structured writing style; on the other hand, we are looking for the integration of such a tool in an Automatic-Grading System (AGS) developed to support teachers in the context of *Programming courses*.

Besides the systematic characterisation of the underlying problem domain, the tools were surveyed with the objective of identifying the most successful approach in order to design the aimed plugin for our AGS.

1998 ACM Subject Classification I.5.4 Applications, Text processing

Keywords and phrases software, plagiarism, detection, comparison, test

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.143

1 Introduction

Plagiarism is concerned with the use of work without crediting its author, including the cases where someone uses previous code. It overrides copyrights in all the areas from arts or literature to sciences, and it is from the old days a forensic subject. Plagiarism does not only affect creativity or commercial businesses but also has negative effects in the academic environment. If a student plagiarises, a teacher will be unable to properly grade his ability.

In an academic context, it is a problem that applies not only to text documents but to source-code as well. Very often students answer the teacher assessment questions, submitting plagiarised code. This is why teachers have a strong need to recognise plagiarism, even when students try to dissimulate it. However, with a large number of documents, this becomes a burdensome task that should be computer aided.

This paper is precisely concerned with this subject, discussing approaches and tools aimed at supporting people on the detection of source code files that can be plagiarised.

Due to the complexity of the problem itself, it is often hard to create software that accurately detects plagiarism, since there are many ways a programmer can alter a program without changing its functionality. However, there are many programs for that purpose.

Some of them are off-line tools, like **Sherlock** [13, 11], **YAP** [22, 14, 11], **Plaggie** [1, 11], **SIM** [10, 1, 11], **Marble** [11], and **CPD** [5] which, even though it was only made to detect copies, is still a useful tool. There are also online tools like **JPlag** [16, 14, 7, 1, 11, 15] and **MOSS** [17, 14, 7, 1, 11, 15], and even tool sets like **CodeSuite** [18].


© Vítor T. Martins, Daniela Fonte, Pedro Rangel Henriques, and Daniela da Cruz;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 143–158

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The objective of this paper is to introduce and discuss existing tools in order to compare their performance. As the tools that were analysed use distinct technological approaches, it is important to choose the best candidate as to build the envisaged tool for our Automatic Grading System (namely iQuimera, an AGS under construction in our research group – for details, please see [8]) upon it.

The paper is organised in five sections. The state-of-the-art concerning source code plagiarism detection tools is presented in Section 2. In that section, we start with some basic concepts and we briefly discuss the major methodological approaches supporting the tools. Then we define a criteria composed of eight characteristics that should be kept in mind when studying each tool. This criteria enable us to create and present a comparative table that allows a quick survey. After this, another table, comparing the performance of the tools during the experimental study conducted, is shown and discussed presenting an overview of the problem. Section 3 describes the experimental research done, focusing on the source code files that were carefully prepared to test the tools exhaustively. Section 4 is devoted to the experimentation itself. For that, the tools are introduced in alphabetical order, a short description is provided and the specific results for each test are presented in tables. The conclusion and future work are presented in Section 5.

2 State of the Art

There are several techniques for the detection of plagiarism in source code. Their objective is to stop unwarranted plagiarism of source code in academic and commercial environments.

If a student uses existing code, it must be in conformance with the teacher and the school rules. The student might have to build software from the ground up, instead of using existing source code or tools that could greatly reduce the effort required to produce it but, this will allow a teacher to properly grade the student according to his knowledge and effort.

If a company uses existing source code, it may be breaking copyright laws and be subjected to lawsuits because it does not have its owners consent.

This need led to the development of plagiarism detectors, this is, programs that take text files (natural language documents, programs or program components) as input and output a similarity measure that shows the likelihood of there being copied segments between them. These outputs will usually come as a percentage.

2.1 Background

To understand the method used to create the tools and the offered functionality, it is necessary to understand the basic concepts involved. The following is a list presenting relevant terms:

Token A word or group of characters, sometimes named *n-gram* where n denotes the number of characters per token (as seen in [23]). Since white-spaces and newlines are usually ignored, the input “while(true) {” could produce two 5-grams: “while(” and “true){”.

Tokenization The conversion of a natural text into a group of tokens, as defined above.

Hash A number computed from a sequence of characters, usually used to speed up comparisons. For instance, if we use the ASCII[1] values of each character we could turn the token “word” into 444 (119 + 111 + 114 + 100).

Hashing The conversion of a sequence of tokens into their respective hash numbers.

[1] American Standard Code for Information Interchange

Fingerprint A group of characteristics that identify a program, much like physical fingerprints are used to identify people. An example would be if we consider a fingerprint to be composed by 3 hashes that are multiples of 4 (as seen on [23]).

Matching Algorithm The algorithm used to compare a pair of hashes, which allows the detection of the similarity degree between them. The comparison is performed by verifying if the fingerprints of each file are close, according to a pre-defined threshold. An example is to use the sum of the differences (between hashes) as the matching algorithm and a value of 10 as the threshold. In which case, taking a pair of files with the fingerprints: [41,582,493] and [40,585,490], the program would match, as the sum of the differences is 9 (|41-40|+|582-585|+|493-490| = 1+3+3).

Structural information This is information from the structure of a programming language. An example would be the concept of comments, conditional controls, among others.

The implementation of these concepts is always specific to each tool; since this process is not usually explained, it can only be learnt by inspecting its source code.

2.2 Approaches

From our research, the following list was built detailing the several methodologies that were used.

- An attribute-based methodology, where metrics are computed from the source code and used for the comparison. A simple example would be: using the size of the source code (number of characters, words and lines) as an attribute to single out the source codes that had a very different size. This methodology was mentioned by [20].
- A token-based methodology, where the source code is tokenized and hashed, using those hashes to create the fingerprints. This methodology was used by [21] and by [17].
- A structure-based methodology, where the source code is abstracted to an Internal intermediate Representation (IR), for example, an AST[2] or a PDG[3]) and then this IR is used for the comparison. This enables an accurate comparison. This methodology was used by [2] and by [14].

These methodologies go from the least accuracy with high efficiency to the highest accuracy with low efficiency.

Some examples of the metrics that could be used in an attribute-based methodology would be: files size, number of variables, number of functions and number of classes, among others. These metrics will usually be insufficient as students will usually solve the same exercise, which would cause a lot of suspicion from the tool.

The token-based methodology came with an attempt to balance the accuracy and the efficiency, often using RKS-GST[4] [21] which is a modern string matching algorithm that uses tokenization and hashing. To improve the results even further some tools mix some structure dependent metrics and modifications such as removing comments (used by JPlag 4.3) or sorting the order of the functions (used by Marble 4.4).

The structure-based approach uses abstractions that maintain the structure of the source code. This makes the technique more dependent on the language but it will also make the detection immune to several types of plagiarism such as, switching identifier names, switching the positions of functions and others.

[2] Abstract Syntax Tree
[3] Program Dependency Graph
[4] Running Karp-Rabin matching and Greedy String Tiling

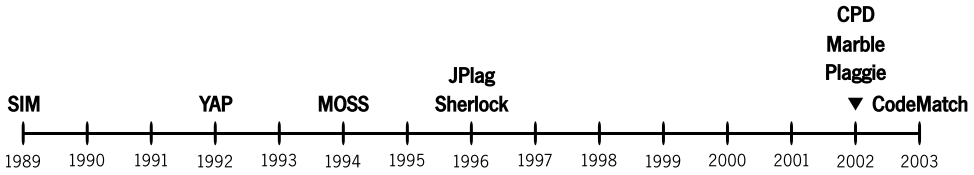


Figure 1 A timeline showing the years in which each tool was developed or referenced.

2.3 Existing Tools

We found several tools for the detection of software plagiarism throughout our research. Some of those tools were downloaded and tested. Other tools, like Plague [19, 22, 13] and GPlag [15, 3] were not taken into account since we could not access them.

A timeline was produced (see Figure 1), indicating the years when the tools were developed or, at least mentioned in an article. More details about the analysed tools are presented in Section 4.

2.3.1 Features Comparison

Inspired on [11], the following criteria were used to compare the tools:

1st) Supported languages: The languages supported by the tool.

2nd) Extendable: Whether an effort was made to make adding languages easier.

3rd) Quality of the results: If the results are descriptive enough to distinguish plagiary from false positives.

4th) Interface: If the tool has a GUI[5] or presents its results in a graphical manner.

5th) Exclusion of code: Whether the tool can ignore base code.

6th) Submission as groups of files: If the tool can consider a group of files as a submission.

7th) Local: If the tool can work without needing access to an external web service.

8th) Open source: If the source code was released under an open source license.

A table (see Table 1) was produced to report the criteria defined above. The values can be ✓(Yes), ✗(No), a ? in the cases where we could not ascertain if the feature is present, or a number in the case of supported languages.

Table 1 Comparison of the plagiarism detection tools.

Name	1	2	3	4	5	6	7	8
CodeMatch	36	✗	✓	✓	✓	✗	✓	✗
CPD	6	✓	✗	✓	?	?	✓	✓
JPlag	6	✗	✓	✓	✓	✓	✗	✗
Marble	5	✓	✓	✗	✓	?	✓	✗
MOSS	25	✗	✓	✓	✓	✓	✗	✗
Plaggie	1	?	?	✓	?	?	✓	✓
Sherlock	1	✗	✗	✗	✗	✗	✓	?
SIM	7	?	✓	✗	✗	✗	✓	?
YAP	5	?	✓	✗	✓	?	✓	✗

[5] Graphical User Interface

We can observe that both the CodeMatch and the MOSS tools support several languages, which makes them the best choices when analysing languages that the other tools do not support. However, others like CPD or Marble are easily extendable to cope with more languages. Note that if the tools support *natural language*, they can detect plagiarism between any text document but will not take advantage of any structural information.

Overall, we found that GUIs are unnecessary to give detailed output, so long as the tool can produce descriptive results that indicate the exact lines of code that caused the suspicion. This is observable with the use of Marble, SIM and YAP tools as they do not offer GUIs but still have descriptive results. On the other hand, tools like Sherlock do not present enough output information.

On academic environments, both the 5^{th} and the 6^{th} criteria are very important as they allow teachers to filter unwanted source code from being used in the matches. This means that tools like JPlag and MOSS allow for the proper filtering of the input source code.

As said in the introduction, the 7^{th} criteria reveals that JPlag and MOSS are the only tools dependent on online services as they are web tools.

In what concerns the availability of tools on open licenses, only CPD and Plaggie satisfy that requirement. For those wanting to reuse or adapt the tools, the 7^{th} and the 8^{th} criteria are important as the tool would need to have a license allowing its free use, and integration would benefit from having the tool distributed alongside the application.

2.3.2 Results Comparison

In order to test each tool and compare the results, we needed to start from an original and produce several files with different cases of plagiarism. To systematise the work, we felt the need to identify different types of plagiarism. So we built the types list below based on [13, 3, 6].

1st type of plagiarism This type of plagiarism is an exact copy of the original. It is the simplest type of plagiarism as no modifications were done to hide it.

2nd type of plagiarism This type of plagiarism is when the comments are changed or removed. It is the easiest modification as it has no risk of affecting the code. Note that most plagiarism detectors either ignore comments or have an option to ignore them and will not be diverted by this type of plagiarism. Of course, to do so the tools need the syntatic information on how comments are defined in the language.

3rd type of plagiarism This type of plagiarism is done by changing every identifier such as variable or function names.

4th type of plagiarism This type of plagiarism is when local variables are turned into global ones, and vice versa.

5th type of plagiarism This type of plagiarism is when the operands in comparisons and mathematical operations are changed (e.g. x < y to y >= x).

6th type of plagiarism This type of plagiarism is when variable types and control structures are replaced with equivalents. Care has to be taken as to avoid breaking the code functionality since the types will need to be converted to have the same behaviour.

7th type of plagiarism This type of plagiarism is when the order of statements (read lines) is switched. This is a common type of plagiarism as one only needs to make sure that the source code will keep the behaviour.

8th type of plagiarism This type of plagiarism is when groups of calls are turned into a function call and vice versa.

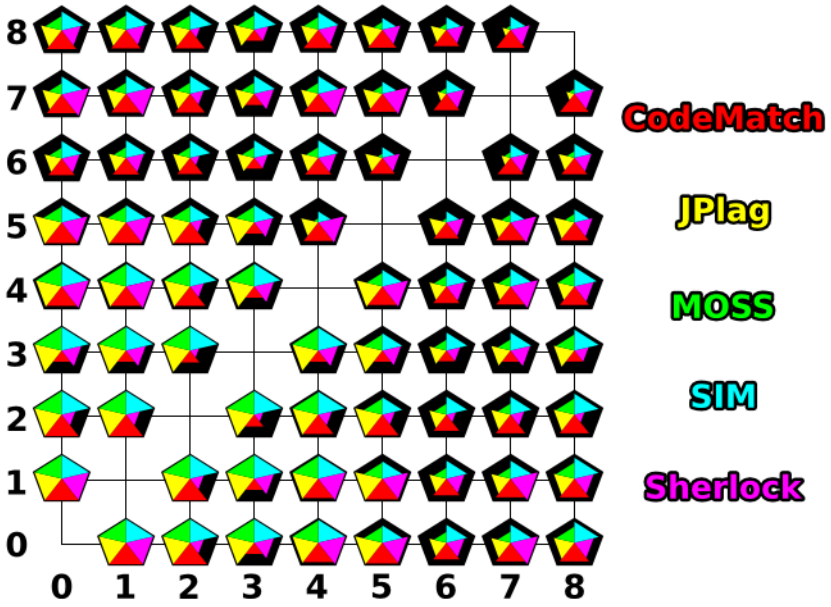


Figure 2 An overview of the results obtained.

The two types of plagiarism listed below were also referred in the cited lists. However, we will not consider them as they depend entirely on human supervision.

- Generating source-code by use of code-generating software.
- Making a program from an existing program in a different language.

Figure 2 shows the results that were obtained from comparing several files (as detailed in Section 3) and gives us an overview. The graphic gives a condensed overview of the results achieved, where each number represents a type of plagiarism that was used on the files that were compared. Each polygon has 5 triangles with each colour representing a tool. The size of the triangles represents the similarity degree between the files in the X and Y axis. While the graphic appears to be symmetric, the results for the MOSS and SIM tools are asymmetrical in a few cases (as detailed in Section 4).

We can see that the 1st type of plagiarism (exact copy) was easily detected by all the tools but the other types of plagiarism always had a big impact on some of the tools. Here are the most noticeable:

- MOSS has the most trouble with a lot of the types of plagiarism as it tries very hard to avoid false-positives, thus discarding a lot of information.
- Sherlock has a lot of trouble with the 2nd type of plagiarism (comments changed) as it compares the entire source codes without ignoring the comments.
- CodeMatch has the most trouble with the 3rd type of plagiarism (identifiers changed) as its algorithms must make some sort of match between identifiers.

We can see that most tools have a hard time when comparing from the 6th to the 8th types of plagiarism as they had a lot of small changes or movements of several blocks of source code. These are the types of plagiarism that would be best detected with the use of a structural methodology as, despite those changes to the structure, the context remains intact.

We can conclude that every tool has a weakness when it comes to certain types of plagiarism but notice that none of the results are 0%, the triangle representing the match is just so small that it is easy to miss. We also confirm that no type of plagiarism can fool all the tools at the same time.

3 Strategy for Testing the Tools

To demonstrate the accuracy of the existing tools detecting the different cases of plagiarism, two sample programs were collected and edited based on the types of plagiarism listed above. As the selected tools are prepared to work with Java or C programs, the exercises were written in both languages.

The following list of actions were performed to create the eight variants.

- to produce the first case, it is a straightforward file copy operation.
- to apply the second strategy, we simply removed the comments or changed their contents.
- to create a third variant, we changed most variable and function identifiers into reasonable replacements.
- to produce a copy of the forth type, we moved a group of local variables into a global scope and removed them from the function arguments. The type declaration was removed but the initialisation was kept.
- to obtain a file for the fifth type, we switched the orders of several comparisons and attributions. For example, we replaced $x+ = y - 5$ by $x+ = -5 + y$.
- to get a file for the sixth type, we replaced variable types such as *int* and *char* with *float* and *string*, along with the necessary modifications to have them work the same way.
- to create another copy according to the seventh type, we moved several statements, even some which broke the behaviour (write after read) to consider cases were the plagiarist was not careful.
- to produce a file exhibiting the eighth type, we applied this type of plagiarism in the easy (move entire functions) and the hard (move specific blocks of code in to a function) ways.

3.1 Source Code used in the Tests

In this subsection we give a brief description about the two sets of source files used in the tests. Each set includes both the original source file and 8 variants, where each one had a different type of plagiarism applied to it, according to the list above (see Section 2.3.2).

As all the files are based on the same original, from a theoretical point of view, every test should return a match of 100%.

Source code samples were written in both Java and C languages. As the samples are crucial for the assessment performed, we would like to show them here. Due to space limitations, it is impossible to include the original programs here (or their variants); so, we decided to make them available online, and just include in the paper a description of their objective and size.

3.1.1 Program 1: Calculator

The first set is based on a program which goal is to implement a simple calculator with the basic operations (addition, subtraction, division and multiplication) and a textual interface.

Only the Java version of the file was used to produce the results presented. The following metrics characterise the original file:

- **Number of lines**: 56
- **Number of functions**: 1
- **Number of variables**: 4

The Java source code files are available online at
`https://tinyurl.com/dopisiae/slate/source/calc_java.zip`,
as well as the C source code files, at
`https://tinyurl.com/dopisiae/slate/source/calc_c.zip`.

3.1.2 Program 2: 21 Matches

The objective was to make an interactive implementation of the 21 matches game where, starting with 21 matches, each player takes 1 to 4 matches until there are no more. The one getting the last match will lose the game. The game can be played against another player or against the computer.

Only the C version of the file was used to produce the results presented. The following metrics characterise the original file:

- **Number of lines**: 189
- **Number of functions**: 4
- **Number of variables**: 8

The C source code files are available online at
`https://tinyurl.com/dopisiae/slate/source/21m_c.zip`,
as well as the Java source code files, at
`https://tinyurl.com/dopisiae/slate/source/21m_java.zip`.

4 Tool Details and Test Results

Along Section 2, nine tools were identified (and ordered in a timeline) and they were compared: CodeMatch, CPD, JPlag, Marble, MOSS, Plaggie, Sherlock, SIM, and YAP. Aiming at providing a deeper knowledge about the tools explored, in this section they will be introduced with some more detail and the output they produced when experimented against the two test sets will be shown.

As described in SubSection 2.3.2, 8 types of plagiarism were considered. Two test sets were created containing an original program file and 8 variant files, modified according to the type of plagiarism.

We have produced tables from the results of each tool when matching each of the files with the other 8. This gives us a look at the accuracy of their metrics for each type of plagiarism. As stated earlier, from a theoretical point of view every test should return a match of 100% so, the higher the results the higher the tools accuracy.

Note that, due to the 16 page constraint, we only show the full result tables for the CodeMatch tool and trimmed the remaining result tables to their first line.

To compare the results pertaining to tables of the same tool, we used the following algorithm: Given two tables, we subtract each value in the second table with the value in the first table and add their results. As an example, consider the tables [4,3] and [2,1]. In this case the formula is: (2-4)+(1-3), which is equal to -4. This allows us to see whether the results increased or decreased from the first table to the second. We will refer to this metric as the difference metric (DM). If there are several lines of results, we will use the average which is calculated by dividing the sum of every line DM by the number of lines.

Table 2 The results produced the CodeMatch tool for the Calculator set.

	0	1	2	3	4	5	6	7	8
0	100	100	95	70	95	95	76	90	86
1	100	100	95	70	95	95	76	90	86
2	95	95	100	63	90	90	69	84	80
3	70	70	63	100	70	70	63	70	67
4	95	95	90	70	100	91	75	89	85
5	95	95	90	70	91	100	72	84	86
6	76	76	69	63	75	72	100	76	71
7	90	90	84	70	89	84	76	100	82
8	86	86	60	67	85	86	71	82	100

Note that, the DMs we show are from the full results. You can find all the results as well as the DM calculations on the following link: `https://tinyurl.com/dopisiae/slate/index.html`.

Since most tools only give a metric for each pair of files, most tables are symmetrical. For the asymmetrical tables, the percentage must be read as the percentage of the line file that matches the column file.

4.1 CodeMatch

CodeMatch [11] is a part of CodeSuite [18] and detects plagiarism in source code by using algorithms to match statements, strings, instruction sequences and identifiers. CodeSuite is a commercial tool that was made by SAFE[6], which is housed at `http://www.safe-corp.biz/index.htm`. It features several tools to measure and analyse source or executable code.

This tool is only available as an executable file (binary file) and only runs under Windows.

CodeMatch supports the following languages: ABAP, ASM-6502, ASM-65C02, ASM-65816, ASM-M68k, BASIC, C, C++, C#, COBOL, Delphi, Flash ActionScript, Fortran, FoxPro, Go, Java, JavaScript, LISP, LotusScript, MASM, MATLAB, Pascal, Perl, PHP, PL/M, PowerBuilder, Prolog, Python, RealBasic, Ruby, Scala, SQL, TCL, Verilog, VHDL and Visual Basic.

4.1.1 Results for the Calculator Code

CodeMatch returned good results (see Table 2) for the Calculator, Java source codes (see Section 3.1.1). Note that the files were compared with themselves since CodeMatch compares the files in two folders and the same folder was used. This is a moot point as those cases got 100% matches.

4.1.2 Results for the 21 Matches Code

The results (see Table 3) for the 21 Matches source code were similar to the previous (DM=-15.56), except for the 3rd type of plagiarism (Identifiers changed). This is probably due to the fact that the 21 Matches source codes have more identifiers than the Calculator ones.

[6] Software Analysis and Forensic Engineering

Table 3 The results produced by CodeMatch tool for the 21 Matches Code.

	0	1	2	3	4	5	6	7	8
0	100	100	100	54	90	96	78	87	87
1	100	100	100	54	90	96	78	87	87
2	100	100	100	57	90	96	76	87	87
3	54	54	57	100	54	54	52	54	53
4	90	90	90	54	100	86	78	85	86
5	96	96	96	54	86	100	75	84	84
6	78	78	76	52	78	75	100	77	77
7	87	87	87	54	85	84	77	100	85
8	87	87	87	53	86	84	77	85	100

Table 4 The results produced by JPlag tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100	100	100	84.7	100	40.4	43.4	39.6

4.2 CPD

CPD [5] is a similarity detector that is part of PMD, a source code analyser that finds inefficient or redundant code, and is housed at `http://pmd.sourceforge.net/`. It uses the RKS-GST algorithm to find similar code.

It supports the following languages: C, C++, C#, Java, Ecmascript, Fortran, Java, JSP, PHP and Ruby.

Since CPD only returns detailed results without a match percentage metric, we could not produce the tables.

4.3 JPlag

JPlag [16, 14, 7, 1, 11, 15] takes the language structure into consideration, as opposed to just comparing the bytes in the files. This makes it good for detecting plagiarism despite the attempts of disguising it.

It supports the following languages: C, C++, C#, Java, Scheme and Natural language.

JPlag gives us results organised by the average and maximum similarities, without repeated values. It produces an HTML file presenting the result and allows the user to view the detailed comparison of what code is suspected to be plagiarism.

4.3.1 Results for the Calculator Code

As we can see (in Table 4), the results were good since there where a lot of exact matches (100%), showing that JPlag was impervious to several types of plagiarism.

4.3.2 Results for the 21 Matches Code

JPlag was better (DM=114.49) at detecting the plagiarism for this set of source codes since the results (see Table 5) are similar in relation to the exact matches and got better matches on the harder cases like the 6^{th}, 7^{th} and 8^{th} types of plagiarism.

Table 5 The results produced by JPlag tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	100	100	100	93.7	100	62.8	70.5	75.6

Table 6 The results produced by Marble tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100 U	100 U	100 U	94 U	100 S	77 S	81 S	66 U

4.4 Marble

Marble [11], which is described in `http://www.cs.uu.nl/research/techreps/aut/jur.html`, facilitates the addition of languages by using code normalisers to make tokens that are independent of the language. A RKS-GST algorithm is then used to detect similarity among those tokens.

It supports the following languages: Java, C#, Perl, PHP and XSLT.

The results are presented through a suspects.nf file which has several lines in the following structure: "echo *M1 S1 S2 M2 U/S* && edit *File 1* && edit *File 2*". The *M1* and *M2* values indicate the match percentages, *S1* and *S2* give us the size of the matches and the *U/S* flag indicates if the largest percentage was found before (U) or after (S) ordering the methods.

Note that only the Calculator results are available since Marble does not support the C language.

4.4.1 Results for the Calculator Code

As expected, a few source codes accuse the movement of methods (have an S flag). This is verified in the 5^{th}, 6^{th} and 7^{th} types of plagiarism that had operations, variables and statements moved, respectively.

4.5 MOSS

MOSS [17, 14, 7, 1, 11, 15] automatically detects similarity between programs with a main focus on detecting plagiarism in several languages that are used in programming classes. It is provided as an Internet service that presents the results in HTML pages, reporting the similarities found, as well as the code responsible. It can ignore base code that was provided to the students and focuses on discarding information while retaining the critical parts, in order to avoid false positives.

It supports the following languages: A8086 Assembly, Ada, C, C++, C#, Fortran, Haskell, HCL2, Java, Javascript, Lisp, Matlab, ML, MIPS Assembly, Modula2, Pascal, Perl, Python, Scheme, Spice, TCL, Verilog, VHDL and Visual Basic.

This tool gives us the number of lines matching between each pair of files and calculates the match percentages by dividing it by the number of lines in the file. The algorithms used were made to avoid false positives at the cost of getting lower percentages.

Its HTML interface was produced by the author of JPlag (Guido Malpohl) and, the results come in both an overview and detailed forms.

Table 7 The results produced by MOSS tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	99	99	99	85	99	30	55	36

Table 8 The results produced by MOSS tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	99	99	99	94	67	44	45	67

4.5.1 Results for the Calculator Code

As we can see in Table 7, the results are reasonable but the strategies used to avoid false positives decreased the matches (ex.: 100% to 99%). We can also notice that some types of plagiarism, namely the 6^{th}, 7^{th} and 8^{th} had low matches due to their high complexity.

4.5.2 Results for the 21 Matches Code

As expected the increase in the size of the source codes translated into the variance of the results (see Table 8) albeit the increases (DM=4.67) were rather small when compared to other tools.

4.6 Plaggie

Plaggie [1, 11], which is housed at `http://www.cs.hut.fi/Software/Plaggie/`, detects plagiarism in Java programming exercises. It was made for an academic environment where there is a need to ignore base code. It only supports the Java language. As we were unable to compile the tool, we can not present its results.

4.7 Sherlock

Sherlock [13, 11], which is housed at `http://sydney.edu.au/engineering/it/~scilect/sherlock/`, detects plagiarism in documents through the comparison of fingerprints which, as stated in the website, are a sequence of *digital signatures*. Those digital signatures are simply a hash of (3, by default) words.

It allows for control over the *threshold* which hides the percentages with lower values, the number of words per *digital signature* and the *granularity* of the comparison by use of the respective arguments: -t, -n and -z. It supports the following language: Natural language.

Sherlock is an interesting case as its results are a list of sentences in a "*File 1* and *File 2*: *Match%*" format. This format does not help the user find the highest matches, it simply makes the results easier to post-process. Results were produced with all the combinations of the settings from -n 1 to 4 and -z 0 to 5.

4.7.1 Results for the Calculator Code

With the default settings (equivalent to -n 3 -z 4), we can see that some of the results (see Table 9) are good, mainly for the 1st (Unaltered copy), 4th (Scope changed) and 7th (Statements order switched) types of plagiarism. Seeing as Sherlock matches natural language, the similarity values will subside due to textual changes and be mostly unaffected by movements.

Table 9 The results produced by Sherlock tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100	62	44	100	62	25	100	45

Table 10 The results produced by Sherlock tool for the Calculator code with the -n 2 argument.

	1	2	3	4	5	6	7	8
0	100	55	72	100	100	72	100	80

To demonstrate the effect of tweaking the -n and -z parameters, two more tables, Tables 10 and 11, are presented exhibiting the results of Sherlock when invoked with the arguments "-n 2" and "-z 3" respectively.

Table 10 shows us that using the -n parameter can greatly improve (DM=198.78) the results. This argument changes the number of words per digital signatures, Meaning that with a smaller value, Sherlock will find plagiarism in smaller sections of source code. This translates into better matches in small changes and source code movements, but worse matches on longer modifications (as seen for the 2^{nd} type of plagiarism).

The -z argument changes Sherlocks "granularity," a parameter that serves to discard part of the hash values. As seen on Table 11, Sherlock was more sensitive to changes which resulted in an overall decrease (DM=-36).

Overall, we can see that Sherlock is a very specific tool and that further studies would have to be made in order to ascertain the adequate parameters to use for specific situations.

4.7.2 Results for the 21 Matches Code

For the 21 Matches source codes, the results (see Table 12) seem more accurate (DM=146.44) than the results for the Calculator source codes (see Table 9). This was likely due to the increased size of the source codes.

4.8 SIM

SIM [10, 1, 11], which is housed at `http://dickgrune.com/Programs/similarity_tester/`, is an efficient plagiarism detection tool which has a command line interface, like the Sherlock tool. It uses a custom algorithm to find the longest common sub-string, which is order-insensitive.

It supports the following languages: C, Java, Pascal, Modula-2, Lisp, Miranda and Natural language.

SIM is an interesting tool. Despite not having a GUI, its results are quite detailed. The -P argument can be used to report the results in a "*File 1* consists for *Match%* of *File 2* material" format, giving a quick rundown of the results. It has a few arguments that change its behaviour; however, unlike Sherlock, its default values seem to work well for most cases of plagiary.

4.8.1 Results for the Calculator Code

This tool is quite good with results showing exact matches for most types of plagiarism.

Table 11 The results produced by Sherlock tool for the Calculator source code with the -z 3 argument.

	1	2	3	4	5	6	7	8
0	100	31	41	88	76	42	88	41

Table 12 The results produced by Sherlock tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	97	32	62	90	85	89	93	88

4.8.2 Results for the 21 Matches Code

On Table 14, we can see that it was harder to detect plagiarism in bigger source codes (DM=-120.33), although it was beneficial in a specific case, the 4^{th} type of plagiarism.

4.9 YAP

YAP[7] [22, 14, 11], which is housed at `http://www.pam1.bcs.uwa.edu.au/~michaelw/YAP.html`, is a tool that currently has 3 implementations, each using a fingerprinting methodology with different algorithms. The implementations have a tokenizing and a similarity checking phase and just change the second phase. The tool itself is an extension of Plague.

YAP1 The initial implementation was done as a Bourne-shell script and uses a lot of shell utilities such as diff, thus being inefficient. It was presented by Wise [20].

YAP2 The second implementation was made as a Perl script and uses an external C implementation of the Heckel [12] algorithm.

YAP3 The third and latest iteration uses the RKS-GST methodology.

The tools themselves are separate from the tokenizers but packaged together. These tools are said to be viable at the detection of plagiarism on natural language, aside from their supported languages.

It supports the following languages: Pascal, C and LISP.

We were unable to produce understandable results.

5 Conclusions

In this paper, plagiarism in software was introduced and characterised. Our purpose is to build a tool that will detect source code plagiarism in academic environments, having in mind that detecting every case is implausible. We hope that it will be able to produce accurate results when faced with complex types of plagiarism, like switching statements and changing variable types, and will be immune to the simpler types of plagiarism, such as modifying comments and switching identifier names. One of the most difficult and challenging problem in this context is the ability to distinguish between plagiarism and coincidence and we are aware of the danger of false positives [4, 9]. However this is a topic that deserves further investigation. To plan and support our working decisions, we devoted some months to the research of the existing methodologies and tools, as related in this paper. The comparative study involving nine tools that were available for download and were ready to run, gave us

[7] Yet Another Plague

Table 13 The results produced by SIM tool for the Calculator Code.

	1	2	3	4	5	6	7	8
0	100	100	100	44	100	91	99	100

Table 14 The results produced by SIM tool for the 21 Matches Code.

	1	2	3	4	5	6	7	8
0	100	100	100	97	72	75	75	88

the motivation to build a tool that adopts a structure based methodology with the AST as the abstract structure. Although the existing tools do not detect accurately all the types of plagiary identified, we found that most of them were easy to use with their default options but Marble and YAP have special file structure requirements and Plaggie has to be compiled. We found JPlags interface to be intuitive and offering a wide variety of options but, as it is not local, we recommend SIM as a viable alternative. We would like to point out that CodeSuite package offers a good number of tools for the detection of source code theft. The next task will be to develop the prototype tool based on the chosen methodology, and perform its evaluation in real case studies.

Acknowledgements. We give thanks to Dr. Jurriaan Hage for the copy of the Marble tool and Bob Zeidman for the CodeSuite licenses.

This work is funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

References

1 Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In *Proceedings of 6th Koli Calling Intern. Conference on Comp. Ed. Research*, pages 141–142, New York, USA, 2006. ACM.

2 I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of IEEE ICSM 1998*, pages 368–377, 1998.

3 Andrés M. Bejarano, Lucy E. García, and Eduardo E. Zurek. Detection of source code similitude in academic environments. *Computer Applic. in Engineering Education*, 2013.

4 Miranda Chong and Lucia Specia. Linguistic and statistical traits characterising plagiarism. In *COLING 2012*, pages 195–204, 2012.

5 Tom Copeland. Detecting duplicate code with PMD's CPD, 2003.

6 G. Cosma and M. Joy. Towards a definition of source-code plagiarism. *IEEE Trans. on Educ.*, 51(2):195–200, May 2008.

7 Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. Code comparison system based on abstract syntax tree. In *3rd IEEE IC-BNMT*, pages 668–673, 2010.

8 Daniela Fonte, Ismael Vilas Boas, Daniela da Cruz, Alda Lopes Gançarski, and Pedro Rangel Henriques. Program Analysis and Evaluation using Quimera. In *ICEIS'2012*, pages 209–219. INSTICC, June 2012.

9 Cristian Grozea and Marius Popescu. Who's the thief? automatic detection of the direction of plagiarism. In *In CICLing*, pages 700–710, 2010.

10 Dick Grune and Matty Huntjens. Het detecteren van kopieën bij informatica-practica. *Informatie*, 31(11):864–867, 1989.

11 Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. *Utrecht University. Utrecht, The Netherlands*, page 28, 2010.

12 Paul Heckel. A technique for isolating differences between files. *Communications of the ACM*, 21(4):264–268, 1978.

13 Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Trans. on Educ.*, 42(2):129–133, May 1999.

14 Xiao Li and Xiao Jing Zhong. The source code plagiarism detection using AST. In *International Symposium IPTC*, pages 406–408, 2010.

15 Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD'06*, pages 872–881. ACM Press, 2006.

16 Lutz Prechelt, Guido Malpohl, and Michael Phlippsen. JPlag: Finding plagiarisms among a set of programs. Technical report, Fakultät für Informatik, Universität Karlsruhe, 2000.

17 Saul Schleimer. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD*, pages 76–85. ACM Press, 2003.

18 Ilana Shay, Nikolaus Baer, and Robert Zeidman. Measuring whitespace patterns as an indication of plagiarism. In *Proceedings of the ADFSL Conference*, pages 63–72, 2010.

19 Geoff Whale. Software metrics and plagiarism detection. *Journal of Systems and Software*, 13(2):131–138, 1990. Special Issue on Using Software Metrics.

20 Michael J. Wise. Detection of similarities in student programs: YAP'ing may be preferable to plague'ing. In *ACM SIGCSE Bulletin*, volume 24, pages 268–271. ACM, 1992.

21 Michael J. Wise. *Running Karp-Rabin matching and greedy string tiling.* Basser Dept. of Computer Science, University of Sydney, Sydney, 1993.

22 Michael J. Wise. YAP3: Improved detection of similarities in computer program and other texts. In *SIGCSEB: SIGCSE Bulletin*, pages 130–134. ACM Press, 1996.

23 Robert M. Zeidman. Software tool for detecting plagiarism in computer source code, 2003.

Target Code Selection by Tilling AST with the Use of Tree Pattern Pushdown Automaton

Jan Janoušek and Jaroslav Málek

Department of Theoretical Computer Science
Faculty of Information Technologies
Czech Technical University in Prague
Technická 9, 160 00 Prague 6, Czech Republic
Jan.Janousek@fit.cvut.cz

Abstract

A new and simple method for target code selection by tilling an abstract syntax tree is presented. As it is usual, tree patterns corresponding to target machine instructions are matched in the abstract syntax tree. Matching tree patterns is performed with the use of tree pattern pushdown automaton, which accepts all tree patterns matching the abstract syntax tree in the linear postfix bar notation and represents a full index of the abstract syntax tree for tree patterns. The use of the index allows to match patterns quickly, in time depending on the size of patterns and not depending on the size of the tree. The selection of a particular target instruction corresponds to a modification of the abstract syntax tree and also a corresponding incremental modification of the index is performed. A reference to a fully functional prototype is provided.

1998 ACM Subject Classification D.3.4 Processors: Software, Programming languages, Code Generation, Compilers

Keywords and phrases code generation, abstract syntax tree, indexing, tree pattern matching, pushdown automata

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.159

1 Introduction

A compiler backend transforms an intermediate code representation (IR), which is produced by a compiler frontend, to a target code [1]. One of the most used type of the IR is an abstract syntax tree (AST). The task of the target code selection from the AST can be performed by tilling the AST by tree patterns that correspond to target machine code instructions. This task is usually ambiguous and therefore often the best possible such tilling is to be found. For this reason a cost function of the particular machine code instructions is used so that the tilling with a minimal overall cost would be computed.

During the tilling of the AST tree pattern matching methods are used. For many linear notations of trees it holds that the linear notation of a subtree is a substring of the linear notation of the tree [12]. A tree pattern is a tree whose leaves can be labelled by a special symbol S, which serves as a placeholder for any subtree. A tree pattern in a linear notation corresponds to a substring of the linear notation of the tree, where the symbols S are replaced with the linear notations of subtrees. Therefore, tree pattern matching is analogous to the problem of matching patterns with specific gaps. See [2, 3, 8, 9, 12] for the basic tree pattern matching methods.

Since the problem of tilling the AST is generally ambiguous and NP-hard, many heuristics and methods are used for a "good" selection of particular instructions. Some of the methods perform one pass of the AST, some others perform more passes of the AST. From another


© Jan Janoušek and Jaroslav Málek;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 159–165

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

point of view, various models of computation are used for the description of target code selection methods. A code selection method based on deterministic finite tree automata can be found in [4], where the cost function is computed by an additional semantic evaluation. On the other hand, [7, 10, 13] describe the code selection methods based on deterministic pushdown automata performing the tree pattern matching, where the tree patterns are represented by rules of a context-free grammar, and in this way generally ambiguous and non-LR(0) context-free grammars are created. Consequently, the LR(0) parsers for those grammars contain conflicts. In [7] these conflicts are resolved by some heuristics; in [10, 13] a special construction of a deterministic parser is used, which corresponds to a determinization of the above–mentioned LR(0) parser with the conflicts. We mention also a family of tools BURG, IBURG, etc. (see [5, 6] for example), which use another model of computation, so-called tree rewriting systems, for the tree pattern matching in the code selection problem.

In this paper a new and simple method for target code selection by tilling an abstract syntax tree is presented. As it is usual, tree patterns corresponding to target machine instructions are matched in the AST. Matching tree patterns is performed with the use of tree pattern pushdown automaton, which accepts all tree patterns matching the abstract syntax tree in the linear bar postfix notation and represents a full index of the abstract syntax tree for tree patterns. Tree pattern pushdown automaton is described in details in [12]. The use of the index allows to match patterns quickly, in time depending on the size of patterns and not depending on the size of the tree. The selection of a particular target instruction corresponds to a modification of the AST and also a corresponding incremental modification of the tree pattern pushdown automaton is performed. Given an AST of size n, the number of distinct tree patterns which match the AST is $\mathcal{O}(2^n)$. For the sake of a better space complexity we use the nondeterministic version of the tree pattern pushdown automata. This is feasible and reasonable because tree patterns that correspond to target instructions are not typically large, and the space complexity of the nondeterministic tree pattern pushdown automaton is $\mathcal{O}(n)$. Experimental results of a fully functional prototype can be found in [11].

The new contributions of this paper are:

- The incremental modification of the tree pattern pushdown automaton for a specific modification of the AST.
- A simple use of the tree pattern pushdown automaton for the selection of target code, using modification mentioned in the previous item.

The rest of the paper is organised as follows. The tree pattern pushdown automaton, which is constructed for the AST, and its incremental modification are described in the second section. The third section describes the selection of a target instruction by tilling the AST. Experimental results of a fully functional prototype are presented in the fourth section. The last section is the conclusion.

2 Tree Pattern Pushdown Automaton and its Incremental Modification

Tree pattern pushdown automaton and its modification are demonstrated on a running example. Fig. 1 shows a tree t_1. Its linear postfix bar notation [12], for which the automaton is constructed, is $bar(t_1) = [[a[[b[aab$. The nondeterministic tree pattern pushdown automaton is illustrated in Fig. 2. This automaton accepts all tree patterns which match the tree in the postfix bar notation. In all figures of pushdown automata in this paper, the edges, which

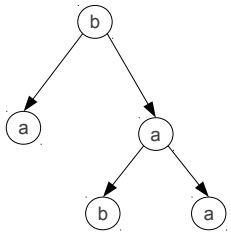


Figure 1 Tree t_1.

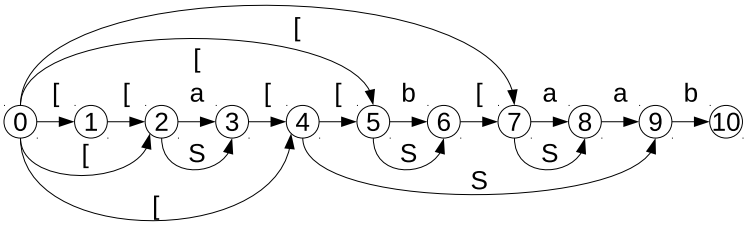


Figure 2 Tree pattern pushdown automaton for tree t_1.

represent transitions, are labelled only by an input symbol that is read by the transition and the pushdown operations are not illustrated. The pushdown operations ensure that a correct (sub)tree is processed in its linear notation. The automaton is input-driven, which means that each pushdown operation is unambiguously given by the input symbol. In our case, reading bar symbol [pushes one symbol onto the pushdown store, whereas reading other symbols pops one symbol from the pushdown store [12].

An example of tilling the AST by tree pattern [[b[aa in the postfix bar notation with a modification to tree pattern [[ed in the postfix bar notation is demonstrated in Figs. 3, 4 and 5. The formal algorithm of this modification can be found in [11].

3 Target Code Selection by Tilling AST

As it is mentioned in the Introduction, the tilling AST by target instructions is an ambiguous problem in general. The tree pattern pushdown automaton is used for computing a set of possible instructions that can be selected. In [11] the instructions that are selected are simply fixed from the set of all possibilities according to a cost function. We note that also other heuristics for the selection from the set of possibilities can be considered.

Fig. 6 shows the AST for statement x[3] = temp*(temp+10).

The AST in Fig. 7 is after the selection the instruction `store`. The tree pattern pushdown automaton is modified accordingly.

The AST in Fig. 8 is after the selection the instruction `add`. The tree pattern pushdown automaton is modified accordingly.

The next instruction to be selected is `load`, which occurs twice in the AST. Both occurrences are selected and the result is shown in Fig. 9.

The tilled AST is illustrated in Fig. 10. The tilling generates the following sequence of instructions:

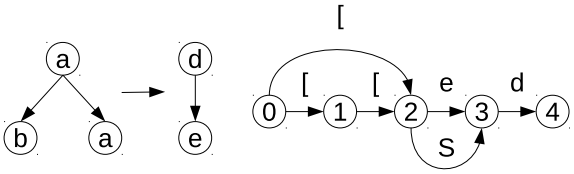


Figure 3 The selection of tree pattern with the modification of the AST and the corresponding part of the pushdown automaton.

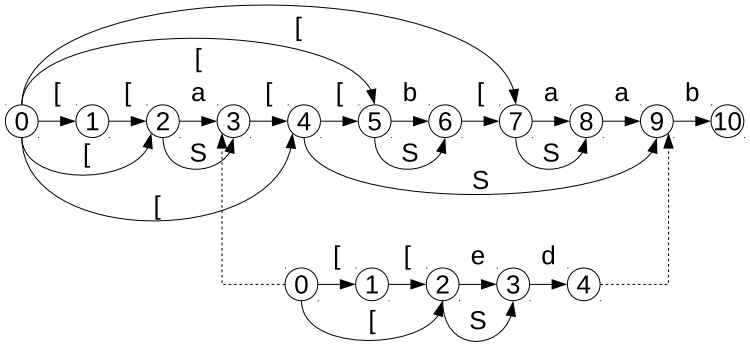


Figure 4 Part of the pushdown automaton to be modified.

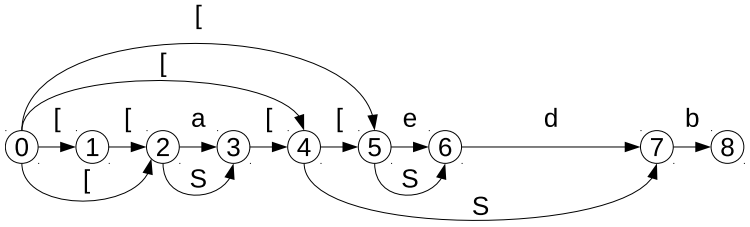


Figure 5 The resulting pushdown automaton after the modification.

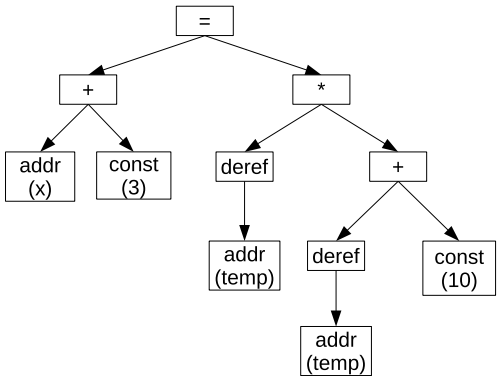


Figure 6 AST for statement x[3] = temp*(temp+10).

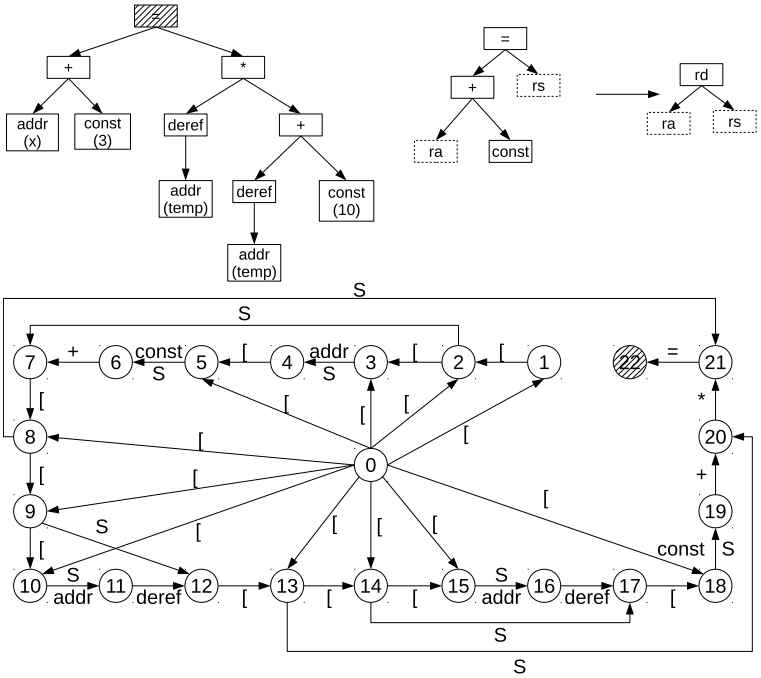


Figure 7 Tilling by tree pattern for instruction store.

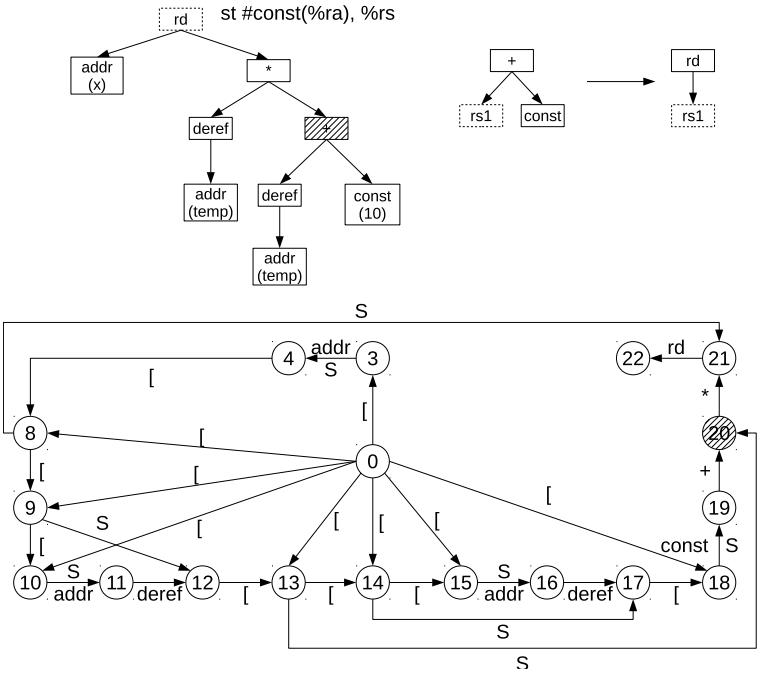


Figure 8 Tilling by tree pattern for instruction add.

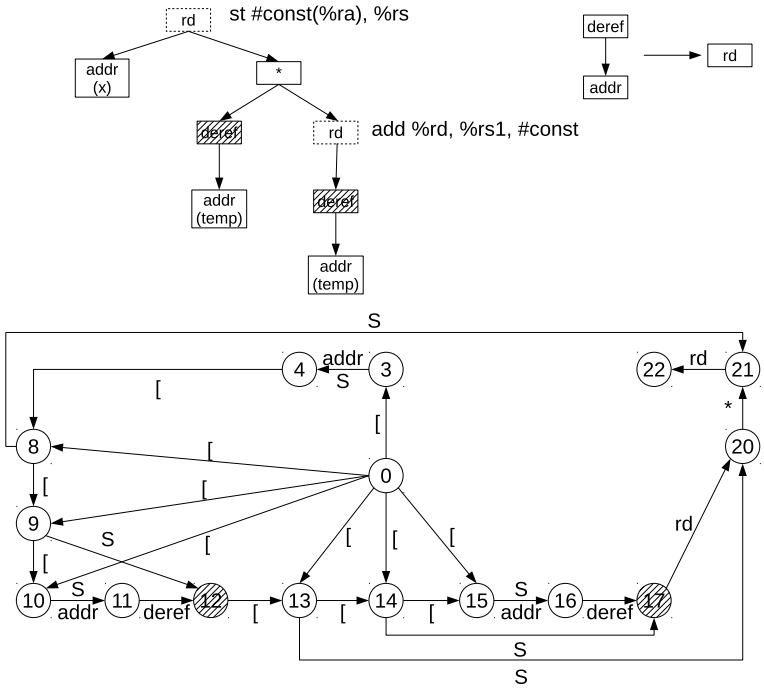


Figure 9 Tilling by tree pattern for instruction load.

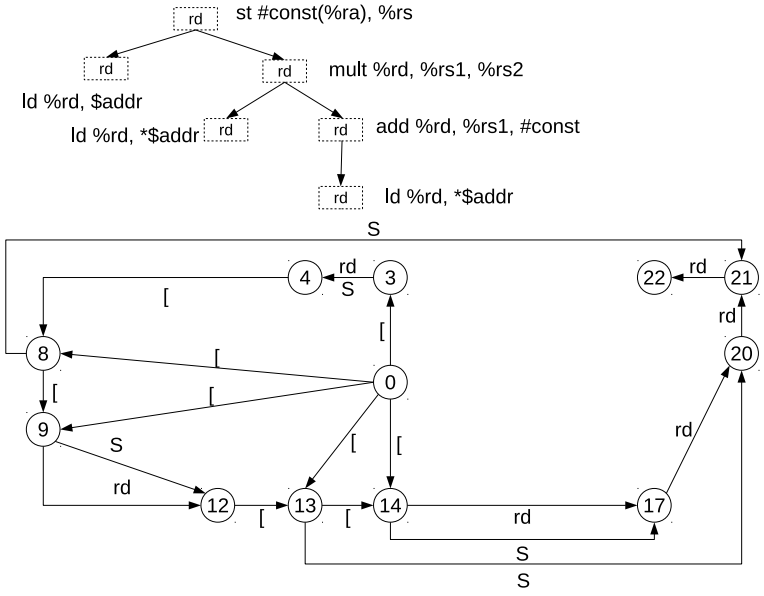


Figure 10 Tilled AST.

```
ld %rd, $addr
ld %rd, *$addr
ld %rd, *$addr
add %rd, %rs1, #const
mult %rd, %rs1, %rs2
st #const(%ra), %rs
```

4 Conclusion

A new and simple method of the code generation by tilling the AST with the use and incremental modification of the tree pattern pushdown automaton, which represents a full index of the AST for tree patterns, has been presented. More details, a prototype and its experimental results can be found in [11].

Acknowledgments. This work was partially supported by GAČR Grant No. GA13–03253S.

References

1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

2 P. Bille. *Pattern Matching in Trees and Strings*. PhD thesis, FIT University of Copenhagen, Copenhagen, 2008.

3 David R. Chase. An improvement to bottom-up tree pattern matching. In *ACM Symp. POPL*, pages 168–177, 1987.

4 Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree automata for code selection. *Acta Inf.*, 31(8):741–760, 1994.

5 Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *LOPLAS*, 1(3):213–226, 1992.

6 Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. Burg: fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, 1992.

7 R. Steven Glanville and Susan L. Graham. A new method for compiler code generation. In *POPL*, pages 231–240, 1978.

8 Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.

9 Jan Lahoda and Jan Žďárek. Simple tree pattern matching for trees in the prefix bar notation. *Discrete Applied Mathematics*, 163, Part 3:343–351, January 2014.

10 Maya Madhavan, Priti Shankar, Siddhartha Rai, and U. Ramakrishna. Extending graham-glanville techniques for optimal code generation. *ACM Trans. Program. Lang. Syst.*, 22(6):973–1001, 2000.

11 J. Málek. Code generation with the use of an index of ast. *FIT, Czech Technical University in Prague, MSc thesis*, In Czech, 2014.

12 Bořivoj Melichar, Jan Janoušek, and Tomáš Flouri. Arbology: trees and pushdown automata. *Kybernetika*, 48, No.3:402–428, 2012.

13 Priti Shankar, Amitrajan Gantait, A. R. Yuvaraj, and Maya Madhavan. A new algorithm for linear regular tree pattern matching. *Theor. Comput. Sci.*, 242(1-2):125–142, 2000.

Part V

Semantics in Natural Language Processing

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Assigning Polarity Automatically to the Synsets of a Wordnet-like Resource

Hugo Gonçalo Oliveira[1], António Paulo Santos[2], and Paulo Gomes[3]

1 **CISUC, Department of Informatics Engineering**
University of Coimbra, Portugal
hroliv@dei.uc.pt
2 **GECAD, Institute of Engineering**
Polytechnic of Porto, Portugal
pgsa@isep.ipp.pt
3 **CISUC, Department of Informatics Engineering**
University of Coimbra, Portugal
pgomes@dei.uc.pt

Abstract

This article describes work towards the automatic creation of a conceptual polarity lexicon for Portuguese. For this purpose, we take advantage of a polarity lexicon based on single lemmas to assign polarities to the synsets of a wordnet-like resource. We assume that each synset has the polarity of the majority of its lemmas, given by the initial lexicon. After that, polarity is propagated to other synsets, through different types of semantic relations. The relation types used were selected after manual evaluation. The main result of this work is a lexicon with more than 10,000 synsets with an assigned polarity, with accuracy of 70% or 79%, depending on the human evaluator. For Portuguese, this is the first synset-based polarity lexicon we are aware of. In addition to this contribution, the presented approach can be applied to create similar resources for other languages.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases sentiment analysis, polarity, lexicon, wordnet, Portuguese

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.169

1 Introduction

The World Wide Web has become an important resource for supporting decision making. People often turn to this infrastructure to search for pieces of information that, hopefully, will help them choose their next smartphone, a country to visit during the holidays or, sometimes, even to decide on what party to vote in the next election. But it is a well-known fact that a rational decision is highly limited both by the available information and by the available time. So, as the Web keeps growing, the aforementioned procedure is losing efficiency.

Sentiment analysis (SA), or opinion mining (see [20] for an extensive survey), deals with the computational treatment of opinions and sentiments in order to provide more efficient decision making. Opinions given by an author towards a subject are typically classified according to their polarity, which might be positive, negative and, sometimes, just neutral. Many approaches to this topic rely on existing linguistic resources to predict the polarity of a given piece of information. One of the main research lines in SA is thus the creation of adequate sentiment resources, including annotated corpora and polarity lexicons.

Most polarity lexicons are structured in word lemmas, identified by their orthographical form, and the typical polarity they express. However, natural language is ambiguous and


© Hugo Gonçalo Oliveira, António Paulo Santos, and Paulo Gomes;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 169–184

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

words have different meanings which, depending on the context, might lead to different polarities for the same word. So, the previous representation fails to capture words with senses with different polarities. This problem has been recognised and lead to the creation of polarity lexicons based on concepts, materialised, for instance, by the attribution of polarities to the synsets of a wordnet. For English, SentiWordNet [3] and Q-WordNet [1] include a subset of Princeton WordNet [5] synsets and an automatically assigned polarity. Similarly to most language resources, the manual creation of polarity lexicons involves time-consuming human effort, which lead to the development of automatic tools for this task, either by exploiting corpora or other resources, such as wordnets (see section 2).

A polarity resource based on synsets enables the combination of SA with word sense disambiguation (WSD) techniques [19] to identify the polarity of words in context – WSD algorithms based on the structure of a wordnet can be used to identify the synset corresponding to a word meaning in context and then access a synset polarity lexicon to obtain the transmitted polarity. In fact, the benefits of WSD to SA were emphasised by the acknowledgement that a supervised sentiment classifier modeled on word senses performs better than one based on word-based features, when classifying the polarity transmitted by textual documents [2].

Given the importance of having such a resource, we set our goal to the automatic creation of a synset-oriented polarity lexicon for Portuguese. Our effort towards this goal involved the automatic assignment of polarities to the synsets of a Portuguese wordnet-like lexicon, Onto.PT [7] – a resource currently in development, extracted automatically from textual resources. Having our goal in mind, we describe a procedure with two steps: (i) one for initial polarity assignment; (ii) and another for polarity propagation. Both steps are relatively straightforward and can be applied to other languages, provided that there is a polarity lexicon on the words of that language. Moreover, during this work, there was nothing like a concept-oriented/synset-oriented polarity resource for Portuguese. Therefore, this work can be viewed as an important contribution to the development of Portuguese SA. For the same reason, we had to perform several manual evaluations, which we also describe.

In the proposed procedure, polarity is first assigned to synsets according to the polarity of the words they contain. The main idea is that the synset polarity will be the same as the typical polarity of the majority of its lemmas, when uncontextualised. Then, through several iterations, polarity is propagated to synsets connected, by semantic relations, to the previously polarised synsets.

After evaluating the results of this procedure manually, we concluded that the initial polarity assignment is an effective way of moving from existing polarity information based on lemmas, to polarity based on meanings, as the accuracy of this step is between 73% or 86%, depending on the human evaluator. As for propagation, regarding that polarities are not transmitted in the same way by all relation types, we conducted an experiment to investigate which relations could be exploited for this task. We concluded that relations between adjectives and qualities or states, as well as those between adjectives and adverbs, tend to preserve the polarity. Furthermore, antonymy relations invert the propagated polarity. Using only the best performing relations, polarity propagation is about 63% accurate. In the end, a polarity lexicon with about 10,300 synsets is obtained, with an estimated global accuracy between 70% and 79%.

This introduction is followed by section 2, where some related work is introduced, with special focus on the automatic creation of polarity lexicons. Section 3 gives a general presentation of our approach for assigning polarities to the synsets of a wordnet in two automatic steps: initial assignment and polarity propagation. Section 4 describes the resources

used when following the proposed approach for Portuguese, Onto.PT and SentiLex-PT [24]. Section 5 reports on the results of applying the initial assignment on Onto.PT synsets, including their evaluation. Following, section 6 reports on the results of applying polarity propagation on Onto.PT, including the evaluation performed towards the selection of the relation types to exploit. Before concluding, section 7 presents and discusses the overall evaluation of the polarity lexicon generated with the proposed approach and using the resources described earlier.

2 Related Work

In order to treat opinions computationally, in SA, sentiments are commonly converted to a simpler "formal language", such as the representation of polarity as numeric values. Polarity lexicons, or sentiment lexicons, are lists of words (or conceptual representations) classified according to their polarity, which may be positive, negative and, possibly, neutral. They are typically used as external sources of knowledge in the automatic classification of textual information, according to its polarity. Polarity lexicons are thus valuable resources for SA, and it is no surprise that their creation is one of the main research lines in this area. There are two main approaches for the automatic construction of polarity lexicons: (i) corpus-based; and (ii) wordnet-based.

Corpus-based approaches (e.g. [9, 26, 14, 12]) explore the co-occurrence of words in large collections of texts. Co-occurrence is explored using linguistic and statistical heuristics. For instance, conjunctions (e.g. *and*, *or*, *but*) between adjectives provide indirect information about their orientation [9] – while sequences like *fair and legitimate* and *corrupt and brutal* have the same orientation and may co-occur in a corpus, the pairs *fair and brutal* and *corrupt and legitimate* would be semantically anomalous. Other approaches compute the orientation of sentences based on their association with positive (e.g. *excellent*, *good*) and negative (e.g. *poor*, *bad*) references [26].

WordNet-based approaches (e.g. [16, 13, 23]) explore information provided by Princeton WordNet [5], or similar resources, to generate polarity lexicons. Exploited information goes from the semantic relations (e.g hypernymy, antonymy) connecting synsets, to textual glosses, which are additional sources of related words.

As words have different meanings, the same word might have senses with different polarities. Polarity lexicons structured in simple lemmas are thus impractical for most applications. This problem has been recognised and lead to the creation of polarity lexicons structured on concepts. For English, SentiWordNet [3] is an example of such a resource. Each relevant WordNet synset is assigned three numerical scores indicative of how positive, negative, and objective (neutral) is the sentiment it transmits. Synsets are classified after combining the results of eight ternary classifiers. The classification score is proportional to the number of classifiers that have assigned one of the three polarities.

Instead of relying on supervised classifiers, the creation of Q-WordNet [1] is based on an unsupervised binary classifier that tries to link each synset to a positive or negative quality. For this purpose, they start with the WordNet synsets that are in an attribute relation with a sense of the word *quality*, which include the adjective synsets with senses of *good*, *bad*, *positive*, *negative*, *superior* and *inferior*. Synsets that are accessible from the previous are then polarised, according to their connection.

Synset-based polarity lexicons for other languages have also been created. Some are based on the automatic translation of existing English resources, including SentiWordNet (see [15, 17]). Of those, some start with a set of manually labelled synset seeds and propagate

polarities to other synsets, through some of the semantic relations [17]. In addition to the ideas developed in Q-WordNet, the authors of SentiWordNet have shown that a random-walk model as the PageRank algorithm may be used for assigning polarities automatically to synsets, and thus expand polarity lexicons [4]. For this purpose, WordNet is seen as a graph, where synsets are nodes connected by relations $<synset_1$ referred-by $synset_2>$. PageRank is ran twice: first, to obtain the positivity strength, only the positive synsets of SentiWordNet have inital weights; then, the same is done for negative synsets.

For Portuguese, the field is growing and there have been a few attempts to create or enrich polarity lexicons automatically. But so far, existing resources are structured in words and not concepts. Exploratory work on the automatic construction of a word-based polarity lexicon for Portuguese includes combining information from different sources [25], propagating polarity through dictionary entries [21], and exploiting synonymy resources for expanding a handcrafted polarity lexicon [24]. On the first [25], a polarity lexicon was obtained after combining information from: (i) the application of Turney's method [26] to a Portuguese corpus of movie reviews; (ii) the application of Kamps' method [13] to the Portuguese thesaurus TeP [18]; (iii) the translation of Liu's English Opinion Lexicon [11] to Portuguese. On the second, starting with a small set of positive and negative seeds (about 10), textual patterns in the definitions of an electronic dictionary were exploited for polarity propagation [21]. The third is JALC [24], an algorithm for the automatic expansion of a handcrafted polarity lexicon of Portuguese, SentiLex-PT. Still, about three quarters of the entries of SentiLex-PT are the result of manual labour. Another work [6] describes on the manual creation of a polarity lexicon for Portuguese, though much smaller than SentiLex-PT, based on the analysis of a corpus of book descriptions.

Our work combines several ideas from the aforementioned works. More precisely, it assumes that all the words of a synset contribute to its overall polarity (as in [3]) and that polarity is propagated through several semantic relations (as in [4, 1, 17, 21]).

3 Assigning Polarity to Wordnet Synsets

This section gives a general overview of our approach for assigning polarity to part of the synsets of a wordnet. This can be achieved by following a straightforward automatic procedure with two sequential steps, namely:

1. Initial polarity assignment, described in section 3.2;
2. Polarity propagation through semantic relations, described in section 3.3.

The initial assignment step can be used, for instance, to assign polarities to the synsets of a simple thesaurus, containing just synsets and not synset links. On the other hand, polarity propagation is made through semantic relations, and thus requires that the initially polarised synsets are explicitly connected to other synsets, according to their meaning. Before describing each step, in section 3.1 we refer the kind of pre-existing resources that are required for applying this procedure and, at the same time, we introduce the notation used to describe each step.

3.1 Set-up

The starting point of this procedure in an existing polarity reference R, and a wordnet with synsets W. R is a list of pairs $(l_i,\ pol(l_i))$ that assigns polarities to lemmas, $R = \{< l_1, pol(l_1) >, < l_2, pol(l_2) >, ..., < l_n, pol(l_n) >\}$, $n = |R|$. The polarity of a lemma, $pol(l_i)$, denotes the sentiment typically expressed by the lemma, when alone, which can be

positive (+1), negative (-1) or neutral (0). This is usually the polarity of the most frequent sense of the lemma, and the one that first comes to mind when in its presence.

A wordnet contains synsets and semantic relations. A synset S is a set of synonymous lemmas l_i, $S = \{l_1, l_2, ..., l_m\}$, $m = |S|$. This means there is a context where all the lemmas of a synset have the same meaning and they can be seen as the possible lexicalisations of the same natural language concept. Semantic relations have a defined type, RT, and connect two synsets according to their meaning $<S_i\ RT\ S_j>, S_i \in W, S_j \in W$.

3.2 Initial Polarity Assignment

The initial assignment is based on the assumption that all the lemmas in a synset contribute to its overall polarity, as in SentiWordNet. Even though some lemmas might have other senses, we believe that the majority of lemmas in a positive synset will be labelled as such in R. Likewise, negative synsets will have a majority of negative lemmas and neutral synsets will contain mostly neutral lemmas.

For the initial assignment, each synset has a counter for each polarity value, respectively c_+, c_- and c_0, all initially set to 0. Then, for each lemma in a synset that is also in the reference, $l_i \in S \wedge l_i \in R$, the polarity of the lemma according to R, $pol(l_i)$, is summed to the corresponding counter:

$$\begin{aligned} pol(l_i) > 0, &\quad c_+ = c_+ + 1 \\ pol(l_i) < 0, &\quad c_- = c_- + 1 \\ pol(l_i) = 0, &\quad c_0 = c_0 + 1 \end{aligned}$$

The overall synset polarity, Pol(S), will be positive (+), negative (-) or neutral (0), if the counter with the highest value is c_+, c_- or c_0 respectively. If there is a tie, the synset is considered to transmit an ambiguous sentiment and will not have an assigned polarity. To illustrate this step, take the following example:

- **Reference** (R):
  - $nice(+)$
  - $overnice(-)$
  - $squeamish(-)$
  - $prissy(-)$
- **Synset** (S):
  - $\{squeamish, prissy, overnice, nice, dainty\}$
- **Counters**: $c_+ = 1, c_- = 3, c_0 = 0$
- $max(c_x) = c_- \implies Pol(S) < 0$

3.3 Polarity Propagation

The propagation step is based on the assumption that the polarity of related synsets tends to related. Therefore, after the initial assignment, all synsets with an assigned polarity can propagate it to their adjacent synsets. In other words, polarised synsets transmit their polarity directly to related synsets, which are those directly connected by a semantic relation.

This might not be true for all types of relations. For instance, *joy* and *sadness* are both *emotions*, meaning that <*emotion* hypernym-of *joy*> and <*emotion* hypernym-of *sadness*>.

But *emotion* can have neutral polarity, while *joy* is definitely positive and *sadness* is negative. Similar situations could happen for meronymy relations, as an object can have both "good" and "bad" parts. In order to identify the relations that propagate polarity more consistently, and could thus be exploited for automatic polarity propagation, we conducted the experimentation described in section 6.1.

Propagation can go through several iterations and occur for both ways. Therefore, in a relation between synsets S_a and S_b, $<S_a\ RT\ S_b>$, if S_a is polarised and S_b is not, the polarity of S_a is propagated to S_b, as well as, if S_b is polarised and S_a is not, S_b propagates its polarity to S_a, such that $Pol(S_a) = Pol(S_b)$ or, depending on the relation, possibly $Pol(S_a) = -Pol(S_b)$. Take the following illustrative example, considering that *similar-to* is a semantic relation that propagates the same polarity and *antonym-of* is a semantic relation that propagates an inverted polarity.

- **Synsets**:
  - $S_0 = \{squeamish, prissy, overnice, nice, dainty\}$
  - $S_1 = \{fastidious\}$
  - $S_2 = \{unfastidious\}$
- $Pol(S_0) < 0$
- $<S_0$ similar-to $S_1> \implies Pol(S_1) < 0$
- $<S_0$ antonym-of $S_2> \implies Pol(S_2) > 0$

This process can occur for several iterations and synsets already polarised can be reached again. But, as the accuracy in polarity attribution decreases for higher iterations, we decided to keep only the polarity transmitted in the first iteration the synset is reached. Its polarity does not change in further iterations. Still, if a synset is reached more than once in the same iteration, we have used the three counters, in a similar fashion to the initial assignment. This way, in the end of each iteration, the synset gets the polarity corresponding to the counter with the highest value.

4 Used Resources

The automatic creation of a synset-based polarity lexicon for Portuguese relied on two existing resources, both freely available from the Web. In this section, we present both of them, namely: SentiLex-PT [24], a lemma-based polarity lexicon, and Onto.PT [7], a wordnet-like lexical knowledge base.

4.1 SentiLex-PT

SentiLex-PT is a polarity lexicon for Portuguese, compiled from several publicly available Portuguese resources. The sentiment entries of this lexicon are words, associated with their morphological properties and predicted sentiment towards human subjects. SentiLex-PT is especially suitable for opinion mining in Portuguese, particularly for detecting and classifying sentiments and opinions targeting human entities.

We used SentiLex-PT02[1], the most recent version of this resource. It is available in two files: (i) one where the word entries are inflected; (ii) and a "compressed" file where

[1] Available from `http://dmir.inesc-id.pt/project/SentiLex-PT_02`

all entries are lemmatised. In this work, we used the second one, because Onto.PT also contain lemmatised words. This lemma-oriented file covers 7,014 lemmas, of which 4,779 are adjectives, 1,081 are nouns, 489 are verbs and 666 are idiomatic expressions.

Besides other properties, each entry of SentiLex-PT02 contains the polarity of the word, which can be positive, neutral or negative, and also its kind of annotation, which is either manual (5,473 lemmas) or automatic (1,541 lemmas). Manual polarity labels were given by a linguist and automatic labels were assigned by the JALC algorithm, which its authors claim to be 87% accurate [24]. Out of curiosity, the lemma-oriented file of SentiLex-PT02 contains about three times more lemmas with negative polarity (4,596) than positive (1,548), and just 860 lemmas have neutral polarity.

4.2 Onto.PT

Onto.PT is a lexical-semantic knowledge base for Portuguese, structured similarly to Princeton WordNet [5]. As typical wordnets, Onto.PT tries to cover the whole language and not just specific domains. It is also structured on synsets, which group synonymous word senses, represented by lemmas, that may be seen as natural language concepts. For each of their senses, polysemous words are included in a different synset. Synsets may be connected to other synsets by means of semantic relations, which help describing possible interactions between their meanings.

However, we refer to Onto.PT as a wordnet-like resource because, in opposition to typical wordnets, it is not handcrafted, but created automatically, by exploiting Portuguese dictionaries and thesauri. The construction of Onto.PT is briefly described in three steps, that comprise the ECO approach [7]:

1. Regularities in the definitions of dictionaries are exploited for the extraction of instances of semantic relations, connecting words, identified by their lemma.
2. If possible, each synonymy relation is attached to a synset in an existing Portuguese thesaurus. TeP [18], an electronic thesaurus for Brazilian Portuguese, is used in this step. Clusters are then identified in the set of unattached synonymy relations, and added as new synsets.
3. Graph-based similarities are used to integrate the rest of the semantic relations automatically. Each lemma argument of a relation is assigned to the most suitable synset. If there are no synsets with the lemma, a new synset is created with that lemma.

Following this procedure, it is possible to have a larger resource without having to rely on time-consuming manual work. This lead to other differences towards typical wordnets, including more relation types covered. In Onto.PT, relations go from well-known hypernymy and part-of, to relations established between words of different parts-of-speech (POS), including, for instance, purpose-of, manner-of, or has-quality.

Onto.PT was released in 2012 and, as the result of an automatic approach, it is always under development[2]. In this work, we have used version 0.3 of Onto.PT, which contains 160,791 unique lemmas, organised in 105,500 synsets – 60,197 nouns, 25,346 verbs, 17,961 adjectives and 1,996 adverb synsets – connected by 184,521 instances of semantic relations.

Other Portuguese wordnets, as MultiWordNet.PT[3] or OpenWordNet.PT [22], would not apply for this work because they are both smaller and cover mostly hypernymy and

[2] Check `http://ontopt.dei.uc.pt` for updates and additional information on Onto.PT.
[3] Check `http://mwnpt.di.fc.ul.pt/` for additional information on MultiWordNet.PT.

Table 1 Examples of synsets and their polarity in the initial assignment.

Synset	Polarity
contente (+), *alegre* (+), *satisfeito* (+), *radiante* (+), *feliz* (+), *jubiloso* (+) (*content*, *cheerful*, *satisfied*, *radiant*, *happy*, *joyant*)	+
verdadeiro (+), *veraz* (+), *verídico* (+), *fidedigno* (+), *fiel* (+), *exacto* (*true*, *truthful*, *veridical*, *reliable*, *faithful*, *exact*)	+
esmorecido (-), *débil* (-), *sumidiço*, *mortiço* (-), *fraco* (-), *apagado* (-) (*faltering*, *feeble*, *dull*, *weak*, *out*)	-
médio (0), *mediano* (0), *medíocre* (-), *moderado* (+) (*average*, *median*, *mediocre*, *moderate*)	0
severo (-), *implacável*, *justiceiro* (+), *estrito*, *rigoroso* (+), *incomplacente*, *inflexível* (-) (*severe*, *implacable*, *justicer*, *strict*, *rigorous*, *austere*, *inflexible*)	*null*

part-of relations. As discussed earlier, those are not the best suited for polarity propagation. Moreover, MultiWordNet.PT is not free for research purposes and only covers noun synsets.

5 Results of Initial Assignment

The polarity assignment procedure, presented in section 3 was followed in the creation of a synset-based polarity lexicon for Portuguese, using the resources described in section 4. We recall that this approach encompasses two steps: initial polarity assignment and polarity propagation. This section reports on the results of the first step, which consisted of assigning a polarity to the synsets of Onto.PT, using SentiLex-PT as the polarity reference. Section 6 presents the results of the polarity propagation in Onto.PT.

5.1 Quantities and Examples

The initial polarity assignment was applied to Onto.PT, using the full SentiLex-PT as a polarity reference. This resulted in 7,556 Onto.PT synsets with an assigned polarity, more precisely: 1,875 positive, 4,792 negative and 889 neutral. Of those, 1,374 synsets included both lemmas with positive and lemmas with negative polarities, but one of the three polarity counters was higher than the others. An additional 424 synsets were considered to be sentiment ambiguous. Table 1 shows examples of synsets, the polarity of their lemmas according to SentiLex-PT, and their consequently assigned polarity. The few lemmas without polarity are not covered by SentiLex-PT. Examples include synsets where all lemmas have the same polarity, even though some are not in SentiLex-PT, synsets where the resulting polarity is the most common, and a synset with ambiguous polarity (*null*).

5.2 Evaluation

In order to assess the results produced by the initial polarity assignment, we asked two human judges, both native speakers of Portuguese, to independently classify a sample of 390 synsets according to their polarity. Their goal was to select the polarity that the concept denoted by each synset transmits. All the synsets were randomly collected from the 7,556 synsets with polarities automatically assigned in this initial step, but the automatic polarity was not shown to the judges. Also, all the synsets of the sample had more than one lemma. First, this procedure is suited precisely for synsets with more than one lemma. Second, assuming that the sentiment labels in SentiLex-PT are correct, there was no need to evaluate

Table 2 Evaluation of the initial assignment step.

Sample	Reference	Target	P(+)	P(-)	P(0)	P(all)	Kappa
390 sets	H1	Aut	0.92	0.88	0.38	0.86	0.73
390 sets	H2	Aut	0.72	0.75	0.50	0.73	0.53
390 sets	H1	H2	0.77	0.81	0.82	0.80	0.66

any of the 1,315 polarised synsets with only one lemma[4].

We recall that each of the other synsets is a group of lemmas that, all together, denote a concept. Therefore, even if there is one or more lemmas that, in different contexts, may transmit different polarities, there should only be one meaning shared between all the lemmas and, in this context, only one polarity common to all of them. This fact is important because it minimises the ambiguity issues of polarity classification, as compared to the classification of single lemmas, outside a context.

Concerning the illustration of the aforementioned phenomena, we first present two meanings of the Portuguese word *queda*, which might either denote a downfall/tumble (negative) or an ability/capacity (typically positive), respectively in the following synsets:

- ***queda**, tombo, trambolhão, choque, baque, boléu*
- ***queda**, jeiteira, vocação, qualidade, aptidão, jeito, habilidade, capacidade*

Similarly, in the following synsets, the adjective *simples* might respectively refer to something simple/easy (typically positive) or an ignorant/uneducated (negative) person:

- ***simples**, fácil, desintrincado*
- ***simples**, inculto, bronco, ignorante, burgesso, néscio, desiluminado*

Table 2 presents the results of this evaluation. Besides the number of evaluated synsets (Sample), 'Reference' indicates the reference set of polarised synsets, which can be viewed as a golden set, while 'Target' is the evaluated set. In the later columns, 'H1' stands for the first human judge, 'H2' for the second human and 'Aut' refers to the synsets with polarities assigned automatically. In the same table, we provide the accuracy of our target according to the reference, which is the proportion of matches per polarity value (P(+), P(-), P(0)), the total accuracy (P(all)), and the agreement between the reference and the target, expressed by the Cohen's Kappa coefficient (Kappa). Although not common, this includes the agreement between the judges annotation and the system.

Evaluation shows that the initial assignment is an adequate and straightforward approach for moving from polarised lemmas to polarised synsets/meanings. Using the judge H1 as reference, the accuracy of the initial polarity assignment is 86%, and it is 73% against judge H2. Curiously, even though the agreement between the judges was good (0.66) [8], H1 had higher agreement with the system than with H2.

Accuracy is always higher than 70% for positive and negative synsets, but it is lower for the neutral synsets. Besides suggesting that it is not easy to identify objective/neutral synsets automatically, this highlights the fact that it is not easy for humans as well. Due to these problems, several works (e.g. [26, 23, 21]) just classify words as either positive or negative. In fact, the difference of accuracy between the two annotators is explained by their

[4] We did not consider the possibility that the sense of the lemma in the Onto.PT synset was different than in SentiLex-PT. The main reason for this is that, in Onto.PT, single-lemma synsets are unique. In fact, for rare situations, such as words with two completely different senses without synonyms, single-lemma synsets might merge different senses.

different sensibilities towards neutral synsets – the amount of synsets classified this way is 2.5 times higher for H2 than for H1. If neutral synsets were ignored, the accuracies would actually be 97% and 96% respectively for H1 and H2, with $\kappa = 0.96$.

6 Results of Polarity Propagation

On the proposed approach, the second step for creating a synset-based polarity lexicon is polarity propagation. However, before propagating polarities, blindly, through all types of relations, we only selected the types which seemed adequate for propagation. Then, we evaluated the result of their propagation in the first iteration, where four types of candidate relations revealed to be inadequate for this task. Only the remaining five types were used for propagation.

6.1 Candidate Relations

Regarding the analysis of several examples, as those in section 3.3, we decided not to use hypernymy nor meronymy for polarity propagation. There are works [17] where hypernymy is used for polarity propagation, but only when this relation is held between adjectives. This is not the case for Onto.PT, where hypernymy only connects nouns. On the other hand, we believed that several types of Onto.PT relations could transmit polarities more consistently, namely:

- Relations connecting an object or a process to a resulting/goal state or another object/process (causation, producer, purpose);
- Relations connecting properties, qualities or states with nouns or adjectives (refers-to, has-quality, has-state);
- Relations connecting nouns or adjectives with adverbs (manner).

Additionally, we believed that there were types of relations connecting positive with negative synsets, which thus transmit an inverted polarity. In Onto.PT, two types of relations fit in that group:

- Those connecting synsets with an opposite meaning (antonymy, which, in Onto.PT 0.3, only occurs between adjectives);
- Relations connecting nouns or verbs with manners that do not characterise them (manner-without).

For each of the aforementioned relation types, Table 3 presents an illustrative example, together with the number of instances in Onto.PT 0.3.

6.2 Selection of the Adequate Relations

Instead of using all the selected relations in polarity propagation, once again, we asked two human judges to independently classify the polarity of Onto.PT synsets of seven samples. Each sample contained synsets connected by one of the seven relation types in Table 3, to those polarised in the initial assignment. For evaluation purposes, the automatic results of propagation were compared to the manual classifications. Table 4 shows the results of this evaluation. The number of evaluated synsets differs according to the reference because, when judges could not attribute a well-defined meaning to a synset, they did not classify it. This happened because, although the judges were advised to look in online dictionaries for unknown meanings, Onto.PT contains some unfrequent words, as well as unfrequent senses of well-known words.

Table 3 Relations of Onto.PT, evaluated for propagation.

Has-quality: 2,219 instances
{*inábil*} → {*desajeitamento,desjeito,inabilidade,desabilidade*}
{*unskilful*} → {*clumsiness,inability,lack_of_skill*}

Has-state: 573 instances
{*estável,permanente,efectivo*} → {*beatitude,paz,concordia,tranquilidade*}
{*stable,permanent*} → {*bliss,peace,tranquility*}

Manner-of: 3,924 instances
{*avidamente,vorazmente,sofregamente*} → {*sede,avidez,sofreguidão,avareza,cobiça,...*}
{*greedily*} → {*greed*}

Antonym-of: 687 instances
{*inconcludente,inconclusivo*} → {*liquidante,conclusivo,terminativo,terminante*}
{*inconclusive*} → {*conclusive,terminative*}

Manner-without: 316 instances
{*caladamente,silenciosamente,secretamente, ...*} → {*exposição,manifestação,declaração*}
{*quietly,silently,secretly*} → {*exposition,expression,declaration*}

Causation-of: 12,148 instances
{*causticar,cauterizar,calcinar,...*} → {*combustão,cremação,cauterização,calcinação,...*}
{*to_etch,to_cauterise*} → {*combustion,cauterization*}

Producer-of: 2,335 instances
{*destilação_de_petróleo*} → {*gasolina*}
{*oil_distillation*} → {*gasoline*}

Purpose-of: 16,918 instances
{*explorar_espaço*} → {*cosmonave,astronave,espaçonave,nave*}
{*to_explore_the_space*} → {*spacecraft,spaceship*}

Refers-to: 37,491 instances
{*caricaturesco,caricatural*} → {*caricatura,cartoon,cartum*}
{*caricatural*} → {*caricature,cartoon*}

On the performed evaluation, depending on the relation, the judge's agreement is between moderate and good [8]. Yet, we believe that we can rely on these results to conjecture on the adequacy of the Onto.PT relations for polarity propagation. As discussed earlier, the task of classifying the polarity of synsets depends on the judge's intuition. For instance, in the previous evaluation, we noticed different sensibilities towards the classification of the synsets as neutral. As for the actual evaluation results, Table 4 shows that manner-of and manner-without were the best performing relation types, which indicates that means and adjectives transmit their polarity to their corresponding adverbs. Not just these two, but all five relation types with best accuracy denote a pattern, as they all connect at least one adjective or adverb synset to another synset. Given that both adjectives and adverbs are used as modifiers, they are often connected to qualities and thus to sentiment, which explains this pattern. On the other hand, purpose-of and producer-of had the lowest accuracy. In fact, purpose-of is not as semantically well-defined as the other relations because it can connect very different things. To give an idea, it relates an action (verb), which can either be a general purpose (e.g. *to desinfect, to calculate, to censor, to dissociate*) or just something one can do with (e.g. *to punish, to transport, to climb, to spend, to entertain*), for instance, an instrument (e.g. *desinfectant, whip*), a concrete object (e.g. *van, stairs*), an abstract means (e.g. *credit, calculation, satire*), a human entity (e.g. *clown*), or a property (e.g. *dissociation*). And we should recall that the same instrument/means can be used either for positive or negative actions. As for the producer-of relation, most of its instances relate fruits and vegetables with their trees, which are rarely related to sentiment.

Table 4 Evaluation of different relations in the first propagation iteration.

Relation	Sample	Ref	Target	P(+)	P(-)	P(0)	P(all)	Kappa
Causation-of	99 sets	H1	Aut	0.36	0.69	0.00	0.59	0.26
	100 sets	H2	Aut	0.63	0.42	0.00	0.56	0.26
	99 sets	H1	H2	0.66	0.80	0.50	0.68	0.45
Producer-of	79 sets	H1	Aut	0.29	0.33	0.00	0.30	0.09
	77 sets	H2	Aut	0.29	0.43	0.00	0.36	0.14
	77 sets	H1	H2	0.73	0.75	0.93	0.84	0.73
Purpose-of	99 sets	H1	Aut	0.29	0.23	0.83	0.29	0.12
	98 sets	H2	Aut	0.24	0.20	0.50	0.23	0.05
	97 sets	H1	H2	0.57	0.77	0.86	0.78	0.57
Refers-to	118 sets	H1	Aut	0.37	0.54	0.33	0.48	0.17
	119 sets	H2	Aut	0.54	0.56	0.27	0.53	0.23
	117 sets	H1	H2	0.52	0.78	0.61	0.67	0.48
Has-quality	85 sets	H1	Aut	0.85	0.81	0.17	0.78	0.60
	85 sets	H2	Aut	0.85	0.75	0.50	0.76	0.60
	85 sets	H1	H2	0.90	0.90	0.57	0.85	0.75
Has-state	60 sets	H1	Aut	0.38	0.80	0.60	0.67	0.37
	60 sets	H2	Aut	0.50	0.77	0.40	0.67	0.38
	60 sets	H1	H2	0.43	0.86	0.60	0.72	0.48
Manner-of	90 sets	H1	Aut	0.90	0.75	0.22	0.74	0.56
	90 sets	H2	Aut	0.83	0.75	0.00	0.70	0.48
	90 sets	H1	H2	0.88	0.84	0.29	0.81	0.67
Antonym-of	60 sets	H1	Aut	0.52	0.79	0.43	0.62	0.42
	60 sets	H2	Aut	0.55	0.71	0.43	0.60	0.40
	60 sets	H1	H2	0.71	0.92	0.74	0.80	0.70
Manner without	85 sets	H1	Aut	0.62	0.82	0.00	0.74	0.51
	85 sets	H2	Aut	0.66	0.82	0.00	0.75	0.51
	85 sets	H1	H2	0.70	0.85	0.56	0.78	0.59

According to our interpretation, these results make sense, so we relied on them for selecting the relations to use. This means that polarity was propagated only through the five relation types with accuracy higher than 60%, namely: manner-of, has-quality, has-state, antonymy and manner-without.

6.3 Polarity Propagation through Selected Relations

After selecting the adequate relation types, the polarities assigned in the first step were propagated until every synset, connected directly or indirectly through one of the five selected types, had been reached. The algorithm ran for eight iterations and then stopped, with 10,318 polarised synsets. This number is lower than if all the relations in Table 4 are used (43,468), but we preferred to have a smaller but more reliable polarity lexicon. We can however, in the future, generate larger polarity lexicons, in a trade-off for lower reliability.

Table 5 has two examples of polarity propagation from the initial assignment until iteration 3, through different semantic relations. In the same table, 'Iter' is the iteration number, or 0 for the initial assignment, and 'Pol' is the propagated polarity.

7 Overall Evaluation

In order to complement the evaluation of the generated lexicon, we performed one last evaluation, where the polarity of 500 synsets, polarised in iterations 1 to 8, was compared, once again, with the polarity given manually by two human judges. The results of this evaluation are shown in Table 6, all together, and in Table 7, according to the iteration and starting with the evaluation of the 390 synsets polarised in the initial assignment (same as Table 2). As expected, accuracy becomes lower for higher iterations. The agreement becomes

Table 5 Examples of initial assignment (0) and propagation.

Iter	Pol	Propagation
0	-	**(adj)** *incorrigível* (-), *destravancado*, *irregenerável*, *indisciplinável*, *insubordinável* (*incorrigible* (-), *unregenerable*, *undisciplinable*, *rebellious*)
1	-	**Has-quality** → **(n)** *incorrigibilidade*, *irreparabilidade* (*incorrigibility*, *irreparability*)
2	-	**Quality-of** → **(adj)** *irrecuperável*, *irremediável*, *incompensável*, *irreparável*, *insubstituível*, *insuprível* (*irrecoverable*, *irremediable*, *not_compensable*, *irreparable*, *irreplaceable*)
3	-	**Has-manner** → **(adv)** *irreparavelmente* (*irreparably*)
0	+	**(n)** *concordância*, *consentimento*, *autorização*, *beneplácito*, *tolerância* (+), *permissão*, *licença* (*agreement*, *consent*, *permission*, *tolerance* (+))
1	+	**Has-manner** → **(n)** *outorgadamente* (*consentingly*)
2	+	**Manner-of** → **(adj)** *deferido*, *outorgado*, *concedido* (*deferred*, *granted*)
3	-	**Antonym-of** → **(adj)** *impedido*, *tolhido*, *vedado*, *proscrito*, *negado*, *defeso*, *interdito*, *proibido*, *inconcesso* (*prevented*, *fenced*, *denied*, *forbidden*, *prohibited*)

Table 6 Evaluation of iterations 1 to 8.

Sample	Ref	Target	P(+)	P(-)	P(0)	P(all)	Kappa
500 sets	H1	Aut	0.65	0.66	0.38	0.63	0.42
500 sets	H2	Aut	0.68	0.65	0.42	0.64	0.43
500 sets	H1	H2	0.75	0.84	0.62	0.75	0.62

lower as well for higher iterations. Nevertheless, until iteration 4, it is always higher than 0.54. These results also suggest that, for more reliable results, the algorithm should stop before there are no more reachable synsets, for instance, in iteration 2, or maybe 4. Also in Table 7, we present the combined accuracy for the whole lexicon, which is about 78.9% or 70% using H1 or H2 respectively as reference. This value considers the number of synsets polarised in the initial step, polarised through propagation, and the accuracy of those steps.

For the analysis of these results, it is worth reminding that Onto.PT is a resource created automatically and in development. Consequently, it is not 100% reliable. For instance, it contains a few incorrect relations, which have a negative impact on the propagation results. On the other hand, Onto.PT tends to keep growing and to improve its reliability, which will consequently have a positive impact on future polarity lexicons generated by the proposed approach. In fact, there are more recent version of this resource, released after 0.3.

8 Concluding Remarks

We have described an approach for moving from the polarity of single lemmas to the polarity of concepts/meanings, represented as wordnet synsets. This kind of approach is an alternative to tedious and time-consuming annotation tasks, performed by humans.

Table 7 Evaluation and quantity of synsets according to iteration (0 to 5).

Iteration	Synsets	Sample	Ref	P(all)	Kappa
0	7,556	390	H1 H2	0.86 0.73	0.66
1	1,971	330	H1 H2	0.71 0.72	0.63
2	549	112	H1 H2	0.48 0.50	0.57
3	163	38	H1 H2	0.45 0.34	0.55
4	56	15	H1 H2	0.60 0.60	0.54
5	16	4	H1 H2	0.25 0.00	0.00
...					
Total	10,318	890	H1 H2	0.79 0.70	0.65

The proposed approach is language independent, but we have applied it to a Portuguese wordnet-like resource. Though relatively straightforward, the accuracy of the polarised synsets confirms that this approach is quite solid. In its first step, we have shown that synsets can be polarised according to the sum of the polarity of their lemmas alone, with accuracies close to 90%. In the second step, polarities were propagated for several iterations, through a set of selected relation types. For selecting the relations to use, we measured the accuracy of different types for this task and observed that relations connecting adjectives or adverbs to other synsets were more suitable for this. We also concluded that the accuracy of polarity attribution decreased for higher iterations.

In the end, we obtained a polarity lexicon for Portuguese derived from Onto.PT, with 10,318 polarised synsets, and polarities between 70% and 79% accurate, depending on the judge. As far as we know, while writing this paper, the resulting polarity lexicon was the first synset-based resource of this kind targeting Portuguese. Though, very recently, we became aware that, taking advantage of the alignment between OpenWordNet.PT and Princeton WordNet, SentiWordNet polarities have been assigned to OpenWordNet.PT synsets [22]. Although this effort might suffer from issues regarding the translation of lexicons from one language to another (different languages represent different socio-cultural realities, they do not cover exactly the same part of the lexicon and, even where they seem to be common, several concepts are lexicalised differently [10]), it is another relevant contribution for the development of Portuguese SA applications. We should recall that, if combined with WSD techniques, synset-based polarity lexicons will improve the attribution of a polarity to words in context, and thus the polarity classification of various kinds of text. Therefore, following the free availability of Onto.PT, the result of assigning polarities to a more recent version of Onto.PT is available from this project's website (`http://ontopt.dei.uc.pt`).

Even though the obtained results are interesting, additional work is needed, in order to get a more reliable resource. Therefore, we end by leaving some lines for further work. First, looking at the obtained results, it might be a good idea to decrement polarity strength in each propagation iteration. This would enable the algorithm to stop either when there are no more reachable synsets or when the propagated polarity is below some threshold. Second, more than adequate relation types for polarity propagation, we believe that there are combinations of types that lead to good polarity propagation, and combinations that don't. It would be interesting to learn these combinations, which can be seen as paths (e.g. A *hyponym-of* B *has-quality* C), semi-automatically, by selecting paths between synsets with

correct polarities. Finally, we believe that polarity represented by three parameters (positive, negative and neutral) is limitative and should be rethought. For instance, instead of being represented by only one value, polarity can be represented by a real value from 0 to 1, or by three parameters, respectively indicating the positive, negative and neutral strength, in a similar fashion of what is done in SentiWordNet. This could be achieved by propagating polarity using a method as PageRank, in a similar fashion to existing work for English [4]. If the polarity values obtained this way are normalised, it will be possible to compare them to the results obtained with the approach proposed in this article.

Acknowledgements. This work was supported by the iCIS project (CENTRO-07-ST24-FEDER-002003), co-financed by QREN, in the scope of the Mais Centro Program and European Union's FEDER.

References

1 Rodrigo Agerri and Ana García-Serrano. Q-wordnet: Extracting polarity from WordNet senses. In *Proceedings of the 7th International Conference on Language Resources and Evaluation*, LREC 2010, La Valletta, Malta, 2010. ELRA.

2 A. R. Balamurali, Aditya Joshi, and Pushpak Bhattacharyya. Harnessing wordnet senses for supervised sentiment classification. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, EMNLP 2011, pages 1081–1091, Edinburgh, Scotland, UK, 2011. ACL Press.

3 Andrea Esuli and Fabrizio Sebastiani. SentiWordNet: A publicly available lexical resource for opinion mining. In *Proceedings of the 5th Conference on Language Resources and Evaluation*, LREC 2006, pages 417–422, 2006.

4 Andrea Esuli and Fabrizio Sebastiani. PageRanking WordNet synsets: An application to opinion mining. In *Proceedings of 45th Annual Meeting of the Association for Computational Linguistics*, ACL'07, pages 424–431, Prague, Czech Republic, 2007. ACL Press.

5 Christiane Fellbaum, editor. *WordNet: an electronic lexical database (Language, Speech, and Communication)*. The MIT Press, 1998.

6 Cláudia Freitas. Sobre a construcção de um léxico da afetividade para o processamento computacional do português. *Revista Brasileira de Linguística Aplicada*, 13(4):1013–1059, 2013.

7 Hugo Gonçalo Oliveira and Paulo Gomes. ECO and Onto.PT: A flexible approach for creating a Portuguese wordnet automatically. *Language Resources and Evaluation*, to be published, 2013.

8 Annette M. Green. Kappa statistics for multiple raters using categorical classifications. In *Proceedings of the 22nd Annual Conference of SAS Users Group*, San Diego, USA, 1997.

9 Vasileios Hatzivassiloglou and Kathleen R. Mckeown. Predicting the semantic orientation of adjectives. In *Proceedings of 35th Annual Meeting of the Association for Computational Linguistics*, ACL 1997, pages 174–181, Madrid, ES, 1997. ACL Press.

10 Graeme Hirst. Ontology and the lexicon. In Steffen Staab and Rudi Studer, editors, *Handbook on Ontologies*, International Handbooks on Information Systems, pages 209–230. Springer, 2004.

11 Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 168–177, New York, NY, USA, 2004. ACM.

12 Nobuhiro Kaji and Masaru Kitsuregawa. Building lexicon for sentiment analysis from massive collection of HTML documents. In *Proceedings of the Joint Conference on Empir-*

ical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2007, pages 1075–1083, 2007.

13 Jaap Kamps, Robert J. Mokken, Maarten Marx, and Maarten de Rijke. Using WordNet to measure semantic orientation of adjectives. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*, volume IV of *LREC 2004*, pages 1115–1118, Paris, France, 2004. ELRA.

14 Hiroshi Kanayama and Tetsuya Nasukawa. Fully automatic lexicon expansion for domain-oriented sentiment analysis. In *Proceedings of the 2006 Conference on Empirical Methods in Natural Language Processing*, pages 355–363, Sydney, Australia, 2006. ACL Press.

15 Jungi Kim, Hun-Young Jung, Yeha Lee, and Yeha Lee. Conveying subjectivity of a lexicon of one language into another using a bilingual dictionary and a link analysis algorithm. *International Journal of Computer Processing Of Languages*, 22(02-03):205–218, 2009.

16 Soo-Min Kim and Eduard Hovy. Determining the sentiment of opinions. In *Proceedings of the 20th international conference on Computational Linguistics*, COLING 2004, pages 1267–1373, Geneva, Switzerland, 2004. ACL Press.

17 Isa Maks and Piek Vossen. Different approaches to automatic polarity annotation at synset level. In *Proceedings of the 1st International Workshop on Lexical Resources*, WoLeR'11, Ljubljana, Slovenia, 2011.

18 Erick G. Maziero, Thiago A. S. Pardo, Ariani Di Felippo, and Bento C. Dias-da-Silva. A base de dados lexical e a interface web do TeP 2.0 – Thesaurus Eletrônico para o Português do Brasil. In *VI Workshop em Tecnologia da Informação e da Linguagem Humana (TIL)*, pages 390–392, 2008.

19 Roberto Navigli. Word sense disambiguation: A survey. *ACM Computing Surveys*, 41(2):1–69, 2009.

20 Bo Pang and Lillian Lee. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval*, 2(1-2):1–135, January 2008.

21 António Paulo-Santos, Hugo Gonçalo Oliveira, Carlos Ramos, and Nuno C. Marques. A bootstrapping algorithm for learning the polarity of words. In *Proceedings of Computational Processing of the Portuguese Language – 10th International Conference (PROPOR 2012)*, volume 7243 of *LNCS*, pages 229–234, Coimbra, Portugal., April 2012. Springer.

22 Alexandre Rademaker, Valeria De Paiva, Livy Maria Real Gerard de Melo, Coelho, and Maira Gatti. Openwordnet-pt: A project report. In *Proceedings of the 7th Global WordNet Conference*, GWC 2014, pages 383–390, Tartu, Estonia, jan 2014.

23 Delip Rao and Deepak Ravichandran. Semi-supervised polarity lexicon induction. In *Proceedings of 12th Conference of the European Chapter of the Association for Computational Linguistics*, EACL 2009, pages 675–682, Athens, Greece, 2009. ACL Press.

24 Mário J. Silva, Paula Carvalho, and Luís Sarmento. Building a sentiment lexicon for social judgement mining. In *Proceedings of 10th International Conference on Computational Processing of Portuguese, PROPOR 2012*, LNCS/LNAI, Coimbra, Portugal, April 2012. Springer.

25 Marlo Souza, Renata Vieira, Debora Busetti, Rove Chishman, and Isa Mara Alves. Construction of a portuguese opinion lexicon from multiple resources. In *Proceedings of 8th Brazilian Symposium in Information and Human Language Technology*, STIL 2011, Cuiabá, Brazil, 2011.

26 Peter D. Turney. Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL 2002, pages 417–424, Philadelphia, PA, USA, 2002. ACL Press.

Detecting a Tweet's Topic within a Large Number of Portuguese Twitter Trends

Hugo Rosa[1], João Paulo Carvalho[2], and Fernando Batista[3]

1 **INESC-ID Lisboa**
IST – Universidade de Lisboa, Portugal
hugohrosa@gmail.com
2 **INESC-ID Lisboa**
IST – Universidade de Lisboa, Portugal
joao.carvalho@inesc-id.pt
3 **INESC-ID Lisboa**
ISCTE-IUL, Lisboa, Portugal
fernando.batista@iscte.pt

Abstract

In this paper we propose to approach the subject of Twitter Topic Detection when in the presence of a large number of trending topics. We use a new technique, called Twitter Topic Fuzzy Fingerprints, and compare it with two popular text classification techniques, Support Vector Machines (SVM) and k-Nearest Neighbours (kNN). Preliminary results show that it outperforms the other two techniques, while still being much faster, which is an essential feature when processing large volumes of streaming data. We focused on a data set of Portuguese language tweets and the respective top trends as indicated by Twitter.

1998 ACM Subject Classification I.2.7 Natural Language Processing, H.2.8 Database Applications, I.5.4 Applications

Keywords and phrases topic detection, social networks data mining, Twitter, Portuguese language

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.185

1 Introduction

No one can deny the importance of public social networks in current modern world society. From event advertising or idea dissemination, to commenting and analysis, social networks have become the de facto means for individual opinion making and, consequently, one of the main shapers of an individuals perception of society and the world that surrounds him. The Arab Spring [10], the Indignant movement protest [16], or presidents tweeting and posting messages on Facebook instead of using official public addressing are just a few examples of how influential social networks have become. Nowadays important events are often commented in social networks even before they become "public news", and even news agencies and networks had to adapt and start using social networks as sources of information.

Among public social networks, Twitter has become a major tool for sociological analysis. However, in order to properly analyse Twitter data, it is necessary to filter which tweets are relevant for a given subject or topic. This is not a trivial problem since there are currently more than 340 millions of daily tweets covering thousands of different topics [9]. Twitter already helps by providing a list of top trends [22] and the hashtag # mechanism: when referring to a certain topic, users are encouraged to indicate it through the use of a hashtag,


© Hugo Rosa, João Paulo Carvalho, and Fernando Batista;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 185–199

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

e.g., "#Obamacare has been approved!" indicates the topic of the tweet is Obamacare. Websites such as #hastags.org make good use of this information to present Twitter trends, e.g., `http://www.hashtags.org/analytics/Obamacare/`.

Other tools such as Twittermonitor [14] can also be used to obtain Twitter trends. However not all tweets related to a given topic are hashtagged. In fact, only roughly 16% of all tweets are hashtagged [15]. These numbers have been confirmed by our (assumedly) small experiments. Therefore, in order to properly analyse a given topic, it is essential to include the most of the remaining 84% of the untagged information.

This task, which we shall refer to as Tweet Topic Detection, involves deciding if a given tweet is related to a given #hashtagged topic. Basically this can be categorized as a classification problem, albeit one with some particular characteristics that need to be addressed specifically: it is a text classification problem, with an unknown and large number of categories, where the texts to be classified are very short texts (up to 140 characters), and it is a problem that fits the Big Data paradigm due to the huge amounts of streaming data.

We distinguish between Topic Classification and Topic Detection. The former defines a short and generic set of categories, ranging from politics to sports and the documents will often belong to at least one of those categories. It is very rare that a tweet does not fit into any topic. The latter takes on a more detailed approach, where an attempt is made to determine the topic of the document, given a predetermined large set of possible topics. In addition, the topics are so unique amongst themselves that there is a high probability that a tweet without a hashtag may very well not belong to any of the current trends.

When considering this difference, the most similar works on Topic Detection within Twitter are those related with emerging topics or trends, for example [14, 3, 11, 19]. In these works the authors use a wide variety of techniques regarding text analysis to find the most common related words and hence detect topics. In our work we already assume the existence of trending topics and we aim at efficient detecting tweets that are related to them, despite not being explicitly marked as so.

It is also possible to find several works regarding Topic Classification. In [13], an attempt is made to classify Twitter Trending Topics into 18 broad categories, such as: sports, politics, technology, etc, and their experiments on a database of randomly selected 768 trending topics (over 18 classes) show that, using text-based and network-based classification modelling, a classification accuracy up to 65% and 70% can be achieved, respectively. Another interesting article, despite not on the theme of Topic Detection, demonstrates how to use Twitter to automatically obtain breaking news from the tweets posted by Twitter users [20]. In 2009, when Michael Jackson passed away, "the first tweet was posted 20 minutes after the 911 call, which was almost an hour before the conventional news media first reported on his condition". This further enforces the importance of automatically analysing the massive amount of information on Twitter.

In what concerns text classification, K-Nearest Neighbours (kNN) and the Support Vector Machine (SVM) are amongst the most widely used and best performing classifiers. In [23], Yang and Liu, performed several tests in a controlled study and reported that SVM and kNN are at least comparable to other well-known classification methods, including Neural Networks and Naive Bayes, and that significantly outperform the other methods when the number of positive training instances per category are small.

In this paper we propose a new approach to the subject of Twitter Topic Detection when in the presence of a large number of Portuguese trending topics: the use of and adaptation of the Fuzzy Fingerprints introduced in [8], associated with the use of Filtered Space Saving algorithm [7][9] and the fuzzy based automatic error correction mechanism presented in

$$A_{m,n} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Figure 1 Binary Bag-of-Words Representation.

[2]. This work is integrated within the MISNIS framework, being developed with the goal of Intelligent Mining of Public Social Networks' Influence in Society [8]. We present some preliminary results that show that the adapted Fuzzy Fingerprints outperform some of the most commonly classifiers (SVM and kNN) when applied to this particular problem. Additionally, in conjunction with the other techniques, the proposed Twitter topic detection process has additional advantages over the existing methods. In particular, it is significantly faster, and the resulting models are much smaller than SVMs.

The paper is organized as follows: First, we discuss several "Related Techniques" from similar fields of study such as Text Categorization and Document Representation. Secondly, we explain how our proposed method (Twitter Topic Fuzzy Fingerprints) works and how it stems from the Author Fuzzy Fingerprint in [8]. Then, we present the characteristics of the used data set and how evaluations were performed. Finally we evaluate the Twitter Topic Fuzzy Fingerprints with our data set of Portuguese tweets and present the comparison results to SVM and kNN.

2 Related Techniques

The goal of this work is essentially to automatically classify tweets into a set of trending topics. This process is broadly known in Natural Language Processing (NLP) as Text Categorization, and consists of finding the correct topic (or topics) for each document, given a set of categories (subjects, topics) and a collection of text documents [5].

The text contained in each tweet is the most relevant source of information for classification. However, text is an unstructured form of data that classifiers and learning algorithms cannot directly process [5]. For that reason, our documents/tweets must be converted into a more manageable form, during a preprocessing step.

2.1 Document Representation

One of the simplest and commonly used representation is the *bag-of-words* model. Frequently used in NLP and Information Retrieval (IR), it consists of representing a document as a set (bag) of its words, ignoring the syntax and even the word order, but keeping the frequency of each word. It uses all words in a document as features. Thus, the dimension of the feature space is equal to the number of different words in all documents [5]. This form of representation can be illustrated with the following example:

- John bought a car
- I love driving my car
- I love John

Based of these three texts, a dictionary of unique words can be constructed: {John, bought, a, car, I, love, driving, my}. The text collection can then be represented as a binary matrix, containing 8 columns, one per dictionary word, and 3 rows, one per text entry:

The matrix presented in Figure 1 indicates whether a given term exists in the document, without detailing on its importance to the collection of documents. An extended representation is known as the TF-IDF scheme and combines the concept of the term frequency with the inverse document frequency. The TF-IDF scheme is a scoring method that can tell the importance of a word in a collection of documents, and can be calculated as a simple multiplication:

$$tfidf = tf \times idf\,. \tag{1}$$

The concept of term frequency (tf) is simply the number of occurrences of the word in the document. The more common the word, the higher the term frequency will be. On the other hand, words that occur in few documents, are probably richer in details that could better characterize the document. The inverse document frequency (idf) spans from the principle that a word that occurs in many documents is not relevant in differing each document from each other. idf can be obtained by dividing the total number of documents N by the number of documents n_i containing the term, and then taking the logarithm of that quotient, as expressed in Eq. (2):

$$idf = \log \frac{N}{n_i}\,. \tag{2}$$

By combining the Eqs. (1) and (2), the TF-IDF of word in a document can be expressed by Eq. (3):

$$tfidf_i = tf \times \log \frac{N}{n_i}\,. \tag{3}$$

As succinctly explained in [17], *tfidf* assigns a weight to a term in document that is:

1. highest when the term occurs many times within a small number of documents;
2. lower when the term occurs fewer times in a document, or occurs in many documents;
3. lowest when the term occurs in virtually all documents;

Different categorization methods can be applied to a structured document representation. In general, categorization algorithms follow the following four steps [5]:

1. Decide the categories that will be used to classify the instances;
2. Provide a training set for each of the categories;
3. Decide on the features that represent each of the instances;
4. Choose the algorithm to be used for the categorization;

2.2 k-Nearest Neighbors Algorithm – kNN

The kNN is an example-based classifier. This means it will not "build explicit declarative representations of categories, but instead rely on computing the similarity between the document to be classified and the training documents" [5]. In this case, the training data is simply the "storing of the representations of the training documents together with their category labels".

In order for kNN to "decide whether a document d belongs to a category c, kNN checks whether the k training documents most similar to d belong to c. If the answer is positive for a sufficiently large proportion of them, a positive decision is made."

In Figure 2, there are 3 different categories: blue, orange and green. The yellow dot represents the document to be categorized. If $k = 1$ (smaller circle), the document will look for its closest neighbour and determine that it belongs to the blue category, therefore, it

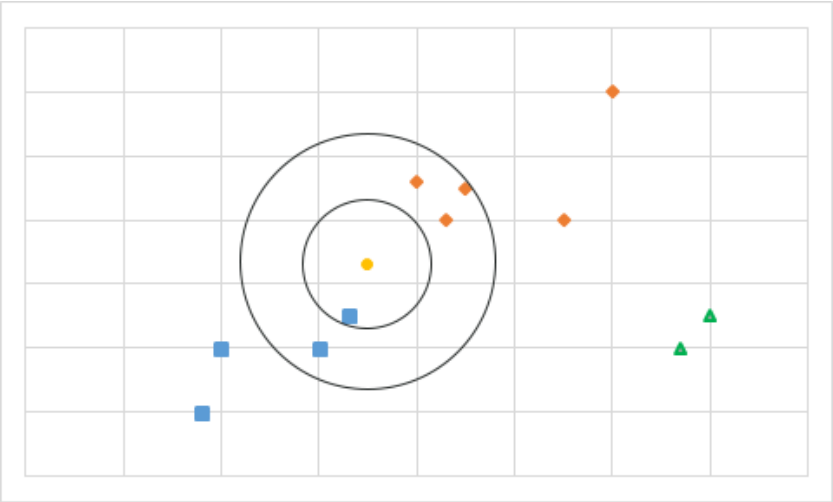


Figure 2 Example of the K Nearest Neighbour Algorithm.

will be classified as blue. However, if $k = 5$, (larger circle), it will determine that 3 of its neighbours belong to the orange category and 2 to the blue category. By a majority rule, the document will be classified as orange.

An appropriate value of k is of the utmost importance. While $k = 1$ can be too simplistic, as the decision is made according only to the nearest neighbour, a high value of k can have too much noise in it and favour dominant categories. In fact, this algorithm is known to be affected by noisy data.

The kNN is considered to be one of the simplest and best performing text classifiers, whose main drawback is "the relatively high computational cost of classification – that is, for each test document, its similarity to all of the training documents must be computed" [5]. In kNN, "the training is fast, but classification is slow. Computing all the similarities between a document that has not been categorized and a collection of documents, is slow" [12].

2.3 Support Vector Machines – SVM

A support vector machine (SVM) is a very fast and effective binary classifier. According to [12] "every category has a separate classifier and documents are individually matched against each category". Given the vector space model in which this method operates, geometrically speaking, [5] describes SVM as a "hyperplane in the feature space, separating the points that represent the positive instances of the category from the points that represent the negative instances. The classifying hyperplane is chosen during training as the unique hyperplane that separates the known positive instances from the known negative instances with the maximal margin".

Consider Figure 3 as a two dimensional example of SVM. As one would expect, in this scenario, the hyperplanes are lines. The figure reveals that the hyperplane H_1 does not separate the positive from the negative instances. H_2 does, but it does not guarantee the maximum distance between them. Finally, H_3 offers the necessary solution. "It is interesting to note that SVM hyperplanes are fully determined by a relatively small subset of the training instances, which are called the support vectors" [5].

According to [12], SVM has at least three major differences with the previous categorization method:

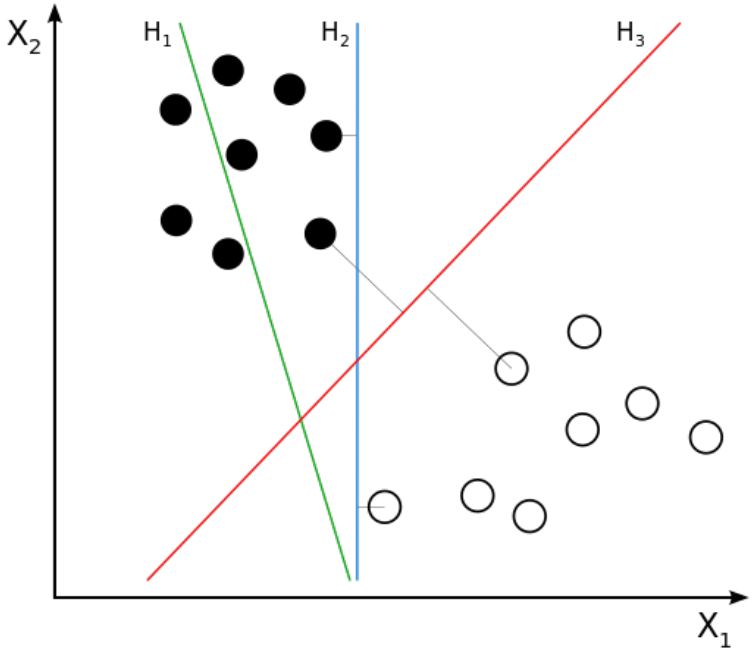


Figure 3 Two dimensional Support Vector Machine.

1. Not all training documents are used. The SVM function is built only by documents near to the classification border;
2. An SVM can construct an irregular border to separate positive and negative training documents;
3. Not all features (unique words) from training documents are necessary for classification;

SVM methods for text categorization have recently attracted some attention since they are amongst the most accurate classifiers [12].

3 Twitter Topic Fuzzy Fingerprints

In this work we propose the use of an adaptation of the Fuzzy Fingerprints classification method described in [8] to tackle the problem of Topic Detection in Twitter. In [8] the authors approach the problem of text authorship by using the crime scene fingerprint analogy to claim that a given text has its authors writing style embedded in it. If the fingerprint is known, then it is possible to identify whether a text whose author is unknown, has a known author's fingerprint on it.

The algorithm itself works as following:

1. Gather the top-k word frequencies in all known texts of each known author;
2. Build the fingerprint by applying a fuzzifying function to the top-k list. The fuzzified fingerprint is based on the word order and not on the frequency value;
3. Perform the same calculations for the text being identified and then compare the obtained text fuzzy fingerprint with all available author fuzzy fingerprints. The most similar fingerprint is chosen and the text is assigned to the fingerprint author;

The proposed fuzzy fingerprint method for Tweet Topic Detection, while similar in intention and form, differs in a few crucial steps.

Listing 1 Pseudo-Code to explain data structure used.

```
createDataStructure(trainingSet,topTrends)
        trendFP = Set of topicFingerprints()
        for tweet in trainingSet
                tokens = tokenize(tweet)
                kw = words in tokens and topTrends
                for k in kw
                        for t in tokens
                                if t not in topTrends
                                        trendFP{k}[t]++
```

First it is important to establish the parallel between the context of author ownership and Tweet Topic Detection. Instead of author fingerprints, in this work we are looking to obtain the fingerprints of hashtagged Twitter topics (#). Once we have a topics fingerprint library, each unclassified tweet can be processed and compared to the fingerprints existing in the topic library.

Secondly, different criteria were used in selecting the top-k words for the fingerprint. While [8] uses word frequency as the main feature to create the top-k list, here we use an adaptation of an Inverse Document Frequency technique, aiming reducing the importance of frequent terms that are common across several topics, such as "follow", "RT" and "like".

Lastly, the similarity score differs from the original, based on the fact that tweets are, by design, very short texts, while the original Fuzzy Fingerprint method was devised to classify much longer texts (newspaper articles, books, etc. ranging from thousands to millions of characters). Here we propose the use of a normalized score with values between 0 and 1, where the lowest score indicates that the tweet in question is in no way similar to the topic fingerprint, and the highest value indicates that the tweet is totally similar.

3.1 Building the Fingerprint Library

In order to build the fingerprint library, the proposed method goes over the training set, which, in this situation, are tweets containing the Trending Topics of the day. For each tweet, it acknowledges the existence of the # and adds each word in the tweet to a #topic table alongside with its counter of occurrences. Only the top-k most frequent words are considered. The algorithm presented in Figure 1 presents further details the this process.

The main difference between the original method and ours, is that due to the small size of each tweet, its words should be as unique as possible in order to make the fingerprints distinguishable amongst the various topics. Therefore, in addition to counting each word occurrence, we also account for of its Inverse Topic Frequency (ITF), an adaptation of the Inverse Document Frequency in Eq. (2), where N becomes the topic fingerprint library size (i.e., the total number of topics), and n_i becomes the number of #topics where the word is present.

Table 1 shows an example of a possible top-k output produced by the algorithm Figure 1 for a fingerprint size $k = 3$, after going through a small training set. By multiplying the occurrences of each word per topic with its ITF, we obtain the third column of Table 1. As expected, the term "help", which was the only one that occurred in more than one fingerprint, got dropped to last position in the ranking of fingerprint words for the topic "#derek".

After obtaining the top-k list for a given #topic, we take the same approach as the original method, and use the membership Eq. (4) to build the fingerprint, where k is the size

Table 1 Fingerprint hash table before and after ITF.

Key	Feature	Counter	Feature	ITF
#michaeljackson	dead	4	dead	1.90
	rip	2	rip	0.95
	sing	1	sing	0.48
#haiti	earthquake	10	earthquake	4.77
	rip	5	rip	1.43
	help	1	help	0.17
#derek	show	8	show	3.81
	help	3	australia	0.95
	australia	2	help	0.52

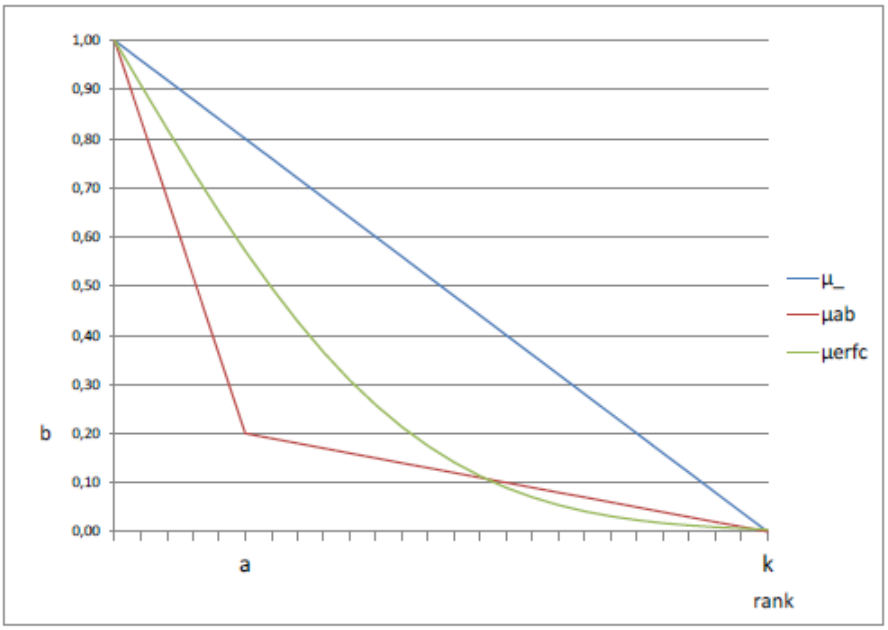


Figure 4 Fuzzyfing functions.

of the top-k fingerprint and i represents the membership index:

$$\mu_{ab}(i) = \begin{cases} 1 - (1-b)\frac{i}{kb} & i < a \\ \frac{a(1-\frac{i-a}{k-a})}{k} & i \geq a \end{cases} \tag{4}$$

Figure 4 shows the three membership functions that were considered and the impact of the parameters a and b on it. Much like in the original method, Eq. (4) was the best performing function and thus, the chosen one.

The fingerprint is a k sized bi-dimensional array containing in the first column the list of the top-k words, and in the second column its membership value $\mu_{ab}(i)$ obtained by the application of Eq. (4).

3.2 Tweet-Topic Similarity Score

In the original method, Eq. (4), in order to check the authorship of a given text, a fingerprint would be built for the document (using the procedure described above), and then the document fingerprint would be compared with each fingerprint present in the library. Within the Twitter context, such approach would not work due to the very small number of words

contained in one tweet – it simply does not make sense to count the number of individual word occurrences. Therefore we developed a Tweet-Topic Similarity Score (T2S2) that tests how much a tweet fits to a given topic. The T2S2 function, Eq. (5), provides a normalized value ranging between 0 and 1, that takes into account the size of the (preprocessed) tweet (i.e., its number of features):

$$T2S2(\Phi, T) = \frac{\sum_{v} \mu_{\Phi}(v) : v \in (\Phi \cap T)}{\sum_{i=0}^{j} \mu_{\Phi}(w_i)} \tag{5}$$

In (5) Φ is the #topic fingerprint, T is the set of words of the (preprocessed) tweet, $\mu_{\Phi}(v)$ is the membership degree of word v in the topic fingerprint, and j is the is the number of features of the tweet. Essentially, T2S2 divides the sum of the membership values $\mu_{\Phi}(v)$ of every word v that is common between the tweet and the #topic fingerprint, by the sum of the top j membership values in $\mu_{Phi}(w_i)$ where $w \in (\Phi)$. Eq. 5 will tend to 1.0 when most to all features of the tweet belong to the top words of the fingerprint, and tend to 0.0 when none or very few features of the tweet belong to the bottom words of the fingerprint.

4 Twitter Data

Using Twitter's developer tools [21], we extracted samples of the public data flowing through Twitter by establishing a connection to a Twitter streaming endpoint. It is important to note that, using the sample API, only 1% of the actual public tweets can be retrieved [4]. Using this method, we obtained just over 1.2 million Portuguese tweets, from March, 14th to March 20th, 2014.

By executing Twitters DEV "GET Trends/place" method, one can obtain the top 10 trending topics of the moment in a given place. Using the WOEID (Where On Earth ID) for Brazil and Portugal, we extracted the top trends on the 17th of March mid-afternoon, and among them we selected two topics that seemed to have the most interesting content:

- #AnittaNarizDeCapivara, regarding Anitta's (Brazilian singer) new nose job which made an impact during the annual award show "Melhores do Ano";
- #FicaVanessa, for people supporting Big Brother's Brazil participant Vanessa who was at risk of leaving the show;

Despite being top trends, we found that these hashtags only occurred 289 and 822 times in our whole set of 1.2 million tweets. This can be explained by Twitter's view on what constitutes a trending topic.

According to [22], "Twitter Trends are automatically generated by an algorithm that attempts to identify topics that are being talked about more right now than they were previously. The Trends list is designed to help people discover the most breaking news from across the world, in real-time. The Trends list captures the hottest emerging topics, not just what is most popular. Put another way, Twitter favours novelty over popularity".

4.1 Training Data Set

Even though we only used 2 topics for testing purposes (due to the difficulty in annotating a high number of topics), the training set was built using 100 different topics created after the most popular hashtags on the database. The training set is composed of over 600,000 tweets

in Portuguese language, where the most popular trend is #kca (18,000 tweets) and the rarer is #1dnamix (139 tweets).

The fact that not all categories are trained with the same amount of samples makes for what is known as an unbalanced dataset. In this case, it may happen that one single category dominates the training set in such fashion, that some classifiers will incorrectly categorize most of the test set as belonging to that one category.

4.2 Test Data Set

The test set was impartially built from uncategorized tweets belonging to the original set of 1.2 million documents.

Because of the TV broadcasting nature of the two target trends (#AnittaNarizDeCapivara and #FicaVanessa), where the owners of the trends encourage its use and propagation, it was very difficult to find many uncategorized tweets that clearly belonged to those topics. Only 82 and 43 respectively were annotated, i.e., manually assigned to one of the two target trends despite not the trend itself in the tweets' text.

In order to increase the size of test set, a few more uncategorized tweets were added to it, making for a total of 210 samples. Whilst still short, it provides a chance for our algorithm to detect negative scenarios efficiently, since the added tweets belong to untrained top trends.

5 Evaluation Metrics

In this section, we take a look at the metrics used to determine how good or poorly a classifier performs. Typically there are three key concepts: Precision, Recall and F-Measure.

Before the formulas are presented, it is important to grasp the statistical definitions that constitute those formulas, within the scope of Twitter topic detection:

1. True Positive (TP): This means that a tweet belonging to a given topic, has been correctly identified as belonging to that topic;
2. False Positive (FP): This means that a tweet that does not belong to a given topic, has been incorrectly identified as belonging to that topic;
3. True Negative (TN): This means that a tweet that does not belong to a given topic, has been correctly identified as not belonging to that topic;
4. False Negative (FN): This means that a tweet belonging to a given topic, has been incorrectly identified as not belonging to that topic;

With this in mind, the definition of the metrics are:

$$Precision = \frac{\#TP}{\#TP + \#FP} \tag{6}$$

$$Recall = \frac{\#TP}{\#TP + \#FN} \tag{7}$$

$$F\text{-}Measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{8}$$

6 Results

Here we compare the Twitter Topic Fuzzy Fingerprint method up against the two algorithms we presented earlier: k-Nearest Neighbour (kNN) and Support Vector Machine (SVM). The

exact same training data sets and test data sets were used for all methods. Several test scenarios were built to find each algorithm optimal performance setting.

6.1 Twitter Topic Fuzzy Fingerprint Performance

The following parameters were considered for the Twitter Topic Fuzzy Fingerprint:

1. k, size of the fuzzy fingerprint. Several increasing values were taken into to account, in order to determine whether a higher or lower k value would provide better results;
2. *stopwords*. For each scenario, the results provided were measured with and without the removal of stopwords. This aims to ascertain the true impact of the removal of stopwords. The stopword list was provided by the Natural Language Toolkit, [1];
3. *stemming*. For each scenario, the option to return words in their stem form can be either turned on or off. With this parameter, we aim to determine the impact of this preprocessing technique towards getting better results.
4. *minimum j sized words*. For each scenario, different values of j were considered as being the minimum size of the words to feature in the tweets' list of terms. The purpose of this variable, is to test how the removal of small words may help keep richer tokens and get better results;
5. *threshold value*. It represents the T2S2 value from which our method will declare that a certain tweet belongs to a given trend. For the purpose of this work, values of 0.5, 0.25, 0.15 and 0.10 were tested;

Through extensive testing, we found that the best results for the Twitter Topic Fuzzy Fingerprints Algorithm were achieved when:

- considering a low threshold value for acceptance of a tweet belonging to a topic (T2S2 = 0.10);
- configuring a value of $k = 40$ for the size of the list of the fingerprint;
- removing short words from the corpus, only keeping words with a minimum length of 4 characters($j = 4$);
- removing stopwords from the corpus;
- not performing Stemming operations;

While the removal of stopwords provided better results (approximately 2%), the stemming technique provided literally no improvement. A possible explanation for this, may derive from the nature of language in Twitter itself. Since tweets are short in nature, words may often occur in their stem form or in such fashion that a formal stemmer cannot process. Twitter's lingo is very unique due to the informal nature of social networking communication, and a Stemming algorithm can only truly be effective with formal and well written texts.

Table 2 summarizes the algorithm's performance. Regardless of the value of k, precision values are always high, which indicates that False Positive scenarios are rare or non-existent. However, recall is low for small values of $k = [5; 10; 15]$ peaking at $k = 40$, when no False Negative scenarios are identified.

A low recall is also a consequence of typically small T2S2 similarity values, which means that Positive cases are not being identified because they were below the threshold= 0.10. Consider the example of a preprocessed tweet with 8 features, two of which match a given fingerprint: even if the matching words are the highest ranked membership terms, T2S2 would score approximately under $\frac{2}{8} = 0.25$.

Table 2 Twitter Topic Fuzzy Fingerprint Performance, with stopword removal but no stemming.

j	k	Precision	Recall	F-Measure
4	5	1.000	0.492	0.659
4	10	1.000	0.621	0.766
4	15	1.000	0.694	0.819
4	20	1.000	0.815	0.898
4	25	1.000	0.952	0.975
4	30	1.000	0.976	0.988
4	40	0.992	1.000	**0.996**
4	50	0.992	1.000	0.996
4	75	1.000	0.976	0.988
4	100	1.000	0.968	0.984
4	150	1.000	0.968	0.984
4	250	1.000	0.976	0.988
4	500	1.000	0.976	0.988

As k increases, either more matching words between the tweet features and the fingerprint are found, or the same ranked words have a higher membership value which encourages a better T2S2 score.

The best case scenario scores f-measure= 0.996, which, while extremely positive, is suspicious due to the fact that the data set is short and possibly over trained. In [18], the exact same Twitter Fuzzy Fingerprints method reached f-measure= 0.840 for a larger multi-language data set and with more target top trends (35).

Here we tested for 2 target topics out of the possible 100 training trends. The purpose behind this approach was to study how the Twitter Fuzzy Fingerprints method would behave when approaching a larger and more realistic number of different possible trends, and to study the impact of using the Inverse Topic Frequency (ITF) which should theoretically improve the results with the increase in the number of topics. On the other hand, this experiment also shows how the competing techniques are not as scalable: in [18], kNN also performed poorly, but the SVM f-measure= 0.79 was very close to the one obtained using the Twitter Fuzzy Fingerprints, albeit much less efficient in what concerns execution time. Here the performance of the competing techniques degrade to the point of being unusable.

6.2 *k*NN and SVM Performance

In order to test kNN and SVM, stopwords were removed but stemming was not performed. The final representation of either training and test set is a bag-of-words type, Figure 1, with TF-IDF weighting, Eq. (3). The tests were performed using the WEKA framework [6].

kNN was executed with the $k = [3, 5, 10, 30, 50]$ and, due to the limitations of WEKA, the distance measure considered was the Euclidean distance as opposed to the more common cosine similarity. Despite extensive testing and parameter tunning, the algorithm was incapable of identifying a single True Positive case, i.e., precision= 0, recall= 0, f-measure= 0. Instead, it classified all the samples as a part of the majority trained class #kca, which is non-existent in the test set. In [18], when $k = 5$, an f-measure= 0.445 could be achieved, despite also being a consequence of classifying all test tweets in the majority class.

For SVM, a number of different parameters were tested and optimized, but [18] suggests that the best performance was achieved using a linear kernel and a small soft-margin value: $C = 0.01$, providing competitive results with the Twitter Fuzzy Fingerprints method.

For our test set of 210 Portuguese tweets, SVM behaved exactly as kNN did, i.e., precision= 0, recall= 0, f-measure= 0.

There are three possible explanations for such a poor performance. Firstly, this is a very

specific problem to which such broad and well known algorithms may not apply. Secondly is the fact that there were so many different classes for either method to train. In addition, the bag-of-words representation of the test data set was a very sparse matrix, with 210 documents (lines) and over 69000 unique features (columns). This would make these space-vector oriented algorithms highly ineffective. Finally, there is the unbalanced nature of the training data set, as explained by Zang and Inderjeet in [24]. When dealing with unbalanced data sets, kNN completely ignores the minority classes and will often mistakenly classify a tweet to the majority category.

6.3 Method Comparison

When comparing all 3 methods, it is evident that the Twitter Topic Fuzzy Fingerprint algorithm outperforms both kNN and SVM, in this particular case.

Table 3 Execution Speed Comparison.

Method	Preprocessing	Build Model	Evaluate
Fuzzy Fp	124.0s		0.02s
kNN	$>$ 1800s	0.02s	89.7s
SVM	$>$ 1800s	$>$ 14400s	

kNN is a lazy algorithm, where all computation is deferred until classification, which justifies that it takes so much longer evaluating such a small test data set. In what concerns to SVM, a significant part of the time is attributed to creating the model after the preprocessing stage. In our approach, building the model is just a linear function of the number of words being considered for training.

Finally, the size of the model is also a significant issue, specially when one aims at processing big quantities of data, in a distributed fashion. The fuzzy fingerprint model for a given topic corresponds to a vector of k fixed elements, each one containing the value of a feature (e.g. the word) and its score. Therefore it is fixed in size and corresponds to prunning the list of relevant words at k.

7 Conclusions and Future work

We proposed a method for topic detection for micro bloging content, such as Twittter, when in the presence of a large number of trending topics. This method, known as Twitter Topic Fuzzy Fingerprints, outperforms two other commonly used algorithms, kNN and SVM.

Even when ignoring the obtained f-measure results, which can be biased by the size of the test dataset, the proposed method still presents several advantages that make it an undeniable alternative for on-the-fly, and possibly distributed, topic detection: 1) the size of the resulting model is significantly smaller than SVM models, which is an important issue to consider when performing distributed computation in different machines; 2) the classification is significantly faster than the other two methods, making it an interesting solution for on-the-fly processing of Big Data streams.

Since the annotated Portuguese data used in the paper is undeniably small, extended manually annotated test sets must be created in a near future in order to further confirm and validate the results here presented.

Acknowledgments. This work was supported by national funds through FCT Fundação para a Ciência e a Tecnologia, under project PTDC/IVC-ESCT/4919/2012 and project PEst-OE/EEI/LA0021/2013.

References

1 Steven Bird. NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, COLING-ACL'06, pages 69–72, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

2 J. P. Carvalho and L. Coheur. Introducing UWS – a fuzzy based word similarity function with good discrimination capability: Preliminary results. In *FUZZ-IEEE*, pages 1–8, 2013.

3 Mario Cataldi, Luigi Di Caro, and Claudio Schifanella. Emerging topic detection on Twitter based on temporal and social terms evaluation. In *Proceedings of the Tenth International Workshop on Multimedia Data Mining*, MDMKDD'10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

4 Sunil D. M. *et al.* Twitter developers – limit on streaming tweets. `https://dev.twitter.com/discussions/6789`. Accessed: 2014-03-28.

5 Ronen Feldman and James Sanger. *Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge University Press, New York, NY, USA, 2006.

6 Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

7 N. Homem and J. P. Carvalho. Finding top-k elements in data streams. *Inf. Sci.*, 180(24):4958–4974, December 2010.

8 N. Homem and J. P. Carvalho. Authorship identification and author fuzzy fingerprints. In *30th Annual Conference of the North American Fuzzy Information Processing Society*, NAFIPS2011, 2011.

9 N. Homem and J. P. Carvalho. Finding top-k elements in a time-sliding window. *Evolving Systems*, 2(1):51–70, 2011.

10 Carol Huang. Facebook and Twitter key to Arab Spring uprisings: report. *The National*, 6 June 2011. `http://www.thenational.ae/news/uae-news/facebook-and-twitter-key-to-arab-spring-uprisings-report`.

11 Shiva Prasad Kasiviswanathan, Prem Melville, Arindam Banerjee, and Vikas Sindhwani. Emerging topic detection using dictionary learning. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM'11, pages 745–754, New York, NY, USA, 2011. ACM.

12 Manu Konchady. *Text Mining Application Programming*. Charles River Media, 2006.

13 Kathy Lee, Diana Palsetia, Ramanathan Narayanan, Md. Mostofa Ali Patwary, Ankit Agrawal, and Alok Choudhary. Twitter trending topic classification. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining Workshops*, ICDMW'11, pages 251–258, Washington, DC, USA, 2011. IEEE Computer Society.

14 Michael Mathioudakis and Nick Koudas. Twittermonitor: Trend detection over the twitter stream. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 1155–1158, New York, NY, USA, 2010. ACM.

15 Allie Mazzia and James Juett. Suggesting hashtags on twitter. Master's thesis, University of Michigan, 2010.

16 El País. El 15-M sacude el sistema. `http://politica.elpais.com/politica/2011/05/21/actualidad/1305999838_462379.html`, May 2011.

17 Anand Rajaraman, Juri Leskovec, and Jeffrey Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011.

18 H. Rosa, J. P. Carvalho, and F. Batista. Twitter topic fuzzy fingerprints. *IEEE International Conference on Fuzzy Systems (FUZZ-IEEE 2014)*, 2014.

19 Ankan Saha and Vikas Sindhwani. Learning evolving and emerging topics in social media: A dynamic nmf approach with temporal regularization. In *Proceedings of the Fifth ACM*

International Conference on Web Search and Data Mining, WSDM'12, pages 693–702, New York, NY, USA, 2012. ACM.

20 Jagan Sankaranarayanan, Hanan Samet, Benjamin E. Teitler, Michael D. Lieberman, and Jon Sperling. Twitterstand: News in tweets. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS'09, pages 42–51, New York, NY, USA, 2009. ACM.

21 Twitter. Twitter developer tools. `https://dev.twitter.com/`. Accessed: 2014-03-28.

22 Twitter. To trend or not to trend... `https://blog.twitter.com/2010/trend-or-not-trend`, 8 December 2010.

23 Yiming Yang and Xin Liu. A re-examination of text categorization methods. In *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'99, pages 42–49, New York, NY, USA, 1999. ACM.

24 J. Zhang and I. Mani. KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction. In *Proceedings of the ICML'2003 Workshop on Learning from Imbalanced Datasets*, 2003.

Multiscale Parameter Tuning of a Semantic Relatedness Algorithm

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

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

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

Abstract

The research presented in this paper builds on previous work that lead to the definition of a family of semantic relatedness algorithms that compute a proximity given as input a pair of concept labels. The algorithms depends on a semantic graph, provided as RDF data, and on a particular set of weights assigned to the properties of RDF statements (types of arcs in the RDF graph). The current research objective is to automatically tune the weights for a given graph in order to increase the proximity quality. The quality of a semantic relatedness method is usually measured against a benchmark data set. The results produced by the method are compared with those on the benchmark using the Spearman's rank coefficient. This methodology works the other way round and uses this coefficient to tune the proximity weights. The tuning process is controlled by a genetic algorithm using the Spearman's rank coefficient as the fitness function. The genetic algorithm has its own set of parameters which also need to be tuned. Bootstrapping is based on a statistical method for generating samples that is used in this methodology to enable a large number of repetitions of the genetic algorithm, exploring the results of alternative parameter settings. This approach raises several technical challenges due to its computational complexity. This paper provides details on the techniques used to speedup this process. The proposed approach was validated with the WordNet 2.0 and the WordSim-353 data set. Several ranges of parameters values were tested and the obtained results are better than the state of the art methods for computing semantic relatedness using the WordNet 2.0, with the advantage of not requiring any domain knowledge of the ontological graph.

1998 ACM Subject Classification E.1 Graphs and networks, G.2.2 Graph theory, Path and circuit problems, H.3.1 Content Analysis and Indexing, I.2.4 Knowledge Representation Formalisms and Methods, Semantic networks, I.2.8 Problem Solving, Control Methods, and Search, Graph and tree search strategies

Keywords and phrases semantic similarity, linked data, genetic algorithms, bootstrapping, WordNet

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.201

1 Introduction

Consider a magazine, a pencil and a notepad. Of these three items which is the most related pair? Is it magazine and pencil, pencil and notepad, or newspaper and notepad? People living in more individualistic societies tend to find the magazine and the notepad more related, since they are both made of sheets of paper; while people living in more collectivist societies tend to find the pencil and the notepad more related, since they complement each

© José Paulo Leal and Teresa Costa;
licensed under Creative Commons License CC-BY

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 201–213

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

other (a pencil writes on notepad) [7]. The differences are even more striking when people are asked to assign a value to the relatedness [6]. These experiments reveal the lack of a standard definition of relatedness and the difficulty to measure the relatedness of two concepts.

A standard approach to measure relatedness is to use an ontology [13]. An ontology is a formal and explicit specification of the relationships between concepts. For instance, a thesaurus is a kind of ontology specifying the relationships between words: synonymy and antonymy, as well as hyponymy (words whose semantic field encloses other words, e.g., mammal has as hyponym horse) and hypernymy (words whose semantic field is enclosed in other words, e.g, horse has a hypernym mammal).

This paper presents ongoing work aiming at the development of a new methodology to determine the semantic relatedness between two concepts. This methodology is ontology based, can be applied to an ontological graph, and does not require any knowledge of the ontological domain. It uses a family of semantic relatedness algorithms based on the notion of proximity [10]. An algorithm of this family is parametrized by a semantic graph and a set of weights. The semantic graph is provided as RDF data, where the resources are the graph nodes and the properties are the arcs. Each type of arc has a specific weight value. Tuning these weights in order to improve the quality of the semantic relatedness is the current objective of this research.

Other methods available in the literature [1, 11, 13] measure the quality of their algorithms using as benchmark a standard data set [6]. The reference similarity of concept pairs is the average similarity assigned by a group of persons. The relatedness computed with an algorithm is compared against those of the benchmark using the Spearman's rank order correlation. The quality of an algorithm is as high as the value of this correlation.

A measure of quality is essential for using a genetic algorithm to tune weight values. In this tuning approach, an assignment of values to weights is encoded as a set of genes of a chromosome. New chromosomes are obtained by crossover and mutation of the chromosomes from the previous generation and the best are selected using a fitness function plus randomness. The fitness function receives as input a weight assignment and returns the Spearman's rank order correlation for a subset of benchmark data.

The genetic algorithm has in turn its own set of parameters that need to be tuned. Bootstrapping is done through a statistical procedure that produces a large number of samples that is used to explore the most promising settings of the genetic algorithm. The best settings are finally used to run the genetic algorithm a large number of times with the complete data set.

This methodology was validated with the ontology of WordNet 2.0 [5], using as benchmark the WordSim-353 [6] data set. The obtained results were better than the best results available on the literature for the same ontology and benchmark [13, 9].

Due to its computational complexity this tuning methodology raises several technical challenges. Firstly, the semantic algorithms must collect a large number of paths connecting each pair of labels in the graph. Secondly, in order to compute the Spearman's rank order, the semantic relatedness algorithm must be executed with several hundreds of pairs of concept labels. Thirdly, the genetic algorithm must compute a correlation for each chromosome (a set of weight assignments) in the genetic pool for hundreds of generations. And finally, the evolution process of the genetic algorithm has to be repeated hundreds of times as part of the bootstrapping method. This paper presents also approaches used to speedup the tuning process.

The rest of the paper is organized as follows. The next section present the state of the art on semantic relatedness. Section 3 describes the tuning methodology and Section 4 details

its implementation. The experimental results and their analysis can be found in Section 5. Finally, Section 6 summarizes this work and identifies opportunities for further research.

2 Related Work

The problem of computing semantic relatedness can be approached in several ways. Most approaches fall in one of two types: path methods, based on the topology of the relationships between concepts; and content methods, based on the frequency of word occurrence in corpora. In many cases the paths relating the concepts traverse an ontology. The research described in this paper follows the ontological approach.

Some of the ontological methods use only the underlying taxonomy, for instance, the taxonomy created by the *is-a* relationships, or the hypernymy and hyponymy relationships of a thesaurus. An example of this kind of approach is the work of Mazuel and Sobouret [11]. Their approach measures the relatedness based on the taxonomical part of the ontology of the WordNet and discards paths that are not "semantically correct" working only with a subset of "semantically correct" paths. To measure the semantic distance this methodology selects the best one from the subset.

Other ontological methods explore the full range of relationships in an ontology. An example of this approach is the work of Hirst and St-Onge [8] that used the WordNet as a knowledge source to create a lexical *chainer* (SIC). A lexical chain is a chain where words are included if they have a cohesive relationship with another word already in the chain. In this work they defined three types of relations: extra-strong, strong and medium-strong. The weight of a relation is higher as stronger is the relationship between the words.

Some path approaches use also statistical concepts. J. Garcia and E. Mena [2] developed a method that uses the Web as knowledge source, based on the Normalized Google Distance. This approach uses the frequencies of concepts provided by search engines to define a new semantic relatedness measure among ontology terms.

The work of Michael Strube and Simone Paolo Ponzetto [13] analyses several path and content approaches to choose the best one. The approaches they analysed were assigned to three categories: path, content and text overlap. The approaches in this last category compute an overlap score by using stemming to explore related words.

Several of the mentioned approaches use the WordNet. The WordNet [5] is a large lexical knowledge base of English words. It groups nouns, verbs, adjectives and adverbs into *synsets* (sets of cognitive synonyms) that express distinct concepts. *Synsets* are interlinked by lexical and conceptual-semantic relationships. This knowledge base is well-known and widely used but lacks some specialized vocabularies and named entities, such as Diego Maradona or Freddie Mercury. On the other hand, it is a comparatively small knowledge base and thus it is ideal for the initial tests of a tuning methodology.

3 Multiscale Weight Tuning

This section is to describe an approach for tuning weights in a family of semantic relatedness algorithms. The proposed approach for tuning weights operates at different scales. Although each scale has its own distinctive features, there are self-similar patterns common to all scales.

Consider a fractal as a metaphor. At each scale a fractal exhibits features that are found also on other fractal dimensions. That is, if we zoom in (or zoom out) on a fractal we observe a identical pattern. Mathematical fractals are exact and infinite repetitions of the same

pattern. Nonetheless, fractals observed in nature, such as shells, leaves or coastlines, exhibit self-similar patterns that are neither infinite nor an exact repetition of other scales.

In this tuning approach, the common pattern is the concept of function, with input and output values, and a set of parameters that can be tuned. At the lowest scale this function is the semantic relatedness algorithm. It takes as input a pair of strings and produces a value in the interval [0,1]. This function takes as parameters a semantic graph and a set of weights that must be tuned.

Zooming out to the next scale there is a genetic algorithm. A genetic algorithm can be seen as a function taking another function as input a fitness function and producing a result. It has also its own set of parameters that need to be tuned: the number of generations, the mutation rate, etc.. In this case the fitness function takes as input a set of weights and computes the correlation between the relatedness obtained by the algorithm and the standard benchmark. Hence, each application of a genetic algorithm (seen as a function) aggregates thousands of applications of functions from the previous scale – the semantic relatedness algorithm.

Continuing to zoom out to the next and final scale there is a statistical method, the bootstrapping method. This method measures the accuracy of the results obtained by the genetic algorithm with different parameter sets. It can also be seen as a function taking as input genetic algorithms, the candidates for producing a weight tuning, and producing an estimate of which is the best. Again, each application of the bootstrapping method (seen as a function) aggregates thousands of applications of the of functions from the previous scale – the genetic algorithm – since each candidate configuration set is repeated hundreds of times.

The following subsections detail each of these "fractal scales". Each scale describes the function that is used as input for its upper scale, identifying its parameters. At least metaphorically, these functions can be seen as a self-similar pattern that is present in the three different layers in which the proposed approach operates.

3.1 Proximity Measure Layer

The core of the methodology for calculate proximity between concepts is an algorithm to compute semantic relatedness using ontological information in RDF graphs. It uses the notion of proximity, rather than distance, as the underlying concept for computing semantic relatedness between two nodes.

Concepts in ontological graphs are represented by nodes. Take for instance the music domain. Singers, bands, music genres, instruments or virtually any concept related to music is represented as nodes in an ontology. These nodes are related by properties, such as `has genre` connecting singers to genres, and thus form a graph. This graph can be retrieved in RDF format using the SPARQL endpoint of a knowledge base, such as DBpedia or Freebase.

The core idea of the research presented in this paper is to use the RDF graph to compute the relatedness between nodes. Actually, the goal is the relatedness between terms, but concept nodes of this graph typically have a label – a string representation or stringification – that can be seen as a term.

At first sight relatedness may seem to be the inverse of the distance between nodes. Two nodes far apart are unrelated and every node is totally (infinitely) related to itself. Interpreting relatedness as a function of distance has an obvious advantage: computing distances between nodes in a graph is a well studied problem with several known algorithms. After assigning a weight to each arc one can compute the distance as the minimum length of all the paths connecting the two nodes.

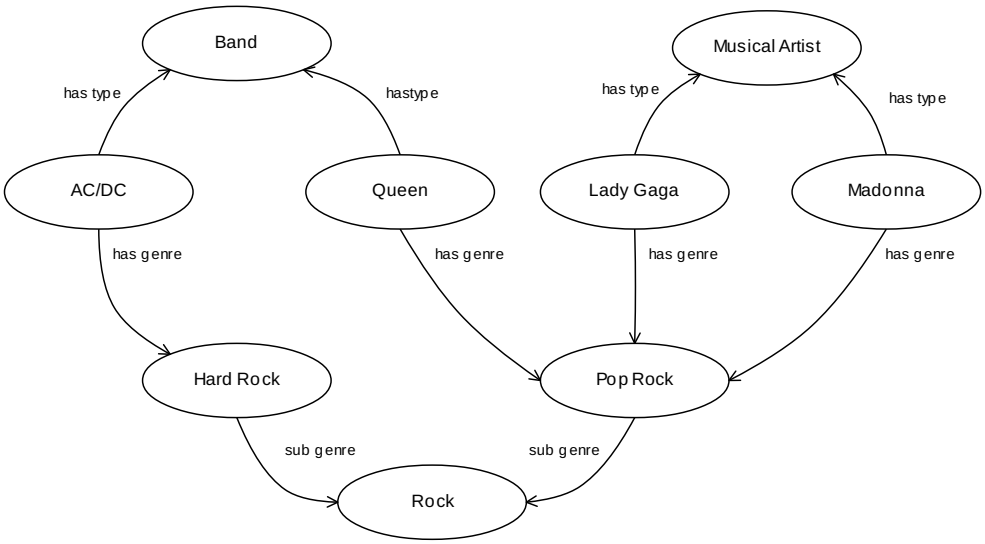


Figure 1 RDF graph for concepts in music domain.

On a closer inspection this interpretation of relatedness as the inverse of distance reveals some problems. Consider the graph in Figure 1. Depending on the weight assigned to the arcs formed by the properties `has type` and `has genre`, the distances between Lady Gaga, Madonna and Queen are the same. If the `has genre` has less weight than `has type`, this would mean that the band Queen is as related to Lady Gaga as Madonna, which obviously should not be the case. On the other hand, if `has type` has less weight than `has genre` then Queen is more related to AC/DC than to Lady Gaga or Madonna simply because they are both bands, which also should not be the case.

In the semantic relatedness methodology proposed, we consider *proximity* rather than distance as a measure of relatedness among nodes. By definition[1], proximity is closeness; the state of being near as in space, time, or relationship. Rather than focusing solely on minimum path length, proximity balances also the number of existing paths between nodes. As an example consider the proximity between two persons. More than resulting from a single common interest, however strong, it results from a collection of common interests.

With this notion of proximity, Lady Gaga and Madonna are more related to each other than with Queen since they have two different paths connecting each other, one through `Musical Artist` and another `Pop Rock`. By the same token the band Queen is more related to them than to the band AC/DC.

An algorithm to compute proximity must take into account the several paths connecting two nodes and their weights. However, paths are made of several edges, and the weight of an edge should contribute less to proximity as it is further away in the path. In fact, there must be a limit in number of edges in a path, as RDF graphs are usually connected graphs.

The main issue with this definition of proximity[2] is how to determine the weights of

[1] `https://en.wiktionary.org/wiki/proximity`
[2] See [10] for a detailed description of the algorithm.

transitions. The first attempt was to define these weights using domain knowledge. For instance, when comparing musical performers one may consider that being associated with a band or with another artist is more important than their musical genre, and that genre is more important than their stylistics influences and even more important than instruments they play.

This naïve approach to weight setting has several problems. Firstly, this kind of "informed opinion" frequently has no evidence to support it, and sometimes is plainly wrong. How sure can one be that stylistics influences should weight more than the genre in musical proximity? Even if it is true sometimes, how can one be sure it is true in most cases? Secondly, this approach is difficult to apply to a large ontology encompassing a broad range of domains. Is a specialist required for every domain? How should an ontology be structured in domains? What domain should be considered for concepts that fall in multiple domains? To be of practical use, the weights of a proximity based semantic relatedness algorithm must be automatically tuned.

3.2 Genetic Algorithm Layer

Genetic algorithms are a family of computational models that mimic the process of natural selection in the evolution of the species. These algorithms use the concepts of *variation*, *differential reproduction* and *heredity* to guide the co-evolution of a set of problem solutions. This type of algorithm is frequently used to improve solutions of optimization problems [14].

There are two necessary conditions for using a genetic algorithm. Firstly, the different candidate solutions must be representable as individuals (*variation*). This encoding of an individual solution is sometimes called a *chromosome* which are a collection of *genes* that characterize the solution. Secondly, it must be possible to compare a set of individuals, decide which are the fittest and allow them to pass their genetic information to the next generation (*differential reproduction*). Also, the representation of solutions as individuals must allow their recombination with other solutions (*heredity*) so that favorable traits are preferred over unfavorable ones as the population of solutions evolves.

A simple approach in the case of weight tuning is to consider as *individual* a vector of weight values. This representation contrasts with the binary representations typically used in genetic algorithms [4]. However it is closer to the domain and it can be processed more efficiently with large number of weights.

Genetic algorithms introduce variance also by *mutation*. The are a number of mutation operators, such as swap, scramble, insertion, that can be used on binary representations [4]. However, the approach taken to represent individuals in this methodology makes these kind of mutations less interesting. Since weights are independent from each other, swapping values among them is as likely to improve the solution as selecting a new random values. Hence, the genetic algorithm created for tuning weights has a single kind of mutation: randomly selecting a new value for a given "gene".

The fitness function plays a decisive role in selecting the new generation of individuals, created by crossover and mutation of their parents. The usual method for estimating the quality of a semantic relatedness function is to compare it with a benchmark data set. The benchmark data set contains pairs of words and their relatedness.

The Spearman's rank order coefficient is commonly used to compare the relatedness values in the benchmark data set with those produced by a semantic relatedness algorithm. Rather than the simple correlation between the two data series, the Spearman's rank order sorts those data series and correlates their rank.

The genetic algorithm of this weight tuning methodology uses as fitness function the

Spearman's rank order coefficient on benchmark data, using as input a vector of weight values assigned to each arc type.

3.3 Bootstrap Layer

The genetic algorithm itself has a number of parameters that must be tuned. Generic parameters of a genetic algorithm include the number of generations and the mutation rate. In this particular case the range of values that may be assigned to weights must also be considered.

Several approaches to tuning parameters of genetic algorithms have been proposed and compared [12]. Although with different approaches, these methods highlight the advantage of using automated parameter tuning over tuning based on expert "informed opinions". In many cases the best solution contradicts the expert best intuitions.

The proposed methodology relies on a single benchmark data set to compare alternative weight attributions. To repeat a large number of experiments using the genetic algorithm to co-evolve a set solutions one needs a larger test sample. Bootstrapping [3] is a statistical method for assigning measures of accuracy to data samples, using simple techniques known as *resampling*.

Resampling is applied to the original data set to build a collection of sample data sets. Each sample data set has the same size as the original data set and is build from the same elements. If the original data set has size n then n elements from that set are randomly chosen to create the sample set. When an element is selected it is not removed from the original data set. Hence, a particular element may occur repeatedly on the sample data set while other may not occur at all.

The bootstrapping method is used for comparing different approaches. Each approach is repeated a large number of times, typically 200, each time with a different sample set. Each approach is summarized by a statistics, such as the mean or the third quartile. In the end, these statistics are compared to select the most effective approach. Since the objective is to select the approach that may lead to the highest Spearman's coefficient, the third quartile is specially relevant since it is a lower bound of the largest solutions.

In this tuning methodology, each approach corresponds to a particular setting of the genetic algorithm. Candidate settings include values for parameters such as the number of generations or the mutation rate. Another important parameter that is specific to this methodology is the range of values that are used as possible values for weights. As these values have to be enumerated, this methodology considers only integer values bellow a certain threshold.

As the result of the bootstrapping method a particular setting of the genetic algorithm's parameters is selected. The final stage is to run the genetic algorithm with these settings, using the full benchmark data set in the fitness function. The selected genetic algorithm is repeated an even larger number of times, typically 1000, and the best result is selected as weights for the relatedness algorithm.

4 Implementation

The methodology presented for parameter tuning has a high computational complexity. At its core it has to find all paths connecting two concepts to compute a single proximity. To test the quality of a vector of weights, the proximity has to be computed for each pair of concepts in a benchmark data set. Bootstrapping repeats 200 times the genetic algorithm for each setting, and this process is repeated for a large number of settings.

The strategy used for improving the efficiency of the methodology has three main components: graph pre-processing (described in the Subsection 4.1), factorization of the proximity algorithm and concurrent evaluation of the bootstrapping method (described in the Subsection 4.2). The remainder of this section details each of these components.

4.1 Graph Pre-processing

The computation of the semantic proximity between two concepts depends on a data graph search that finds all the paths that connect both concepts. The data graph search is implemented in two different ways, supporting queries of remote and local data. The main differences are the methods that retrieve the nodes with a specific label and the methods used to retrieve the transitions from a node used by the semantic relatedness algorithm.

Remote data is usually retrieved from SPARQL endpoints. A SPARQL endpoint is addressed by a URI to which SPARQL queries can be sent and which returns RDF as a response. Paths are built from data collected from two SPARQL queries. The following query retrieves the list of all nodes that have a given string as label. This query is executed twice, one for each label. For each node retrieved with the previous SPARQL query another SPARQL query is executed, as shown bellow, until the set of paths connecting both concepts are finished.

The SPARQL approach raises a number of issues. Firstly, the endpoint or network may be under maintenance or with performance problems. Secondly, some endpoints have configuration problems and do not support queries with some operators, such as `UNION`. And thirdly, the SPARQL queries can have performance issues, mainly when using operators such as `DISTINCT`, and having a large amount of queries per proximity search can cause a huge impact at the execution time.

In order to avoid those issues, this methodology also implements searches in local data. Knowledge bases often provide dumps of their data. Local data are preprocessed RDF graphs that are stored in the local file system, retrieved from those dumps. Graph pre-processing begins with parsing the RDF data. RDF data can be retrieved in several formats, such as Turtle, RDF/XML or N-Triples. To simplify this process, all RDF data is converted to N-Triples, since this is the simplest RDF serialization.

This process takes some time to execute but it is only necessary to execute it once. Also, the most used data is cached in memory which has a significant impact on performance.

The proximity algorithm is based on a previous definition [10]. This algorithm takes two strings as labels and builds a set with all the paths that connect both concepts. In this current implementation, there is a stemming process with labels aiming to increase the meaning scope of each word.

The computation of the proximity of a single pair of concepts using the WordNet 2.0 SPARQL endpoint[3] takes about 20 minutes. With the pre-processed graph[4] that is executed once and takes 30 minutes, the same computation takes about 6 seconds.

4.2 Other Optimizations

Traversing the graph searching for paths connecting two labels is the most frequently executed part of this semantic relatedness methodology. Nevertheless, this procedure is almost the same for each pair of concepts, varying only on the weights that are used for each arc type.

[3] `http://wordnet.rkbexplorer.com/sparql/`
[4] The tests were executed in a 8 core machine at 3.5 GHz and 16Gb of RAM

This computation is repeated many times since the exact same pair of concepts is used each time that the genetic algorithm is run.

The solution found was to alter the proximity algorithm to compute the set of coefficients that are multiplied to each weight. These coefficients are organized in a vector, using the same order of the weight vector used in the genetic algorithm. Thus, computing the proximity of a pair of concepts given a different weight vector is just the inner product of the weight vector and the coefficient vector.

With this modification a single run of the genetic algorithm with 200 generations takes less than a 1 minute and computing the coefficients for all the pairs takes about 30 minutes.

The final optimization was concurrent evaluation of the bootstrapping method. Each of the settings can be processed independently, hence they could be assigned to a different processor of a multi-core machine. Each run of the bootstrapping method takes about 200 minutes. It run 120 configurations that sequentially would take more than 16.5 days in about 2 days.

5 Validation

The validation of the proposed tuning approach consisted of tuning the weights of the relatedness algorithm for WordNet 2.0. The tuning was performed in two rounds. In the first round a large number of settings was explored to determine which were the most relevant. A second round was then performed to explore new settings on those parameters that have more impact on performance.

The tuning process uses as benchmark the WordSimilarity-353 data set [6]. It has 353 pairs of concepts with the mean of the relatedness values given by humans. Since WordNet 2.0 does not have all the words listed in this data set, the pairs with missing elements were removed, creating a new data set with the non-missing pairs. In total 7 pairs were removed.

The bootstrapping process tests three parameters: weight values, mutation rate, and number of generations. The weight values were divided in positive and mixed (positive and negative) values; the positive values ranged in $[0, n]$ with $n \in \mathbb{N}^+$ and $n \leq 10$. The mixed values ranged in $[-n, n]$ for the same values of n. The mutation rate took values in the set $\{0.3, 0.4, 0.5\}$ values and the number of generations in $\{100, 200\}$. Permutating these values, 120 different sets of parameters were tested. Each set of parameters was executed 200 times in the bootstrapping process. The results of those tests can be seen in the following graphs. These graphs show the statistics of the correlation as function of a single variable: number of weight values, mutation rate and number of generations.

Figure 2 shows how the correlation evolves with different amounts and ranges of distinct weights. The correlation obtained when there are only positive values in the weight set is much lower than when positive and negative weights are used. The positive values also appear to reach a maximum value. However, the sets with positive and negative values do not show that stabilization, becoming relevant more tests with a larger range of values.

The graph on the left of Figure 3 shows the impact of changing the mutation rate. Despite the large overall variation, the mean and third quartile values are similar, showing that variations in this parameter have a small impact on the tuning process. Still, the variation of the maximums indicate the relevance of also testing lower mutation rates in the future.

The graph on the right of Figure 3 presents the variation of the number of generations. As it occurs with the mutation rate, changes in the number of generations have no significant impact in the correlation values.

After the first round, changes in range of weight values appear to have a higher impact

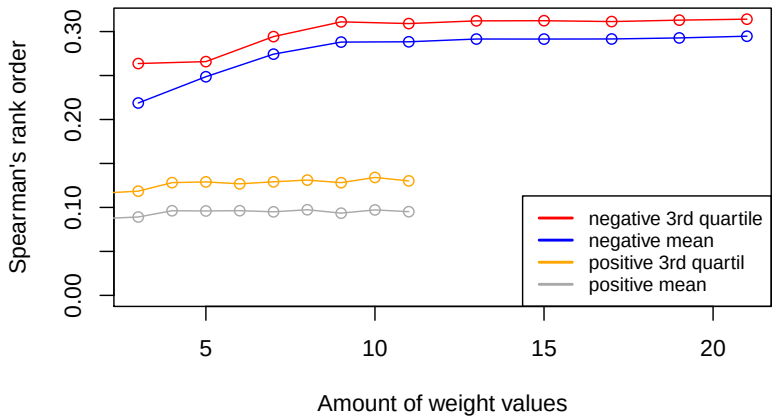


Figure 2 Graph of weights distribution.

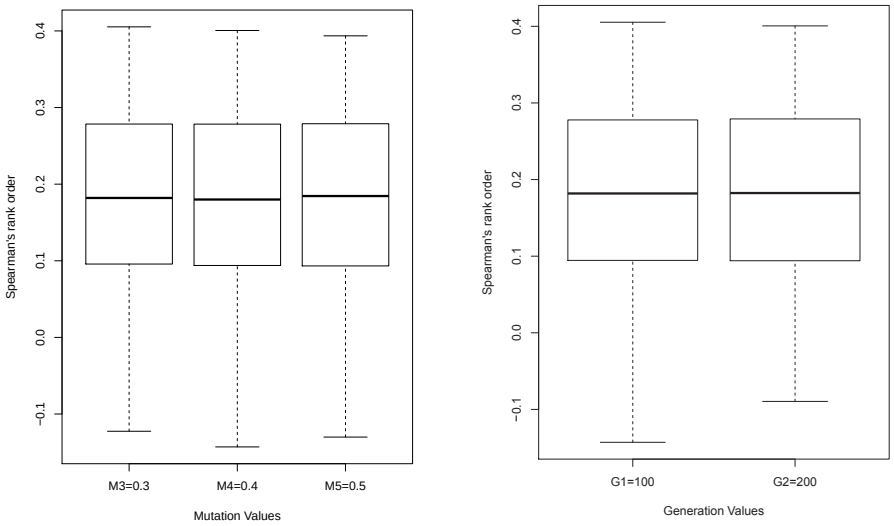


Figure 3 Distribution of different mutation rates (left) and number of generations (right).

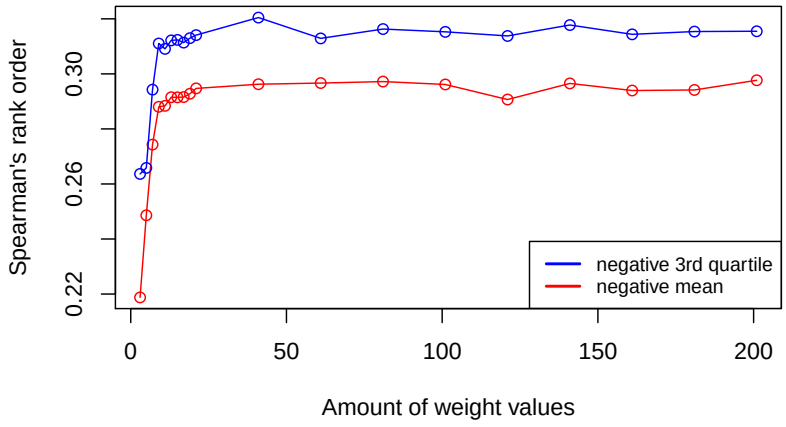


Figure 4 Graph of weights distribution.

Table 1 Weight values obtained after tuning process.

Edge type	Weight	Edge type	Weight
null	1	wn:classifiedByUsage	18
wn:memberMeronymOf	−2	wn:tagCount	1
wn:participleOf	−6	wn:sameVerbGroupAs	−8
wn:antonymOf	−5	wn:derivationallyRelated	0
wn:classifiedByTopic	19	wn:attribute	−15
wn:partMeronymOf	19	wn:synsetId	18
wn:word	12	wn:seeAlso	6
wn:gloss	19	rdfs:type	−2
wn:similarTo	−19	entails	−1
wn:containsWordSense	4	wn:classifiedByRegion	−9
wn:causes	−17	wn:adverbPertainsTo	12
wn:frame	7	wn:hyponymOf	9
wn:adjectivePertainsTo	3	wn:substanceMeronymOf	18

Table 2 Previous work with WordNet and WordSim-353.

Method	Spearman's rank order
Jarmasz (2003)	0.33 - 0.35
Strube and Ponzetto (2006)	0.36
Proposed method	**0.41**

in the correlation values, specially if they allow negative values, increasing the correlation as the range size increases. New tests were needed to investigate for how long the correlation continues to increase, if it converges to an asymptote, or if the correlation degrades after a certain threshold.

A new round of tests was made to investigate these hypothesis. This time only positive and negative values were used, with fixed values of mutation rate and number of generations. These new configurations uses ranges from $[-10 \times n, 10 \times n]$ with $n \in \mathbb{N}^+$ and $n \leq 10$. The mutation rate value was fixed at 0.4 and the number of generations was fixed at 200. The results are displayed in Figure 4. The values obtained by increasing the range of values show a maximum value at the range [-20,20]. Ranges with higher values seem to never exceed the Spearman's rank order obtained at that point, indicating that performance degrades after this threshold.

Using the best configuration obtained by the bootstrap process the genetic algorithm was executed 1000 times aiming to obtain the best correlation value and the related configuration.

The best Spearman's rank order value obtained was 0.409 and the corresponding weight set is listed in the Table 1. The edges with the prefix `wn` correspond to the WordNet 2.0 URI[5] and the prefix `rdfs` to the RDF Schema URI[6]. The edge type `null` is the custom edge created in the stemming process.

Table 2 compares the results obtained by tuning the edge weights without domain knowledge with other methodologies.

[5] `http://www.w3.org/2006/03/wn/wn20/schema/`
[6] `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`

6 Conclusion

The major contribution of the research presented in this paper is a method for tuning a relatedness algorithm to a particular ontological graph, without requiring any domain knowledge on the graph itself. The results obtained with this approach for WordNet 2.0 are better than the state of the art for the same graph. A number of solutions to speedup graph processing and the evaluation of fitness functions are also relevant contributions.

The proposed tuning approach performs a multiscale parameter tuning of an ontology based semantic relatedness algorithm. The main feature of the base algorithm is the fact that it considers all paths in an ontological graph that connect two labels and computes the contribution of each path as a function of its length and of the type of its arcs (properties). The main issue of this algorithm is the selection of parameters (weight values for each type of arc) that maximize the quality of the relatedness algorithm.

The quality of semantic relatedness algorithms is usually measured against a benchmark data set. This data set consists of the relatedness of a set of words, defined as the mean of the relatedness attributed by a group a persons. The quality of the algorithm is computed as the Spearman's rank correlation coefficient between the relatedness produced by the algorithm and the relatedness given by the data set. By defining this correlation as a function of weight assignments it is possible to frame the problem of maximizing the quality of the relatedness algorithm as finding the maximum of a function.

Evolutionary algorithms in general, and genetic algorithm in particular, are popular choices for improving the quality of solutions. Using a genetic algorithm it is possible to use variation and selection to improve the Spearman's coefficient. The proposed genetic algorithm uses as chromosome a set of weights attributions. The range of values used in attributions, as well as the number of generations and the mutation rate are in turn parameters that must also be tuned.

The statistical method of bootstrapping was used to measure the accuracy of different parameter settings. This method generates diversity by producing many sample data sets from the original data set. Bootstrapping is used to compare the results of the genetic algorithm with different settings. After selecting the best candidate parameters for the genetic algorithm, this is rerun with the complete benchmark data set.

The proposed approach for tuning the parameters of the semantic relatedness algorithm was validated with Wordnet 2.0. The tuning procedure was actually executed twice. In the first run several parameters of the genetic algorithm were tested to conclude that the range of weight values is the decisive parameter, in particular if it is allowed to contain negative values. The variation of some of the parameters, such as mutation rate and number of generations, had no impact on the quality. Based on these findings a second run of the tuning procedure explored a wider range of values. It showed that quality improves with the width of range values but also that a small degradation occurs after a certain threshold. The genetic algorithm was finally repeated a large number of times with the settings selected by this approach and the maximum Spearman's correlation obtained is significantly higher than the best result reported on the literature for the same graph.

The Wordnet 2.0 graph used for the evaluation is comparatively small. It has just 26 different types of properties and 464.795 nodes. The next step is to investigate how this approach works with Wordnet 3.0 and with even larger graphs, such as the DBPedia or Freebase. Apart from the challenges of dealing with such large graphs, it will be interesting to compare the semantic relatedness potential of different graphs and try to combine them to improve the accuracy of the semantic relatedness algorithm.

Acknowledgements. This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-037281. Project "NORTE-07-0124-FEDER-000059" is financed by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by national funds, through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT). The authors would also like to thank Luis Torgo for his help in the use of statistical methods.

References

1 Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Paşca, and Aitor Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 19–27. Association for Computational Linguistics, 2009.

2 Danushka Bollegala, Yutaka Matsuo, and Mitsuru Ishizuka. Measuring semantic similarity between words using Web search engines. *Proceedings of the 16th international conference on World Wide Web*, 7:757–766, 2007.

3 Bradley Efron and Robert J. Tibshirani. *An introduction to the bootstrap*, volume 57. CRC press, 1994.

4 Agoston E. Eiben and James E. Smith. *Introduction to evolutionary computing*. Springer, 2003.

5 Christiane Fellbaum. *WordNet*. Wiley Online Library, 1999.

6 Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing search in context: The concept revisited. In *Proceedings of the 10th international conference on World Wide Web*, pages 406–414. ACM, 2001.

7 Yuriy Gorodnichenko and Gerard Roland. Understanding the individualism-collectivism cleavage and its effects: Lessons from cultural psychology. *Institutions and Comparative Economic Development*, 150:213, 2012.

8 Graeme Hirst and David St-Onge. Lexical chains as representations of context for the detection and correction of malapropisms. *WordNet: An electronic lexical database*, 305:305–332, 1998.

9 Mario Jarmasz. Roget's thesaurus as a lexical resource for natural language processing. *CoRR*, abs/1204.0140, 2012.

10 José Paulo Leal. Using proximity to compute semantic relatedness in rdf graphs. *Comput. Sci. Inf. Syst.*, 10(4), 2013.

11 Laurent Mazuel and Nicolas Sabouret. Semantic relatedness measure using object properties in an ontology. In *The Semantic Web-ISWC 2008*, pages 681–694. Springer, 2008.

12 Selmar K. Smit and Agoston E. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 399–406. IEEE, 2009.

13 Michael Strube and Simone Paolo Ponzetto. Wikirelate! computing semantic relatedness using wikipedia. In *AAAI*, volume 6, pages 1419–1424, 2006.

14 Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

Rocchio's Model Based on Vector Space Basis Change for Pseudo Relevance Feedback

Rabeb Mbarek[1], Mohamed Tmar[2], and Hawete Hattab[3]

1 **Sfax University**
Multimedia Information systems and Advanced Computing Laboratory
Sfax, Tunisia
rabeb.hattab@gmail.com

2 **Sfax University**
Multimedia Information systems and Advanced Computing Laboratory
Sfax, Tunisia
mohamedtmar@yahoo.fr

3 **Umm Al-qura University, Department of Mathematics**
Makkah, KSA
hshattab@uqu.edu.sa

Abstract

Rocchio's relevance feedback model is a classic query expansion method and it has been shown to be effective in boosting information retrieval performance. The main problem with this method is that the relevant and the irrelevant documents overlap in the vector space because they often share same terms (at least the terms of the query). With respect to the initial vector space basis (index terms), it is difficult to select terms that separate relevant and irrelevant documents. The Vector Space Basis Change is used to separate relevant and irrelevant documents without any modification on the query term weights. In this paper, first, we study how to incorporate Vector Space Basis Change into the Rocchio's model. Second, we propose Rocchio's models based on Vector Space Basis Change, called VSBCRoc models. Experimental results on a TREC collection show that our proposed models are effective.

1998 ACM Subject Classification H.3.3 Information Search and Retrieval

Keywords and phrases Rocchio model, vector space basis change, pseudo relevance feedback

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.215

1 Introduction

In the Vector Space Model (VSM), each component of the vector represents a term in the document [18] i.e. each component in the vector represents the weight of the term in the document. The set of all index terms is called the original vector space basis. For the most vector space based Information Retrieval (IR) and feedback models, the original vector space basis generates documents and queries. Although several term weighting and feedback methods have been proposed, only a few approaches [4, 11, 8, 9] consider that changing the vector space basis from the original vector space basis into another basis is an issue of investigation. The Vector Space Basis Change (VSBC) consists of using a transition matrix[1]. By changing the vector space basis, each vector coordinate changes depending on this matrix. If we change the basis, then the inner product changes and so the Cosine

[1] The algebraic operator responsible for change of basis.


© Rabeb Mbarek, Mohamed Tmar, and Hawete Hattab;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 215–224

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

function behavior changes [10]. By the same Dice, Jaccard and Overlap functions behavior changes.

Pseudo Relevance Feedback (PRF) is known as a useful method for enhancing retrieval performance. It assumes that the top-ranked n documents (pseudo-documents) of the initial retrieval are relevant and extracts expansion terms from them. PRF has been shown to be effective in improving IR performance [2, 3, 6, 7, 13, 14, 16, 17, 19, 20, 21]. PRF can also fail in some cases. For example, when some pseudo-documents contain terms of the irrelevant contents, then these terms misguide the feedback models by importing noisy terms into the queries. This could influence the retrieval performance in a negative way.

With respect to the original vector space basis, relevant and irrelevant documents share some terms (at least the terms of the query which selected these documents). To avoid this problem, it suffice to separate relevant and irrelevant documents. VSBC is an effective method for the separation of relevant and irrelevant documents. This method has been studied in the past few years [4, 11, 8, 9]. In [8, 9], the authors have been found a basis which gathers the relevant documents and the irrelevant ones are kept away from the relevant ones. These approaches have been evaluated on a Relevance Feedback (RF) framework.

Rocchio's model [16] is a classic framework for implementing (pseudo) RF via improving the query representation. It models a way of incorporating (pseudo) RF information into the VSM in IR. In this paper, first, we study how to incorporate VSBC into the Rocchio's model. Second, we propose Rocchio's models based on the VSBC, called VSBCRoc models.

This paper is organized as follows. section 2 presents the related work. Sections 3 describes our approach based on the VSBC. In Section 4, the experimental results are presented and discussed. A direct comparison is made to compare VSBCRoc models with the classical Rocchio's model. Finally, we conclude our work with a brief conclusion and future research directions in Section 5.

2 Related Work

The VSM [18] is adopted to rank the documents. This model showed good feedback performance on most collections whereas the probabilistic model had problems with some collections [5].

2.1 Vector Space Basis Change

The Latent Semantic Indexing (LSI) [4] exploits the hypothesis that the term-document frequency matrix encloses information about the semantic relations between terms and between documents. This technique is based on Singular Value Decomposition (SVD) aiming at decomposing the matrix and disclosing the principal components used to represent fewer independent concepts than many inter-dependent index terms. This method results on a new vector space basis, with a lower dimension than the original one (all index terms), and in which each component is a linear combination of the indexing terms.

When using a term to express a query or a document, the user gave to the term a semantics which is different from the semantics of the same term used by another user or by the same user in another place, time, need. In other words, the use of a term depends on context. Therefore, context influences the selection of the terms, their semantics and inter-relationships. A vector space basis models a document or query terms. The semantics of a document or query term depends on context. A vector space basis can be derived from a context. Therefore, a vector space basis of a vector space is the construct to model context.

Also, change of context can be modeled by linear transformations from one base to another which is a VSBC [10, 11].

Recently, Mbarek et al. [8, 9] developed a RF algorithms based on a vector space basis change. These RF algorithms improve the results of known models (BM25 model, Rocchio model). They built a basis which gives a better representation of documents. This basis should minimize the sum (S_1) of squared distances between each relevant document and g_R (g_R is the centroïd of relevant documents) and should maximize the sum (S_2) of squared distances between each irrelevant document and g_R. And so this basis should minimize the quotient $\frac{S_1}{S_2}$ [8] and maximize the difference $S_2 - S_1$ [9].

2.2 Pseudo-Relevance Feedback

In IR, PRF via query expansion is referred to as the techniques that reformulate the original query by adding new terms into the query, in order to achieve a better retrieval performance. There are a large number of studies on the topic of PRF. Here we mainly review the work about PRF which is the most related to our research. A classical RF technique was proposed by Rocchio in 1971 for the Smart retrieval system [16]. It is a framework for implementing (pseudo) RF via improving the query representation, in which a set of documents are utilized as the feedback information. Unique terms in this set are ranked in descending order of their $tf * idf$ weights. In the following decades, many other RF techniques and algorithms were developed, mostly derived under Rocchio's framework. A popular and successful automatic PRF algorithm was proposed by [14] in the Okapi system; Amati et al. [1] proposed a query expansion algorithm in his divergence from randomness retrieval framework; Carpineto et al. [2] proposed to compute the weight of candidate expansion terms based on the divergence between the probability distributions of terms in the top ranked documents and the whole collection; Miao et al. [12] studied how to incorporate proximity information into the Rocchio's model, and proposed three proximity based Rocchio's models.

In this paper, first, we will incorporate VSBC into the Rocchio's model. Second, we propose Rocchio's models based on VSBC, called VSBCRoc models.

3 Rocchio's Models based on Vector Space Basis Change

3.1 Rocchio's Formula

Rocchio's model [16] is a classic framework for implementing (pseudo) RF via improving the query representation. It models a way of incorporating (pseudo) relevance feedback information into the VSM in IR. In case of PRF, the Rocchio's model (without considering negative feedback documents) has the following steps:

- All documents are ranked for the given query using a particular retrieval model. This step is called initial retrieval, from which the $|R|$ highest ranked documents are used as the feedback set.
- The representation of the query is finally refined by taking a linear combination of the initial query term vector with the feedback document vector, this initial formula is denoted by VSBCRoc1:

$$VSBCRoc1 : Q_1 = \alpha * Q_0 + \beta * \sum_{d \in R} \frac{d}{|R|} \quad (1)$$

where Q_0 represents the original query vector, Q_1 represents the first iteration query vector, d is the document weight vector, and α and β are tuning constants controlling how

much we rely on the original query and the feedback information. In practice, we can always fix α at 1, and only study β in order to get better performance.

3.2 Vector Space Basis Change

In [8, 9], the authors built a new vector space basis which separates relevant and irrelevant documents without any modification on the query term weights. That is, this basis gathers the relevant documents and the irrelevant ones are kept away from the relevant ones. It can be viewed as a representation that keeps the relevant documents gathered to their centroïd and the irrelevant ones far from it. Each document d_i is represented in a vector space by $d_i = (w_{i1}, w_{i2}, ...w_{iN})^T$ where w_{ij} is the weight of term t_j in document d_i and N is the number of index terms[2]. As for us our approach is independent of the term weighting method.

The Euclidian distance between documents d_i and d_j is given by:

$$\begin{aligned} dist(d_i, d_j) &= \sqrt{\sum_{k=1}^{N} (w_{ik} - w_{jk})^2} \\ &= \sqrt{(w_{i1} - w_{j1}...w_{iN} - w_{jN}) \cdot (w_{i1} - w_{j1}...w_{iN} - w_{jN})^T} \\ &= \sqrt{(d_i - d_j)^T \cdot (d_i - d_j)}\,. \end{aligned}$$

By changing the basis using a transition matrix M, the distance between 2 vectors d_i^* and d_j^* which are respectively d_i and d_j rewritten in the new basis is given by:

$$\begin{aligned} dist(d_i^*, d_j^*) &= dist(M.d_i, M.d_j) \\ &= \sqrt{(M.d_i - M.d_j)^T \cdot (M.d_i - M.d_j)} \\ &= \sqrt{(d_i - d_j)^T \cdot M^T M \cdot (d_i - d_j)}\,. \end{aligned}$$

The vector space basis which optimally separates relevant and irrelevant documents is represented by a matrix M^* called the optimal transition matrix. M^* puts the relevant documents gathered to their centroïd g_R and the irrelevant documents far from it.

g_R is done by:

$$g_R = \frac{1}{|R|} \sum_{d \in R} d$$

where R is the set of relevant documents.

By the same, using a transition matrix M, we obtain:

$$M.g_R = M \cdot (\frac{1}{|R|} \sum_{d \in R} d) = \frac{1}{|R|} \sum_{d \in R} M \cdot d\,.$$

[2] x^T is the transpose of x

The optimal matrix M^* should minimize the sum of squared distances between each relevant document and g_R, i.e.:

$$\begin{aligned} M^* &= \arg\min_{M \in M_n(\mathbb{R})} \sum_{d \in R} dist^2(M \cdot d, M \cdot g_R) \\ &= \arg\min_{M \in M_n(\mathbb{R})} \sum_{d \in R} (Md - Mg_R)^T \cdot (Md - Mg_R) \\ &= \arg\min_{M \in M_n(\mathbb{R})} \sum_{d \in R} (d - g_R)^T \cdot M^T M \cdot (d - g_R)\,. \end{aligned} \tag{2}$$

By the same, the optimal matrix M^* should maximize the sum of squared distances of each irrelevant document and g_R, which leads on the following:

$$\begin{aligned} M^* &= \arg\max_{M \in M_n(\mathbb{R})} \sum_{d \in S} dist^2(M \cdot d, M \cdot g_R) \\ &= \arg\max_{M \in M_n(\mathbb{R})} \sum_{d \in S} (Md - Mg_R)^T \cdot (Md - Mg_R) \\ &= \arg\max_{M \in M_n(\mathbb{R})} \sum_{d \in S} (d - g_R)^T \cdot M^T M.(d - g_R) \end{aligned} \tag{3}$$

where S is the set of irrelevant documents.

In [8], the authors have been showed that the Equations 2 and 3 result on the following single equation:

$$M^* = \arg\min_{M \in M_n(\mathbb{R})} \frac{\sum_{d \in R} (d - g_R)^T \cdot M^T M \cdot (d - g_R) + \alpha}{\sum_{d \in S} (d - g_R)^T \cdot M^T M \cdot (d - g_R) + \alpha} \tag{4}$$

where α is real parameter close to 0.

And in [9], the authors have been showed that the Equations 2 and 3 result on the following single equation:

$$M^* = \arg\max_{M \in M_n(\mathbb{R})} [\sum_{d \in S} (d - g_R)^T \cdot M^T M \cdot (d - g_R) - \sum_{d \in R} (d - g_R)^T \cdot M^T M \cdot (d - g_R)]\,. \tag{5}$$

Let M_1^* be a solution of Equation 4 and M_2^* be a solution of Equation 5. These matrices separate relevant and irrelevant documents. The proposed Rocchio's models based on VSBC are:

$$VSBCRoc2 : Q_2 = Q_0 + \beta * \sum_{d \in R} \frac{M_1^* d}{|R|} \tag{6}$$

$$VSBCRoc3 : Q_3 = Q_0 + \beta * \sum_{d \in R} \frac{M_2^* d}{|R|} \tag{7}$$

We remark that the initial Rocchio's formula, VSBCRoc1 (Equation 1), corresponds to incorporating the identity matrix[3] (there is no basis change).

[3] A square matrix with ones on the main diagonal and zeros elsewhere.

4 Experiments

In this section we give the different experiments and results obtained to evaluate our approach. We describe the environnement of evaluation and the experimental conditions.

4.1 Environment

The test collection TREC-7 was used for the experiments in this article. Data was preprocessed through stop-word removal and Porter's stemming, and one–word terms were stored; the initial rankings of documents (Baseline Model) were weighted by the $BM25$ formula proposed in [15]. BM25 parameters are $b = 0.5$, $k_1 = 1.2$, $k_2 = 0$ and $k_3 = 8$.

- The initial query Q_0 is made from the short topic description, and using it the top 1000 documents are retrieved from the collections (weighted $\alpha = 1$).
- R is the set of top ranking n documents, assumed to be relevant.
- S is the set of retrieved documents $501 - 1000$, assumed to be irrelevant.

For the three approaches, the retrieved documents are ranked by the inner product done by:

$$RSV(Q_i, d) = Q_i^T \cdot d \qquad 1 \leq i \leq 3 \tag{8}$$

4.2 Results

To evaluate the performance we execute several runs using the topics provided by TREC. In detail, the TREC-7 collection has 50 topics. Topics are structured in three fields: title, description and narrative. To generate a query, the title of a topic was used, thus falling into line with the common practice of TREC experiments; description and narrative were not used.

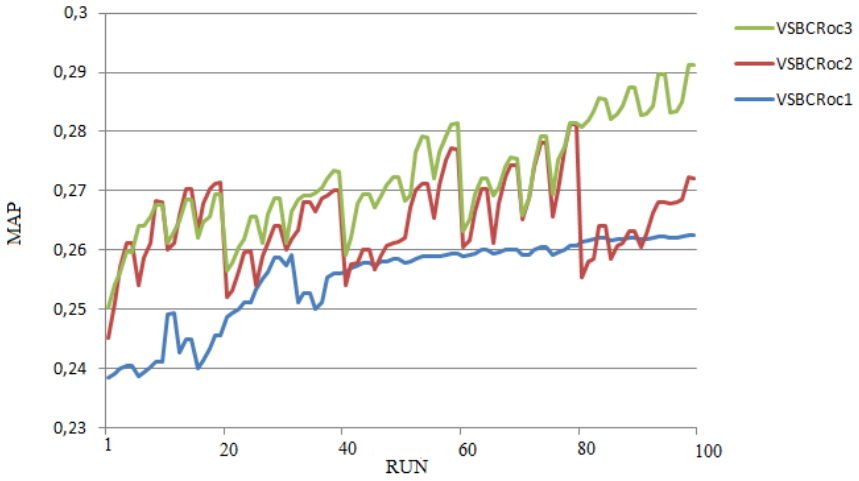
We perform 100 runs by considering all possible combinations of the three parameters involved in the three models. In particular, we take into account: n (the cardinality of R), m (the number of expansion terms) and the parameter β (used for the linear combination): see Equations 1, 6 and 7. We select different ranges for each parameter: n ranges in $(1, 2, 3, 4, 5)$, m ranges in $(10, 20, 30, 50)$ and β ranges in $(0.1, 0.2, 0.5, 1, 2)$.

We evaluate each run in terms of Mean Average Precision (MAP). The experiments and the evaluations are articulated around the comparison between VSBCRoc1, VSBCRoc2 and VSBCRoc3.

Figure 1 plots the MAP values for each run and approach: VSBCRoc1 is the original Rocchio model, VSBCRoc2 and VSBCRoc3 are the new Rocchio models obtained by incorporating the VSBC strategy. These graphs highlights as the system performance vary according to parameters changes. It is possible to note that:

- VSBCRoc2 and VSBCRoc3 models have better performance than VSBCRoc1 model.
- The MAP value of VSBCRoc1 is similar for $\beta = 1$ and $\beta = 2$ (the same remark for VSBCRoc2 and VSBCRoc3).
- The MAP values of VSBCRoc1, VSBCRoc2 and VSBCRoc3 increase if the number of expansion terms increase.
- The MAP values of VSBCRoc1 and VSBCRoc3 increase if the number of pseudo-documents increase.

Figure 1 Plot of MAP values on TREC-7.



For the VSBCRoc1, VSBCRoc2 and VSBCRoc3 models, the lowest MAP value is 0.2385, 0.2451 and 0.2503, respectively. This value occurs when only one relevant document and 10 expansion terms are involved. The highest MAP value for VSBCRoc1 is 0.2625, while for VSBCRoc3 is 0.2913. Both values are obtained with 5 relevant documents and 50 expansion terms. The highest MAP value for VSBCRoc2 is 0.2813. This value occurs when 4 relevant documents and 50 expansion terms are involved.

4.3 Significance of Our Results

Statistical significance is the probability that an effect is not due to just chance. These tests are based on a pre-specified low probability threshold called p-values. P-values are always coupled to a significance level, usually at 0.05. Thus, if a p-value was found to be less than 0.05, then the result would be considered statistically significant. To study the statistical significance of our result we use a free software environment, R, for statistical computing and graphics[4]. Before applying the student's t-test we compute a R data frame in which each row has a measurement and a categorical system identifier.

Listing 1 t-test of significance of the difference of results of VSBCRoc1 and VSBCRoc2.

```
>  MAP<-c(0.2385,0.2392,0.2401,...,0.2685,0.2723,0.2722)
> Sys<-c("VSBCRoc1","VSBCRoc1","VSBCRoc1","VSBCRoc1",...,"VSBCRoc2",
"VSBCRoc2","VSBCRoc2")
> X<-data.frame(MAP=MAP,Sys=Sys)
> X
      MAP       Sys
1   0.2385  VSBCRoc1
2   0.2392  VSBCRoc1
3   0.2401  VSBCRoc1
.     .         .
.     .         .
.     .         .
```

[4] http://www.r-project.org/

```
100 0.2625 VSBCRoc1
101 0.2451 VSBCRoc2

.    .         .
.    .         .
.    .         .
198 0.2685 VSBCRoc2
199 0.2723 VSBCRoc2
200 0.2722 VSBCRoc2
> t.test(MAP ~ Sys, paired=T, data=X)

        Paired t-test

data:  MAP by Sys
t = -11.7418, df = 99, p-value < 2.2e-16
```

Listing 2 t-test of significance of the difference of results of VSBCRoc1 and VSBCRoc3.

```
>  MAP<-c(0.2385,0.2392,0.2401,...,0.2851,0.2913,0.2913)
> Sys<-c("VSBCRoc1","VSBCRoc1","VSBCRoc1",...,"VSBCRoc3",
"VSBCRoc3","VSBCRoc3")
> X<-data.frame(MAP=MAP,Sys=Sys)
> X
     MAP       Sys
1   0.2385 VSBCRoc1
2   0.2392 VSBCRoc1
3   0.2401 VSBCRoc1
.    .         .
.    .         .
.    .         .
100 0.2625 VSBCRoc1
101 0.2503 VSBCRoc3
.    .         .
.    .         .
.    .         .
198 0.2851 VSBCRoc3
199 0.2913 VSBCRoc3
200 0.2913 VSBCRoc3
> t.test(MAP ~ Sys, paired=T, data=X)

        Paired t-test
data:  MAP by Sys
t = -26.4026, df = 99, p-value < 2.2e-16
```

Listing 3 t-test of significance of the difference of results of VSBCRoc2 and VSBCRoc3.

```
>  MAP<-c(0.2451,0.2511,0.2572,...,0.2851,0.2913,0.2913)
> Sys<-c("VSBCRoc2","VSBCRoc2","VSBCRoc2",...,"VSBCRoc3",
"VSBCRoc3","VSBCRoc3")
> X<-data.frame(MAP=MAP,Sys=Sys)
> X
     MAP       Sys
1   0.2451 VSBCRoc2
2   0.2511 VSBCRoc2
3   0.2572 VSBCRoc2
```

```
.     .         .
.     .         .
.     .         .
100  0.2722  VSBCRoc2
101  0.2503  VSBCRoc3
.     .         .
.     .         .
.     .         .
198  0.2851  VSBCRoc3
199  0.2913  VSBCRoc3
200  0.2913  VSBCRoc3
> t.test(MAP ~ Sys, paired=T, data=X)

        Paired t-test
data:  MAP by Sys
t = -8.6917, df = 99, p-value = 7.741e-14
```

In listings 1, 2 and 3 we have the p-values < 0.05, then our results are statistical significant.

5 Conclusion

The main problem with Rocchio's approach is that the relevant and the irrelevant documents overlap in the vector space because they often share same terms (at least those of the query). Therefore it is difficult to select terms that separate relevant and irrelevant documents which cause the query drift problem (Croft and Harper). To guide the RF process, the authors of [8, 9] have been computed a vector space basis which gives a better representation of the documents such that the relevant documents are gathered and the irrelevant ones are kept away from the relevant documents. Vector space basis change discriminates irrelevant documents from relevant ones, thus reducing the potential noise in the vector space after produced by query expansion. The combinations of Rocchio's models with vector space basis change improve the results of classic Rocchio's formula.

This paper reports about incorporating transition matrix (i.e. the algebraic operator responsible for change of basis) into the classic Rocchio's model. We intend to incorporate other algebraic operator (like vector product) into the classic Rocchio's model.

References

1 G. Amati. *Probabilistic models for information retrieval based on divergence from randomness*. PhD thesis, Department of Computing Science, University of Glasgow, 2003.

2 Claudio Carpineto, Renato de Mori, Giovanni Romano, and Brigitte Bigi. An information-theoretic approach to automatic query expansion. *ACM Trans. Inf. Syst.*, 19(1):1–27, January 2001.

3 Kevyn Collins-Thompson. Reducing the risk of query expansion via robust constrained optimization. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM'09, pages 837–846, New York, NY, USA, 2009. ACM.

4 Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

5 Donna Harman. Relevance feedback revisited. In *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'92, pages 1–10, New York, NY, USA, 1992. ACM.

6 Xiangji Huang, Yan Rui Huang, Miao Wen, Aijun An, Yang Liu, and J. Poon. Applying data mining to pseudo-relevance feedback for high performance text retrieval. In *Data Mining, 2006. ICDM'06. Sixth International Conference on*, pages 295–306, Dec 2006.

7 Victor Lavrenko and W. Bruce Croft. Relevance based language models. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'01, pages 120–127, New York, NY, USA, 2001. ACM.

8 Rabeb Mbarek and Mohamed Tmar. Relevance feedback method based on vector space basis change. In *Proceedings of the 19th International Conference on String Processing and Information Retrieval*, SPIRE'12, pages 342–347, Berlin, Heidelberg, 2012. Springer-Verlag.

9 Rabeb Mbarek, Mohamed Tmar, and Hawete Hattab. A new relevance feedback algorithm based on vector space basis change. In Alexander Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing*, volume 8404 of *Lecture Notes in Computer Science*, pages 355–366. Springer Berlin Heidelberg, 2014.

10 Massimo Melucci. Context modeling and discovery using vector space bases. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM'05, pages 808–815, New York, NY, USA, 2005. ACM.

11 Massimo Melucci. A basis for information retrieval in context. *ACM Trans. Inf. Syst.*, 26(3):14:1–14:41, June 2008.

12 Jun Miao, Jimmy Xiangji Huang, and Zheng Ye. Proximity-based rocchio's model for pseudo relevance. In *Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'12, pages 535–544, New York, NY, USA, 2012. ACM.

13 Karthik Raman, Raghavendra Udupa, Pushpak Bhattacharya, and Abhijit Bhole. On improving pseudo-relevance feedback using pseudo-irrelevant documents. In *Proceedings of the 32Nd European Conference on Advances in Information Retrieval*, ECIR'2010, pages 573–576, Berlin, Heidelberg, 2010. Springer-Verlag.

14 Stephen E. Robertson, Steve Walker, Micheline Hancock-Beaulieu, Mike Gatford, and A. Payne. Okapi at trec-4. In *TREC*, 1995.

15 Stephen E. Robertson, Steve Walker, Micheline Hancock-Beaulieu, Aarron Gull, and Marianna Lau. Okapi at trec. In *TREC*, pages 21–30, 1992.

16 G. Salton. *The SMART retrieval system: experiments in automatic document processing.* Prentice-Hall series in automatic computation. Prentice-Hall, 1971.

17 Gerard Salton and Chris Buckley. Improving retrieval performance by relevance feedback. In Karen Sparck Jones and Peter Willett, editors, *Readings in Information Retrieval*, pages 355–364. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

18 Cornelis Joost van Rijsbergen. *The geometry of information retrieval.* Cambridge University Press, 2004.

19 Ryen W. White and Gary Marchionini. Examining the effectiveness of real-time query expansion. *Inf. Process. Manage.*, 43(3):685–704, May 2007.

20 Jinxi Xu and W. Bruce Croft. Improving the effectiveness of information retrieval with local context analysis. *ACM Trans. Inf. Syst.*, 18(1):79–112, January 2000.

21 Chengxiang Zhai and John Lafferty. Model-based feedback in the language modeling approach to information retrieval. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM'01, pages 403–410, New York, NY, USA, 2001. ACM.

Automatic Identification of Whole-Part Relations in Portuguese

Ilia Markov[1,3], Nuno Mamede[2,3], and Jorge Baptista[1,3]

1 **Universidade do Algarve/FCHS and CECL**
Campus de Gambelas, 8005-139 Faro, Portugal
{jbaptis,a48654}@ualg.pt
2 **Instituto Superior Técnico, Universidade de Lisboa**
Av. Rovisco Pais, 1049-001 Lisboa, Portugal
Nuno.Mamede@ist.utl.pt
3 **INESC-ID Lisboa/L2F – Spoken Language Lab**
R. Alves Redol, 9, 1000-029 Lisboa, Portugal
{Nuno.Mamede,jbaptis,Ilia.Markov}@l2f.inesc-id.pt

Abstract

In this paper, we improve the extraction of semantic relations between textual elements as it is currently performed by STRING, a hybrid statistical and rule-based Natural Language Processing chain for Portuguese, by targeting *whole-part* relations (*meronymy*), that is, a semantic relation between an entity that is perceived as a constituent part of another entity, or a member of a set. In this case, we focus on the type of meronymy involving human entities and *body-part nouns*.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases whole-part relation, meronymy, body-part noun, disease noun, Portuguese

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.225

1 Introduction

Automatic identification of semantic relations is an important step in extracting meaning out of texts, which may help several other Natural Language Processing (NLP) tasks, such as Question Answering (QA), Text Summarization (TS), Machine Translation (IR), Information Extraction (IE), Information Retrieval (IR) and others [9, 10, 15].

The goal of this work is to improve the extraction of semantic relations between textual elements in STRING, a hybrid statistical and rule-based NLP chain for Portuguese[1] [17]. At this time, only the first steps have been taken in the direction of semantic parsing. This work will target whole-part relations (*meronymy*), that is, a semantic relation between an entity that is perceived as a constituent part of another entity, or a member of a set. In this case, we focus on the type of meronymy involving human entities and *Nbp*. This paper is structured as follows: Section 2 briefly describes related work on whole-part dependencies extraction, while Section 3 explains with some detail how this task was implemented in STRING; Section 4 presents the evaluation procedure; and Section 5 draws the conclusions from this work.

[1] https://string.l2f.inesc-id.pt/ [last access: 04/05/2014].


© Ilia Markov, Nuno Mamede, and Jorge Baptista;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 225–232

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Related Work

Meronymy is a complex relation that "should be treated as a collection of relations, not as a single relation" [14]. In NLP, various information extraction techniques have been developed in order to capture whole-part relations from texts.

Hearst [12] tried to find lexical correlates to the *hyponymic* relations (type-of relations) by searching in unrestricted, domain-independent text for cases where known hyponyms appear in proximity. The author proposed six lexico-syntactic patterns; he then tested the patterns for validity, and used them to extract relations from a corpus. To validate his acquisition method, the author compared the results of the algorithm with information found in WordNet [5]. The author reports that when the set of 152 relations that fit the restrictions of the experiment (both the hyponyms and the hypernyms are unmodified) was looked up in WordNet: "180 out of the 226 unique words involved in the relations actually existed in the hierarchy, and 61 out of the 106 feasible relations (*i.e.*, relations in which both terms were already registered in WordNet) were found." [12, p. 544]. The author claims that he tried applying the same technique to meronymy, but without great success.

Girju *et al.* [9, 10] present a supervised, domain independent approach for the automatic detection of whole-part relations in text. The algorithm identifies lexico-syntactic patterns that encode whole-part relations. The authors report an overall average precision of 80.95% and recall of 75.91%. The authors also state that they came across a large number of difficulties due to the highly ambiguous nature of syntactic constructions.

Van Hage *et al.* [11] developed a method for learning whole-part relations from vocabularies and text sources. The authors reported that they were able to acquire 503 whole-part pairs from the AGROVOC Thesaurus[2] to learn 91 reliable whole-part patterns. They changed the patterns' part arguments with known entities to introduce web-search queries. Corresponding whole entities were then extracted from documents in the query results, with a precision of 74%.

The Espresso algorithm [23] was developed in order to harvest semantic relations in a text. The algorithm extracts surface patterns by connecting the seeds (tuples) in a given corpus. The algorithm obtains a precision of 80% in learning whole-part relations from the Acquaint (TREC-9) newswire text collection, with almost 6 million words.

Some work has already been done on building *knowledge bases* for Portuguese, most of which include the concept of whole-part relations. These knowledge bases are often referred to as *lexical ontologies*, because they have properties of a lexicon as well as properties of an ontology [13, 26]. Well-known, existing lexical ontologies for Portuguese are Portuguese WordNet.PT [18, 19], later extended to WordNet.PT Global (Rede Léxico-Conceptual das Variedades do Português) [20]; MWN.PT-MultiWordNet of Portuguese[3] [25]; PAPEL (Palavras Associadas Porto Editora Linguateca)[4] [22]; and Onto.PT[5] [21]. Some of these ontologies are not freely available for the general public, while others just provide the definitions associated to each lexical entry without the information on whole-part relations. Furthermore, the type of whole-part relation targeted in this work, involving any human entity and its related *Nbp*, can not be adequately captured using those resources (or, at least, only those resources)[6].

[2] http://www.fao.org/agrovoc [last access: 04.05.2014].
[3] http://mwnpt.di.fc.ul.pt/ [last access: 04.05.2014].
[4] http://www.linguateca.pt/PAPEL/ [last access: 04.05.2014].
[5] http://ontopt.dei.uc.pt/ [last access: 04.05.2014].
[6] Only after submission of this paper, we were alerted for the work of Cláudia de Freitas for annotating

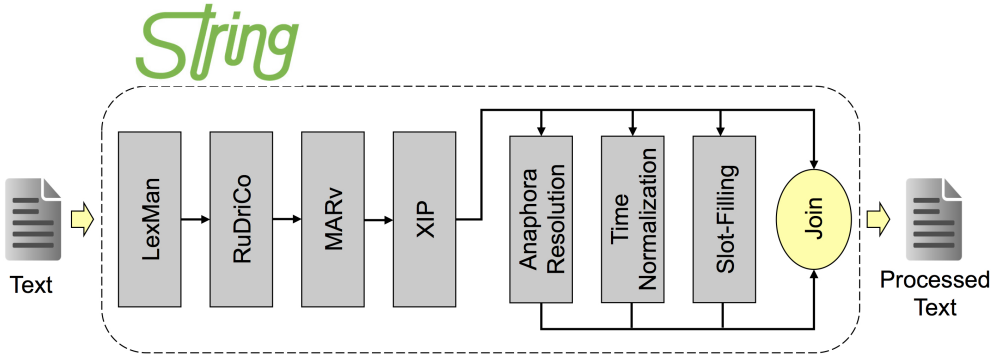


Figure 1 STRING Architecture.

Attention was also paid to two well-known parsers of Portuguee, in order to discern how did they handle the whole-part relations extraction: the PALAVRAS parser [2], consulted using the Visual Interactive Syntax Learning (*VISL*) environment[7], and LX Semantic Role Labeller[8] [3]. Judging from the available on-line versions/demos of these systems, apparently, none of these parsers extracts whole-part relations, at least explicitly.

3 Whole-Part Dependency Extraction Module in STRING

3.1 STRING Overview

STRING [17] is a fully-fledged NLP chain that performs all the basic steps of natural language processing (tokenization, sentence splitting, POS-tagging, POS-disambiguation and parsing) for Portuguese texts. The architecture of STRING is given in Fig. 1.

STRING has a modular, pipe-line structure, where: (i) the preprocessing stage (tokenization, sentence splitting, text normalization) and lexical analysis are performed by LexMan; (ii) followed by RuDriCo, which applies disambiguation rules, handles contractions and several special types of compound words; (iii) the MARv module then performs POS-disambiguation, using HMM and the Viterbi algorithm; and, finally, (iv) the XIP parser (Xerox Incremental Parser) [1] segments sentences into chunks (or elementary sentence constituents: `NP`, `PP`, etc.) and extracts dependency relations among chunks' heads (`SUBJect`, `MODifier`, etc.). XIP also performs named entities recognition (NER). A set of post-parser modules have also been developed to handle certain NLP tasks such as anaphora resolution, temporal expressions' normalization and slot-filling. As part of the parsing process, XIP executes *dependency rules*. Dependency rules extract different types of dependencies between nodes of the sentence chunking tree, namely, the chunks' heads. Dependencies can thus be viewed as equivalent to (or representing) the syntactic relations holding between different elements in a sentence. Some of the dependencies extracted by XIP represent rather complex relations, such as the notion of *subject* (`SUBJ`) or *direct object* (`CDIR`), which imply a higher level of analysis of a given sentence. Other dependencies are much simpler and sometimes quite straightforward, like the determinative dependency `DETD`, holding between an article and the noun it

the human body semantic features in the AC/DC corpora, so we did not considered it here; please refer to: http://www.linguateca.pt/acesso/Esqueleto.pdf [last access: 04.05.2014].

[7] http://beta.visl.sdu.dk/visl/pt/parsing/automatic/dependency.php [last access: 04.05.2014].

[8] http://lxcenter.di.fc.ul.pt/services/en/LXSemanticRoleLabeller.html [last access: 04.05.2014].

determines, *e.g.*, *o livro* (the book) > `DETD(livro,o)`. Some dependencies can also be seen as auxiliary dependencies, and are required to build the more complex ones.

3.2 A Whole-Part Extraction Module in STRING

Next, we describe the way some of whole-part dependencies involving *Nbp* are extracted in the Portuguese grammar for the XIP parser. To this end, a new module of the rule-based grammar was built, which is the first step towards a meronymy extraction module for Portuguese, and it contains most of the rules required for this work. Different typical, syntactic-semantic situations targeted by the meronymy extraction module could be sketched out, but for space limitations only the most simple will be presented here. Example (1) is a simple case where there is a determinative `PP` complement *de N* (of N), so that the meronymy is overtly expressed in the text:

(1) *O Pedro partiu o braço do João* (Pedro broke the arm of João)

The next rule captures the meronymy relation between *João* and *braço* (arm):

```
IF( MOD[POST](#2[UMB-Anatomical-human],#1[human]) & PREPD(#1,?[lemma:de]) &
    CDIR[POST](#3,#2) & ~WHOLE-PART(#1,#2) )
    WHOLE-PART(#1,#2)
```

The rule itself reads as follows: first, the parser determines the existence of a `[MOD]`ifier dependency, already calculated, between an *Nbp* (variable `#2`) and a human noun (variable `#1`); this modifier must also be introduced by preposition *de* (of), which is expressed by the dependency `PREPD`; then, a constraint is defined that the *Nbp* must be a direct object (`CDIR`) of a given verb (variable `#3`); and, finally, that there is still no previously calculated `WHOLE-PART` dependency between the *Nbp* and the human noun (variable `#1`); if all these conditions are met, then, the parser builds the `WHOLE-PART` relation between the human determinative complement and the *Nbp*. A similar rule is required for a dative complement, as in sentence *O Pedro partiu um braço ao João/O pedro partiu-lhe um braço* (Pedro broke him an arm).

Next, in example (2), we present the (apparently) more simple case of a sentence with just a human subject and an *Nbp* direct object:

(2) *O Pedro partiu um braço* (Pedro broke an arm)

In Portuguese, in the absence of a determinative complement, a possessive determiner or a dative complement (eventually reduced to a clitic dative pronoun), sentences like (2) are preferably interpreted as holding a whole-part relation between the human subject and the object *Nbp*. Thus, if there is a subject and a direct complement dependency holding between a verb and a human, on one side, and the verb and an *Nbp*, respectively; and if no `WHOLE-PART` dependency has yet been extracted for that *Nbp*, either for that human subject or another element in the same sentence, then the `WHOLE-PART` dependency is extracted.

Another interesting case is the issue of ambiguity raised by idioms involving *Nbp*. As it is well known, there are many frozen sentences (or idioms) that include *Nbp*. However, for the overall meaning of these expressions, the whole-part relation is often irrelevant, as in the next example:

(3) *O Pedro perdeu a cabeça* (lit: Pedro lost the [=his] head) (Pedro got mad)

The overall meaning of this expression has nothing to do with the *Nbp*, so that, even though we may consider a whole-part relation between *Pedro* and *cabeça* (head), this has no bearing on the semantic representation of the sentence, equivalent in (3) to 'get mad'.

The STRING strategy to deal with this situation is, first, to capture frozen or fixed sentences, and then, after building all whole-part dependencies, exclude/remove only those containing elements that were also involved in fixed sentences' dependencies. In this way, two general modules, for fixed sentences and whole-part relations, can be independently built, while a simple "cleaning" rule removes the cases where meronymy relation is irrelevant (ambiguous idioms, e.g. *à cabeça* (on/at the head), must be addressed in another way). Frozen sentences are initially parsed as any ordinary sentence, and then the idiomatic expression is captured by a special dependency (`FIXED`), which takes as its arguments the main lexical items of the idiom. The number of arguments varies according to the type of idiom. In the example (3) above, this corresponds to the dependency: `FIXED(perdeu,cabeça)`, which is captured by the following rule:

```
IF (VDOMAIN(?,#2[lemma:perder]) & CDIR[post](#2,#3[surface:cabeça])) FIXED(#2,#3)
```

This rule captures any `VDOMAIN`, that is, a verbal chain of auxiliaries and the main verb whose lemma is *perder* (loose), and a post-positioned direct complement whose head is the surface form *cabeça* (head). In order to capture the idioms involving *Nbp*, we built about 400 of such rules, from 10 formal classes of idioms.

4 Evaluation

The first fragment of the CETEMPúblico corpus [27] was used in order to extract sentences that involve *Nbp*. This fragment of the corpus contains 14,715,055 tokens (147,567 types), 6,256,032 (147,511 different) simple words and 260,943 sentences. The existing STRING lexicon of *Nbp* was adapted to be used within the UNITEX corpus processor [24] along with the remaining available resources for European Portuguese, distributed with the system.

Using the *Nbp* (151 lemmas) dictionary 16,746 *Nbp* instances were extracted from the corpus (excluding the ambiguous noun *pelo* (hair) or (by-the), which did not appeared as an *Nbp* in this fragment). Some of these sentences were then excluded for they consisted of incomplete utterances, or included more than one *Nbp* per sentence. A certain number of particularly ambiguous *Nbp*; *e.g.*, *arcada* (arcade), *articulação* (articulation), etc., which showed little or no occurrence at all in the *Nbp* sense, were discarded from the extracted sentences. Finally, the sentences that lacked a full stop were corrected, in order to prevent errors from STRING's sentence splitting module. In the end, a set of 12,659 sentences with *Nbp* was retained for evaluation. Based distribution of the remaining 103 *Nbp*, a random stratified sample of 1,000 sentences was selected, keeping the proportion of their total frequency in the corpus. The output sentences were divided into 4 subsets of 225 sentences each. Each subset was then given to a different annotator, and a common set of 100 sentences was added to each subset in order to assess inter-annotator agreement. For each sentence, the annotators were asked to append the whole-part dependency, as it was previously defined in a set of guidelines, using the XIP format. For example, for (1) the annotators would produce `WHOLE-PART(João,braço)`.

From the 100 sentences that were annotated by all the participants in this process, we calculated the Average Pairwise Percent Agreement, the Fleiss' Kappa [6], and the Cohen's Kappa coefficient of inter-annotator agreement [4] using ReCal3: Reliability Calculator [8], for 3 or more annotators.[9] The four annotators achieved the following results. First, the

[9] http://dfreelon.org/utils/recalfront/recal3/ [last access: 04.05.2014].

Table 1 Average Pairwise Percent Agreement.

Average pairwise percent agr.	Pairwise pct. agr. cols 1 & 4	Pairwise pct. agr. cols 1 & 3	Pairwise pct. agr. cols 1 & 2	Pairwise pct. agr. cols 2 & 4	Pairwise pct. agr. cols 2 & 3	Pairwise pct. agr. cols 3 & 4
85.031%	86.111%	**90.741%**	82.407%	81.481%	80.556%	88.889%

Table 2 Average Pairwise Cohen's Kappa.

Average pairwise CK	Pairwise CK cols 1 & 4	Pairwise CK cols 1 & 3	Pairwise CK cols 1 & 2	Pairwise CK cols 2 & 4	Pairwise CK cols 2 & 3	Pairwise CK cols 3 & 4
0.629	0.65	**0.757**	0.59	0.558	0.518	0.699

Table 3 System's performance for *Nbp*.

Number of sentences	TP	TN	FP	FN	Precision	Recall	F-measure	Accuracy
100	8	73	7	14	0.53	0.36	0.43	0.79
900	73.5	673	55	118	0.57	0.38	0.46	0.81
Total:	81.5	746	62	132	0.57	0.38	0.46	0.81

Average Pairwise Percent Agreement, that is, the percentage of cases each pair of annotators agreed with each other is shown in Table 1. The Average Pairwise Percent Agreement is 85.031%, which is relatively high. Next, the Fleiss' Kappa inter-annotator agreement coefficient was calculated, and it equals 0.625; the observed agreement of 0.85 is higher than expected agreement of 0.601, which we deem as a positive result. Finally, the Average Pairwise Cohen's Kappa is shown in Table 2. The Average Pairwise Cohen's Kappa is 0.629. According to Landis and Koch [16] this figures correspond to the lower bound of the "substantial" agreement; however, according to Fleiss [7], these results correspond to an inter-annotator agreement halfway between "fair" and "good".

In view of these results, we can assume as a reasonable expectation that the remaining, independent and non-overlapping annotation of the corpus by the four annotators is sufficiently consistent, and will use it for the evaluation of the system output.

The system performance was evaluated using the usual evaluation metrics of Precision, Recall, F-measure, and Accuracy. The results are shown in Table 3, where `TP`=*true-positives*; `TN`=*true-negatives*; `FP`=*false-positives*; `FN`=*false-negatives*. The number of instances (`TP`, `TN`, `FP` and `FN`) is higher than the number of sentences, as one sentence may involve several instances. The relative percentages of the `TP`, `TN`, `FP` and `FN` instances are similar between the 100 and the 900 set of sentences. This explains the similarity of the evaluation results and seems to confirm our decision to use the remaining 900 sentences' set as a golden standard for the evaluation of the system's output with enough confidence. The recall is relatively small (0.38), which can be explained by the fact that in many sentences, the *whole* and the *part* are not syntactically related and are quite far away from each other. Precision is somewhat better (0.57). The accuracy is relatively high (0.81) since there is a large number of *true-negative* cases.

5 Conclusions

This paper addressed the problem of whole-part relations extraction involving human entities and body-part nouns (*Nbp*) in Portuguese. A rule-based meronymy extraction module has been built and integrated in the grammar of the STRING system. It contains 27 general rules addressing the most relevant syntactic constructions triggering this type of meronymic relations. A set of 400 rules had also been devised to prevent the whole-part relations being extracted in the case the *Nbp* are elements of idiomatic expressions. From a relatively large corpus, about 17 thousand sentences with *Nbp* were extracted. A stratified random sample of 1,000 sentences was independently annotated by 4 Portuguese native speakers in order to produce a golden standard and confront it against the system's output. The results show 0.57 precision, 0.38 recall, 0.46 F-measure, and 0.81 accuracy. In future work, we intent to improve recall by focusing on the *false-negative* cases already found, which shown that several syntactic patterns have not been paid enough attention, such as coordination.

Acknowledgements. This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013; and Erasmus Mundus Action 2 2011-2574 Triple I – Integration, Interaction and Institutions.

We would like to thank the comments of the anonymous reviewers, which helped to improve this paper.

References

1 S. Ait-Mokhtar, J. Chanod, and C. Roux. Robustness beyond shallowness: incremental dependency parsing. *Natural Language Engineering*, 8(2/3):121–144, 2002.

2 E. Bick. *The Parsing System "Palavras": Automatic Grammatical Analysis of Portuguese in a Constraint Grammar Framework*. PhD thesis, Aarhus Univ. Aarhus, Denmark: Aarhus Univ. Press, 2000.

3 A. Branco and F. Costa. A Deep Linguistic Processing Grammar for Portuguese. In Pardo et al., editor, *Computational Processing of Portuguese*, LNAI 6001, pages 86–89. Springer, 2010.

4 J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

5 C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT, Cambridge, 1998.

6 J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psych. Bull.*, 76(5):378–382, 1971.

7 J. L. Fleiss. *Statistical methods for rates and proportions (2nd ed.)*. New York: John Wiley, 1981.

8 D. Freelon. ReCal: Intercoder Reliability Calculation as a Web Service. *Intl. J. of Internet Science*, 5(1):20–33, 2010.

9 R. Girju, A. Badulescu, and D. Moldovan. Learning Semantic Constraints for the Automatic Discovery of Part-Whole Relations. In *Proceedings of HLT-NAACL*, volume 3, pages 80–87, 2003.

10 R. Girju, A. Badulescu, and D. Moldovan. Automatic discovery of part-whole relations. *Computational Linguistics*, 21(1):83–135, 2006.

11 W. Van Hage, H. Kolb, and G. Schreiber. A method for learning part-whole relations. *The Semantic Web – ISWC 2006, LNAI/LNCS*, 4273:723–725, 2006.

12 M. Hearst. Automatic acquisition of hyponyms from large text corpora. In *Proceedings of the 14th conf. on Computational linguistics*, volume 2 of *COLING 92*, pages 539–545. ACL Morristown, NJ, USA, 1992.

13 G. Hirst. Ontology and the lexicon. In S. Staab and R. Studer, editors, *Handbook on Ontologies*, pages 209–230. Springer, 2004.

14 M. Iris, B. Litowitz, and M. Evens. Problems of the Part-Whole Relation. In M. Evens, editor, *Relational Models of the Lexicon: Representing Knowledge in Semantic Networks*, pages 261–288. Cambridge Univ. Press, 1988.

15 C. Khoo and J.-C. Na. Semantic Relations in Information Science. *Annual Review of Information Science and Technology*, 40:157–229, 2006.

16 J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977.

17 N. Mamede, J. Baptista, C. Diniz, and V. Cabarrão. STRING: An Hybrid Statistical and Rule-Based Natural Language Processing Chain for Portuguese. In *Intl. Conf. on Computational Processing of Portuguese (PROPOR 2012)*, volume Demo Session, Paper available at `http://www.propor2012.org/demos/DemoSTRING.pdf`, 2012.

18 P. Marrafa. *WordNet do Português: uma base de dados de conhecimento linguístico*. Instituto Camões, 2001.

19 P. Marrafa. Portuguese WordNet: general architecture and internal semantic relations. *DELTA*, 18:131–146, 2002.

20 P. Marrafa, R. Amaro, and S. Mendes. WordNet.PT Global – extending WordNet.PT to Portuguese varieties. In *Proceedings of the 1st Workshop on Algorithms and Resources for Modelling of Dialects and Language Varieties*, pages 70–74, Edinburgh, Scotland. ACL Press, 2011.

21 H. Gonçalo Oliveira. *Onto.PT: Towards the Automatic Construction of a Lexical Ontology for Portuguese*. PhD thesis, Univ. of Coimbra/FST, 2012.

22 H. Gonçalo Oliveira, P. Gomes, D. Santos, and N. Seco. PAPEL: A Dictionary-based Lexical Ontology for Portuguese. In *Computational Processing of the Portuguese Language, 8th Intl. Conf., Proceedings (PROPOR 2008)*, volume 5190, pages 31–40, Aveiro, Portugal. Springer, 2008.

23 P. Pantel and M. Pennacchiotti. Espresso: Leveraging generic patterns for automatically harvesting semantic relations. In *Proceedings of Conf. on Computational Linguistics/ACL (COLING/ACL-06)*, pages 113–120. Sydney, Australia, 2006.

24 S. Paumier. *Unitex 3.1.beta, User Manual*. Univ. Paris-Est Marne-la-Vallée, 2014.

25 E. Pianta, L. Bentivogli, and C. Girardi. MultiWordNet: developing an aligned multilingual database. In *1st Intl. Conf. on Global WordNet*, 2002.

26 L. Prévot, C.-R. Huang, N. Calzolari, A. Gangemi, A. Lenci, and A. Oltramari. Ontology and the lexicon: a multi-disciplinary perspective (introduction). In C.-R. Huang, N. Calzolari, A. Gangemi, A. Lenci, A. Oltramari, and L. Prévot, editors, *Ontology and the Lexicon: A Natural Language Processing Perspective*, Studies in Natural Language Processing, chapter 1, pages 3–24. Cambridge Univ. Press, 2010.

27 P. Rocha and D. Santos. CETEMPúblico: Um corpus de grandes dimensões de linguagem jornalística portuguesa. In M. G. Nunes, editor, *V Encontro para o processamento computacional da língua portuguesa escrita e falada (PROPOR 2000)*, pages 131–140. São Paulo: ICMC/USP, 2000.

Part VI

Natural Language Processing Tools and Resources

3^{rd} Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Automatic Detection of Proverbs and their Variants

Amanda P. Rassi[1,2], Jorge Baptista[2], and Oto Vale[1]

1 **Federal University of São Carlos-UFSCar**
Rodovia Washington Luís, km 235 – SP-310. São Carlos – São Paulo – Brasil
CEP 13565-905
aprassi@ualg.pt,otovale@ufscar.br

2 **University of Algarve-FSCH/CECL**
Campus de Gambelas, 8005-139 Faro, Portugal
jbaptis@ualg.pt

Abstract

This article presents the task of automatic detection of proverbs in Brazilian Portuguese, from the intersection of the regular syntactic structure of proverbs and their core elements. We created finite-state automata that enabled us to look for these word combinations in running texts. The rationale behind this method consists in the fact that although proverbs may have a normal sentence structure and often a very commonly used lexicon, their specific word-combinations may enables us to identify them and their variants irrespective of the syntactic or structural changes the proverb may undergo. The goal of this task is to gather the largest number of proverbs and their variants. The results showed precision 60.15%.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases Brazilian Portuguese, proverbs, syntactic structure, core element, variation

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.235

1 Introduction

The existence of proverbial structures in texts, including journalistic texts, is indisputable [12], which raises the problem of identifying them as a complex structure. The main problem concerning the identification of proverbs is that they have the same syntactic structure and the same words as ordinary, free sentences, however, they normally have a non compositional meaning and must be recognized not as an ordinary string of words, but as a complex unit, formed by several words, phrases and even multiple clauses. In this sense, proverbs resemble multiword expressions (MWE), although some authors [13, p.53] consider them as a different type of linguistic units as a quoted speech inside speech itself. In this paper, we adopt the view that proverbs should be treated as MWE.

In general, automatic processing of idiomatic expressions, fixed expressions, semi-fixed expressions, proverbs and other multiword expressions is still a hard task for Natural Language Processing (NLP) [30]. Although there are many studies about the identification of multiword expressions in NLP [20, 21, 23], it is still difficult to identify them automatically in natural language texts [4, 5, 26].

In this paper we focus on the special case of proverbs in view of a double problem they represent to NLP: the fact that proverbs accept both lexical and formal (structural) variation. We aim at developing a method for automatic detection of proverbs and their variants, based


© Amanda P. Rassi, Jorge Baptista, and Oto Vale;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 235–249

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

on existing compilations of proverbs, by exploring the regular syntactic structures that most proverbs present. These regularities led to a formal classification of proverbs, based on their syntactic structure. Finite-state automata will be used to represent the regular patterns found in these classes of proverbs. Results from the automatic identification of Brazilian Portuguese proverbs from real texts are presented. This approach can be used in to two main applications: for lexicographic work, in order to build more complete dictionaries, and for Natural Language Processing, to improve linguistic resources, tools and applications, by allowing systems to signal these micro-texts and a special type of discursive element.

2 Delimitation of the Object

Proverbs, parables, adages, aphorisms, maxims, and so on, these are all different terms used to designate similar types of sentences. Though there are conceptual differences among these terms, in practice, many authors ignore such distinctions and tend to group all these linguistic expressions under the broad umbrella term of *proverb*. In this paper, we also adopt such broad perspective and will consider proverbs as linguistic expressions forming fixed word combinations, in spite of some (limited) lexical or structural variation, often with a sentential status, that may even include subclauses, and whose global meaning is often idiomatic. These micro-texts are usually generic statements, conveying a world view or stating a moral judgement, an eternal truth, an ideal state of affairs.

We distinguish proverbs from *fixed expressions/frozen sentences* (or *idioms*, proper). In idioms, the verb and one of its argument positions are frozen together, that is, they are distributionally invariant, or the argument nouns can only vary within a small and closed paradigm. Usually the subject of frozen sentences is distributionally free, and its selection depends not just on the verb, but on the overall meaning of the combination of the verb and its frozen arguments; *i.e. Ana/Essa mesa não vale um tostão* 'Ana/This table is not worthy a penny'. On the other hand, typically, proverbs are completely frozen sentences, where, in spite of some (reduced) lexical variation and some (even more constraint) syntactical paraphrasing, all the elements are fixed. In other words, proverbs have the subject position necessarily filled by a fixed element [18, p.161], while the subject in fixed expressions usually varies and may be defined intensionally, by distributional constraints.

The second property that distinguishes proverbs and fixed expressions is, according to [24], that the proverbs "always have an autonomous semantic value in communicative terms, unlike idioms that are only constituents of sentences and may never occur as a full sentence." In this sense, proverbs take place in whole sentences while fixed expressions only replace phrases (nominal phrase, verbal phrase or prepositional phrase).

Although proverbs have syntactic structures similar to simple sentences, they can not be recognized as common sentences, but must be understood as a single block, whose syntactic slots should always be filled by specific lexical units. It means that proverbs are formed by words and phrases like any other free sentences, but they must be understood as a complex expression, a combination of words whose use is highly constraint.

When proverbs are introduced by an enunciative mark, such as *como dizem* 'as they say', *como dizia minha avó* 'as my grandmother used to say', *dizem por aí* 'people say/they say', *costuma dizer-se* 'it is often said', etc.; it is then easier to identify them because these type of marks can be extensively described. However, there is often no mark at all introducing proverbs in texts, which renders their spotting more difficult.

Finally, proverbs are prone to certain types of formal variation, particular ellipsis of one of its clause-type components, and they often undergo stylistic reformulation, in order to produce some perlocutionary effect. For example, a banking institution, in one advertisement

of its products, recently "reinvented" the proverb *Tempo é dinheiro* 'Time is money' as *Tempo não é só dinheiro. É valor* 'Time is not just money. It is value'. This capacity of the proverbs to be reinterpreted and reformulated, which some linguists called "défigement" or "unfreezing" is an inherent part of the paremiologic dynamics in language.

3 Related Works

Most of the work done on Brazilian Portuguese proverbs adopt a didatic or pedagogic approach, [14, 25, 31], or analyzes rhetorical relations between the clauses [15, 16, 17]. We did not find any work that describes formally proverb structures in Portuguese or that tried to identify them automatically in large corpus.

For European Portuguese, Lucília Chacoto developed many studies on proverbs, either theoretical and practical works. The author compared Portuguese and Spanish proverbs initiated by *Quem/Quien* 'Who' [6] and also analyzed comparative structures [7] which are two of the structures we describe in this paper.

We can also cite works for other languages, like Lacavalla [22], who compared proverbs initiating by *Quand/Quando* 'When' in Italian and French. The author uses local grammars for searching the proverbs in both languages and describes the data in Lexicon-Grammar Tables, analyzing all syntactic properties and distribution of those units. On the other hand, Navarro Brotons [2] compared proverbs in Spanish and French. The author analyzed syntax, semantics and translation of proverbs and their variants in both languages and also described the data in Lexicon-Grammar tables.

We also cite the extensive work of Mirella Conenna [8, 9, 10, 11], who produced many works about proverbs in French and Italian, comparing their structures in both languages, classifying proverbs in syntactic tables, *i.e.* Lexicon-Grammar tables, and analyzing proverbs and their variants in equivalence classes. In all those works, the author was concerned about the formalization of the data for automatic identification and processing.

There are also some other publications about proverbs in Brazilian Portuguese, but they do not present any systematic analysis. These include didactic materials used in schools, dictionaries, glossaries, and lists of proverbs. Most of them are used in teaching/learning Portuguese as second language or as didactic manuals.

For Brazilian Portuguese it is still necessary to describe formally syntactic structures of the proverbs and their core elements, aiming to contributing for the construction of lexicon-syntactic resources applicable in NLP.

4 Methods

In this section we present a methodology for automatic detection of proverbs and their variants, tested on a Brazilian Portuguese corpus, which can be resumed in 6 steps: (i) creating a database with proverbs searched in dictionaries and other lists; (ii) defining syntactic criteria to organize the collected proverbs into formal classes; (iii) manually identifying the POS tags of their elements; (iv) generating tables with the core elements derived from POS tagging; (v) creating graphs with the basic structure for each class; and (vi) intersecting the graphs with the tables of the proverbs' core elements to produce finite-state transducers that will enable us to identify such word combination in texts. After these steps, we could find other proverbs and their semantic variations within the same syntactic structure.

We searched for the proverbs and their variants in PLN.BR Full corpus [3], which contains 103,080 texts, with 29,014,089 tokens, from *Folha de São Paulo*, a Brazilian newspaper, from 1994 to 2005.

4.1 Collection of Proverbs

The first step for this work consists in creating a list of proverbs that will serve as input seeds to recognize other proverbs and their variants in large corpora. Five different sources were used: a list of proverbs in Wikipedia, three books with proverbs collections [29, 32, 34] and a dictionary of proverbs [19].

Firstly, all the expressions collected in these sources were analyzed manually and many were discarded as they were not considered as proverbs but consist mostly of idiomatic expressions (or idioms), like (1), or aphorisms and maxims, as in (2):

(1) *Matar dois coelhos com uma cajadada só*
[to] kill two bunnies with just one thwack 'kill two birds with a stone'

(2) *Na natureza, nada se cria nada se perde, tudo se transforma*
'In Nature, nothing is created, nothing is lost, everything is transformed'

The idiom in (1) is a frozen sentence with a free subject slot and two frozen complements, a direct object and an instrumental complement [1, 18, 35](class C1P2). On the other hand, (2) is an aphorism or maxim, attributed to the chemist Lavoisier (1743-1794) about the conservation of mass. In spite of its three-clause, parallelistic, proverb-like structure, and its generic nature, the (known) authorship of the maxim lead us to discard it from our study.

After a substantial collection of over 3,502 proverbs (and their variants) has been gathered, the variants of each proverb were grouped together and one of them was selected to be considered as the entry of our lexicon (or its base-form), based on its frequency among the sources consulted. Most differences between variants of the same proverb consist in the variation of their grammatical elements, and the lexical choices for their core meaningful words.

Finally, we tried to confirm whether these proverbs were (still) really in use in current Brazilian Portuguese, checking them with 5 native speakers of Brazilian Portuguese from different geographic regions.[1] Some proverbs are only used in Portugal or in Portuguese-speaking African countries, while others are very old and probably may not be in use anymore.

From the original 3,502 proverbs (and their variants), a final list of 594 proverbs (*types* or *base-forms*) was compiled.[2]

4.2 Classifying Proverbs and POS Tagging their Elements

The list of proverbs (base-forms) was then classified into formal classes. This classification was based on the following criteria, applied in this order:

(i) the number of propositions (one, two, or three clauses or clause-like units);
(ii) coordination (in multiple-clause proverbs);
(iii) order of the main vs the subordinate clauses (in multiple-clause proverbs);
(iv) order of the constituents (in single-clause proverbs);
(v) impersonal constructions; and
(vi) obligatory negation.

Table 1 presents the current classification.

[1] We consider that the sampling by region is not sufficient to confirm the presence or absence of proverbs, and we would need to consult speakers from different genders, ages, social classes, education levels etc, this is out of the main scope of this work.

[2] The list of proverbs and their classification can be consulted at the first author profile in ResearchGate, available in `https://www.researchgate.net/project/PB-proverbs`.

Table 1 Formal Classification of Brazilian Portuguese Proverbs.

Class	Structure	Example (approximate translation)	Types
P1F1	*Ø V w* (impersonal)	*Não há crime sem lei* 'There is no crime without law'	20
P1F2	N_0 *Vcop Adj/N w*	*A carne é fraca* 'The flesh is weak'	53
P1F3	N_0 *V w*	*O hábito (não) faz o monge* 'The cloth (does not) make the monk'	80
P1F4	N_0 *Neg V w*	*Burro velho não aprende línguas* 'Old donkey does not learn languages'	53
P1F5	*Prep* N_i N_0 *V w* (fronted prep. phrase)	*Para bom entendedor, meia palavra basta* 'For the one who understands, half word is enough'	45
P2F1	F_1 *Conjs-comp* F_2 (comparatives)	*Mais vale um pássaro na mão do que dois voando* 'Beter is is a bird in the hand than two flying'	39
P2F2	F_1 *Conjc* F_2 (coordinated)	*A palavra é de prata e o silêncio é de ouro* 'The word is silver and the silence is gold'	71
P2F3	N_1, N_2	*Tal pai, tal filho* 'Like father, like son'	48
P2F4	*Qu-* F_1 F_2 (interrogative subclass)	*Quem tem boca vai a Roma* 'Who has a mouth goes to Rome'	90
P2F5	F_1 *Conjs* F_2 (subordinated)	*Os amigos são muitos quando grande é a abastança* 'Friends are many when abundance is great'	20
P2F6	*Conjs* F_2, F_1 (fronted subord.)	*Quando a esmola é demais, o santo desconfia* 'When alms are too much, the saint gets suspicious'	28
P3	F_1, F_2, F_3	*Um é pouco, dois é bom, três é demais* 'One is little, two is good, three is too much'	47
Total			**594**

Some remarks on this classification are in order:

(i) impersonal constructions involve the verb *haver* 'there be' and *ter* 'to have' with impersonal valency (the later only exists in Brazilian Portuguese);

(ii) sentences with copula verbs *ser* and *estar* 'to be' usually present an adjectival or nominal predicate; these sometimes allow for mirror permutation (*A carne é fraca* = *fraca é carne*[3] 'The flesh is weak');

(iii) proverbs with obligatory negation usually involve negation adverbs, *e.g. não* 'no/not', *nunca* 'never', *jamais* 'never', *nem* 'nor', etc.; negation has precedence over copula verbs, so that proverbs with negated copula were included in this class;

(iv) single-clause proverbs with a fronted prepositional phrase do not admit the basic word-order;

(v) comparative proverbs, including those with subordinate sub-clause, are a type of complex sentences, though other types of comparative structures were also included in this class;

(vi) nominal propositions named N_1, N_2 (in P2F3 class) are treated as clausal propositions, even if they may contain no verbs and only have a 'clausal' or 'propositional content'.

[3] `http://rainhadocarmelo.blogspot.pt/2010_02_01_archive.html` [2014-03-08 13:11]

After classifying the proverbs, we manually annotated their elements for part-of-speech (POS) tags. Since each class is syntactically homogeneous, it was then relatively simple to organize the lexical items in a tabular format, so that the characteristic elements of the proverbs may be aligned, and can easily be identified. For the noun phrases (*NP*), either the subject (N_0) or the complement (N_1), the head noun (or pronoun) is determined, and eventual determiners (*Det*) or modifiers (*Mod*) are tagged and distributed across the corresponding columns. Eventual pre- or post-modifiers of verbs (*Deus escreve* direito *por linhas tortas* 'God writes straight with crooked lines'), including obligatory auxiliary verbs (*Não se entra em briga que não se* pode *ganhar* 'Do not enter into a fight you can not win'), and other elements, such as the impersonal pronouns (*Aqui se faz, aqui se paga* 'Here you do, here you pay')[4], or obligatory negation (*Quem não tem cão caça com gato* 'Who does not have a dog hunts with a cat') are also taken into consideration. Subordinative or coordinative elements are also provided with an adequate slot. In this way, it is relatively simple to automatically extract the core (or more representative) elements from each proverb, based on the classes' formal homogeneity.

4.3 Extracting Core Elements

In order to extract the core words in each proverb, we analyzed all cells in each table and selected as core elements the most frequent grammatical classes in each syntactic position. For example, in almost all classes[5] the initial *NP* is necessarily filled by a noun or, in rare cases, a pronoun. The noun can be accompany by determinants and/or adjectives and/or other nominal adjuncts, but the only position that is fully filled by some element is the column <N> either in the subject or in the complement position, so we selected the item instantiated in column <N> as one of the core elements for identifying the proverb.

In all classes[6], *VP* position is necessarily filled by a verb, so this is selected as a key element in the constitution of the proverbs. Table 2 shows a sample of P1F3 class, in a tabular format, indicating all columns[7].

Depending on the formal class of the proverbs, so the core elements are defined. In the case of class P1F2, the definitory elements are the heads of the subject and of the predicative complement (noun or adjective) as well as the copula verb. In the case the head in null (*e.g. Os últimos serão os primeiros* 'The first shall be the last') the determiner or an adjective may be chosen instead. In comparative proverbs, there is often no main verb, so the determiners 4.3 or the comparative conjunctions 4.3 must be selected, along with the core nouns:

(3) ***Tal*** *pai* ***tal*** *filho*
'**Like** father **like** son'

(4) ***Nem tanto*** *ao mar* ***nem tanto*** *à terra*
'**Not so much** to sea **not so much** to ground'

---

[4] In Portuguese, impersonal clitic pronoun *-se* imposes 3^{rd} person-singular agreement to the verb, thus being indistinguishable from passive-like pronominal constructions. Only some few clear-cut cases of pronominal passives were found; e.g. *Entre mortos e feridos salvaram-se todos* 'Among dead and wounded all were saved'. Both strategies may be considered as a form of subject (agent) degenerescence, hence contributing to the generic effect of the proverbs.

[5] Exception done for class P1F1, which has no explicit subject (null subject).

[6] Exception done for class P2F3, which is constituted by nominal phrases only, and has no verb.

[7] In this table the headings are read as follows: Adj = Adjective, Adv = Adverb, Det = Determinant, Indet_Pass = Pronominal passive-like construction, N = Noun, Prep = Preposition, V = Verb; the words inside chevrons correspond to lemmas

Table 2 Sample of class P1F3.

Proverb		Det	Adj	N	Adj	Indet_Pass	V	Adv	Prep	Det	Adj	N	Adj
A adversidade faz os heróis	<o>	-		<adversidade>	-	-	<fazer>	-	-	<o>	-	<herói>	-
A ambição cega a razão	<o>	-		<ambição>	-	-	<cegar>	-	-	<o>	-	<razão>	-
A intenção faz o agravo	<o>	-		<intenção>	-	-	<fazer>	-	-	<o>	-	<agravo>	-
A justiça começa em casa	<o>	-		<justiça>	-	-	<começar>	-	em	-	-	<casa>	-
A ocasião faz o ladrão	<o>	-		<ocasião>	-	-	<fazer>	-	-	<o>	-	<ladrão>	-
A união faz a força	<o>	-		<união>	-	-	<fazer>	-	-	<o>	-	<força>	-
As aparências enganam	<o>	-		<aparência>	-	-	<enganar>	-	-	-	-	-	-
As más notícias chegam depressa	<o>	<mau>		<notícia>	-	-	<chegar>	depressa	-	-	-	-	-
As paredes têm ouvidos	<o>	-		<parede>	-	-	<ter>	-	-	-	-	<ouvido>	-
Boas contas fazem bons amigos	-	<bom>		<conta>	-	-	<fazer>	-	-	-	<bom>	<amigo>	-
Deus escreve certo por linhas tortas	-	-		<deus>	-	-	<escrever>	certo	por	-	-	<linha>	<torto>
Mentira tem perna curta	-	-		<mentira>	-	-	<ter>	-	-	-	-	<perna>	<curto>
Muitos cozinheiros estragam a sopa	<muito>	-		<cozinheiro>	-	-	<estragar>	-	-	<o>	-	<sopa>	-
O abismo atrai o abismo	<o>	-		<abismo>	-	-	<atrair>	-	-	<o>	-	<abismo>	-
O hábito faz o monge	<o>	-		<hábito>	-	-	<fazer>	-	-	<o>	-	<monge>	-
O justo paga pelo pecador	<o>	-		<justo>	-	-	<pagar>	-	por	<o>	-	<pecador>	-
O peixe se conhece pela boca	<o>	-		<peixe>	-	se	<conhecer>	-	por	<o>	-	<boca>	-
Os fins justificam os meios	<o>	-		<fim>	-	-	<justificar>	-	-	<o>	-	<meio>	-
Roupa suja se lava em casa	-	-		<roupa>	<sujo>	se	<lavar>	-	em	-	-	<casa>	-

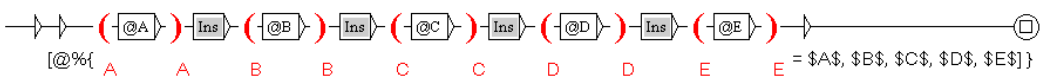


Figure 1 Reference graph for class P2F4.

In the common cases where a lexical element of the proverb allows for variation, all the variants are included in the corresponding slot. This is the case of the proverb *Cachorro mordido de cobra tem medo de linguiça* 'Dog bitten by a snake is afraid of sausage' where the second noun can be replaced by *barbante* 'string' and *salsicha* 'sausage'; notice, however, that the variation of grammatical elements 4.3 was ignored:[8]

(5) *Cachorro (que foi + <E>) mordido (de + por) cobra tem medo até de (barbante + salsicha + linguiça)*
'Dog (that was + <E>) bitten by a snake is afraid of (string + sausage + pork sausage)'

4.4 Creating and Applying the Graphs

Once the characteristic elements of each proverb have been identified, they were structured in a tabular format, one table for each class (residual class "others" was not considered in this paper). Then, using the Unitex 3.1.beta linguistic development platform [27, 28], we produce a reference graph for each class. Fig. 1 illustrates the graph for class P2F4, corresponding to proverbs with a fronted subordinated clause; e.g. *Se queres conhecer o vilão, põe-lhe um pau na mão* 'If you want to know a villain, put a stick in his hand'.

This graph reads as follows: the system explores systematically each line in the table of a class core elements, replacing the variables *@A*, *@B*, etc, by the corresponding content of columns A, B, etc. These input variables are then associated to output variables (in the letters below the brackets) to be reused in the output. In this case, the graph delimits the matched expression by brackets, and produced the content in a normalized form, introduced by the idiom number (the table's line number), represented by variable @%[9]. By intersecting the reference graph with the corresponding table, the system generates one subgraph for each line of the table, and a general result graph, containing all the subgraphs. The result graph can then be used to find patterns in texts. Table 3 shows a sample of a concordance of such matched strings from the PLN.Br corpus.

Each line in the table has been numbered. In this concordance, a small left context is provided, followed by the number of the proverb type in the corresponding class, the actual words in the corpus and the core words that the transducer detected; empty variables are not represented (void commas).

The table presents two matches that are considered False Positives, in lines 16 and 17. The proverb supposed to be found is *Quem sabe faz* 'Who knows makes', but the system found, for example, a free sentence (line 16) and a verse of a brazilian song (line 17). It is also remarkable the transformations (actualizations or adaptations) created by speakers. The proverb we were looking for is *Quem vê cara não vê coração* 'Who sees the face does not see the heart' as in line 22, but the speaker adapted the proverb to the context of smoking and created *Quem vê cara não vê pulmão* 'Who sees the face does not see the lung', as

[8] The items linked by "+" inside parentheses can comute in the given syntactic slot; the symbol <E> represents the empty string.
[9] The shadowed box **Ins** is a subgraph defining a window of 0 to 3 words and separators allowed between the proverbs' core elements.

Table 3 Sample of a concordance of Class P2F4.

1	é o	[0003	barato que pode sair caro=barato, caro„,]
2	não	[0006	mata engorda=mata, engorda„,]
3	Quem	[0015	avisa amigo é=avisa, amigo„,]
4	Quem	[0018	cala consente=cala, consente„,]
5	Quem	[0019	Canta Seus Males Espanta=Canta, Males, Espanta„]
6	e como	[0020	casei e quero casa=casei, quero, casa„]
7	quem	[0023	conta um conto aumenta um ponto=conta, conto, aumenta, ponto,]
8	quem	[0028	diz o que quer ouve o que não quer=diz, quer, ouve, quer,]
9	não	[0042	arrisca não só não petisca=arrisca, petisca„,]
10	que não	[0043	choram nem mamam=choram, mamam„,]
11	não	[0044	deve não teme=deve, teme„,]
12	Quem	[0047	está dentro quer sair e quem está fora não=está, dentro, quer, sair,]
13	não	[0050	sabe não ensina=sabe, ensina„,]
14	quem	[0062	pariu Mateus que o embale=pariu, Mateus, embale„]
15	quem	[0064	procura acha=procura, acha„,]
16	Quem	[0068	sabe alguém faz uma experiência com isso=sabe, faz„,]
17	quem	[0068	sabe faz a hora=sabe, faz„,]
18	Quem	[0068	Sabe Faz ao Vivo=Sabe, Faz„,]
19	Quem	[0069	sabe sabe=sabe, sabe„,]
20	os que	[0070	semeiam ventos colhem tempestades=semeiam, ventos, colhem, tempestades,]
21	"Quem	[0079	tem pressa come cru=tem, pressa, come, cru,]
22	"quem	[0085	vê cara não vê coração=vê, cara, coração„]
23	quem	[0085	vê cara não vê pulmão=vê, cara, vê„]
24	Quem	[0085	vê cara vê muito mais do que coração=vê, cara, vê, coração,]
25	Quem	[0086	viver verá=viver, verá„,]

in line 23. In 24 the obligatory negation of the original proverb has been deleted and the meaning actually inverted in a creative way.

In this way it was possible to find other variants of proverbs than those we had previously collected (from books, dictionaries and the wikipedia) and find several instances of creative reuse and transformations of proverbs for rethoric purposes.

5 Results and Discussion

Since, to our knowledge, there is no available corpus annotated with proverbs and similar expressions, only precision was reported here.

From the previous list of 594 proverbs, 788 matches were found in the PLN.Br corpus, from which 474 matches (60.15%) correspond to actual proverbs. We decided to search these lexical units in journalistic corpus aiming to check if in the common language they also appear. It has been proved [33] that literary corpora contain a large number of proverbs, but the challenge is looking for them in non-literary texts. Table 4 shows the breakdown of these results by class. In spite of the number of matches, only 137 types (different proverbs) were found. The scarcity of the occurrence of proverbs in the corpus (1:36,820 words), as well as its reduced variety (23% types) is most probably linked to the journalist nature of the corpus.

In this respect, it is remarkable the number of instances retrieved from the data in class P2F4 as well as its low precision (27.5%). This class includes only two lexical items, besides the indefinite subject pronoun *quem* 'who', as in *Quem cala consente* '[he] who silence [gives

Table 4 Results of automatic identification of proverbs by class.

Class	Proverbs (types)	Matches	Types	False-Positives
P1F1	20	15	4	2
P1F2	53	91	21	16
P1F3	80	153	24	55
P1F4	53	61	15	0
P1F5	45	63	5	6
P2F1	39	40	7	1
P2F2	71	14	3	9
P2F3	48	40	8	25
P2F4	90	276	37	200
P2F5	20	3	1	0
P2F6	28	1	1	0
P3	47	31	11	0
Total	**594**	**788**	**137**	**314**

his] consent'. Since these are very short proverbs, a window of 5 words between the core elements may be inadequate.

We repeated the experiment without any insertion window, and captured 56 matches, of which 26 were false positives. The local precision of the class P2F4 raised from 27.5% to 53.57%. Considering the global precision (including all classes), global precision raised from 60.15% to 73.35%. This may indicate that, depending of the syntactic structure of the proverb, a more or less wide window between the core elements must be defined.

The system matched 137 different proverbs from the previous list with 594 entries, and their distribution is presented in Fig. 2, below. Some few other proverbs have higher frequencies but they were collapsed in Fig. 2 because they form a small number of proverbs with relatively high frequency.[10]

The small number of different proverbs matched by the system (23% of the total types) is probably due to the nature of the corpus. Some proverbs, as we will see below, have been adapted and reconfigured to fit the discursive needs of the author.

[10] Namely, f=13, f=16, f=20, f=22, f=44, f=52, f=55 and f=88.

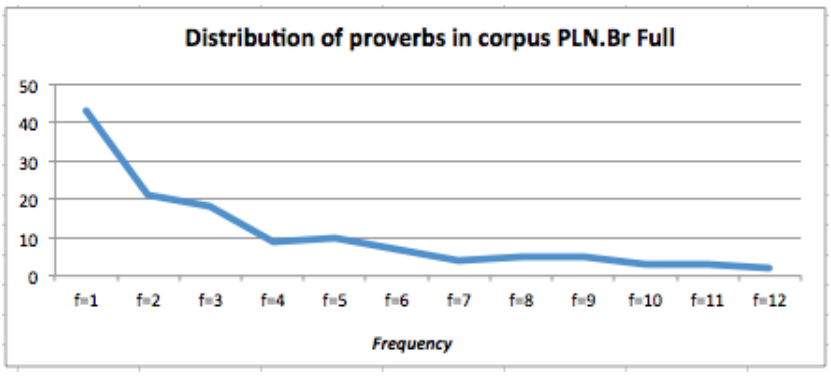


Figure 2 Distribution of proverbs in corpus PLN.Br Full.

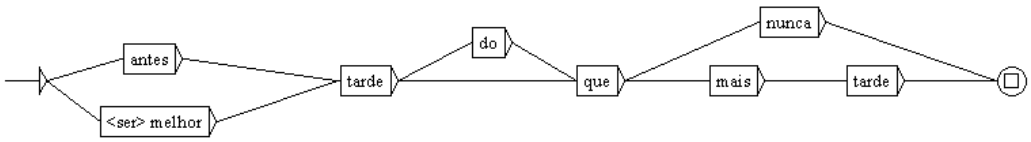


Figure 3 Graph with variants of the proverb *Antes tarde do que nunca* 'Better later than never'.

The matches found allowed us to identify other variants of the same proverb that were not in the initial list. For example, along the form *Antes tarde do que nunca* 'Better later than never', the variants can be represented by the graph presented in Figure 3.

It was also possible to find proverbs that were not in the previous list. For example, we used the structure [*quem V V*] ['who V V'], which was searched in Unitex by the following regular expression: *quem* (<MOT>+<E>)(<V:P3s>+<V:J3s>) (<MOT>+<E>) <V:P3s>. This syntaxe means: pronoun *quem* followed by a verb in the third singular person of the verb in simple present or simple past, which is followed by a verb in simple present in third singular person; between these elements a single, facultative word could also appear. This regular expression could be instantiated by *Quem sabe faz* 'Who knows makes' 5 and another similar syntactic structure was found 5:

(6) *Quem sabe faz*
'Who knows makes'

(7) *Quem sabe faz ao vivo*
'Who knows makes it viva'

These are two different proverbs, not only variants, because their meanings are different, so the task is also valid for searching more proverbs.

While the definition of the core elements is basically a lexical decision, the length of the insertion window between them is a matter of empirical decision, and it can vary, as we have seen, depending on the type of proverb involved. Several tests were conducted with insertion windows of different lengths, and, in general, results fell rapidly when more than 5 words could be inserted. The two examples 5–5, below, show 5 words between the core elements.

(8) *o buraco* [das negociações com o Congresso] *é muito mais embaixo*
'the hole [in negotiations with Congress] is much more down'

(9) *a justiça* [que o brasileiro tanto almeja] *começa dentro de casa*
'the justice [that the Brazilian so much craves] begins at home'

Another issue that had to be considered in the insertion window is the fluctuation of punctuation marks. In Portuguese proverbs, the use of comma is not systematic, and in many cases it can be considered to be optional. Particularly, in verse-like proverbs, with parallel metric in each hemistich, an hyphen '–' or even a slash '/' can be found. The reference graphs allow the facultative presence of punctuation between the core words of the proverb so that both forms are retrieved; *e.g.* 5–5:

(10) *Quem sai ao vento (,) perde o assento* (comma facultative)
'Who leaves to the the wind, loses the seat'

(11) *Quando a esmola é demais (,) o santo desconfia* (comma facultative)
'When the alms are too much, the saint suspects'

The lemmatization of the core words also raises several interesting issues. Many words were lemmatized aiming to identify all inflected forms of the verbs and the nouns, but for proverbs with the structure [*V Cop V*], such as *Recordar é viver* 'To remember is to live',

Amar é sofrer 'To love is to suffer', *Querer é poder* 'To want is to be able', among others, only the infinitive can be used, so we decided that the surface form should appear in the lexicon-grammar table.

Some proverbs admit transformations. For example, almost every proverb in class P1F2 allows the mirror permutation, which consists in reversing the order of constituents (subject and predicative) around the copula verb *ser* 'to be'; *e.g.* 5–5:

(12) *O ataque é a melhor defesa* [Mirror Perm.]= *A melhor defesa é o ataque*
'The attack is the best defense = The best defense is the attack'

(13) *A fome é o melhor tempero* [Mirror Perm.]= *O melhor tempero é a fome*
'Hunger is the best seasoning = The best seasoning is hunger'

The mirror permutation was only found in proverbs with a *NP* in the predicative position. In the case of adjectival structures, as in the proverbs *A carne é fraca* 'The flesh is weak', *O amor é cego* 'Love is blind' and *Errar é humano* 'To make mistake is human', this transformation is more rarely observed, though it can still be found in the web, so we extended it to the entire set of this class:

"*Quão fraca é a carne humana!*" [11];
"*O que você quis dizer com "Eu não sabia o quão cego é o amor."?*" [12];
"*Eu a amo, já relevei mtas coisas, mas humano é errar, burrice é repetir os erros. Cansei.*"[13]

Class P1F4 was distinguished from P1F2 and P1F3 because of the presence of an obligatory negation element, such as *não* 'not', *nunca* 'never', *jamais* 'never', among others. However, wordplay often involves the removal of this negation, to produce some type of effect. For example, on par with the proverb *Beleza não põe mesa* 'Beauty does not set the table', an affirmative variant 5 was found in the corpus :

(14) *Como a maioria das outras entrevistadas, Astrid diz que beleza põe mesa, sim*
'Like most other interviewees, Astrid says that beauty does set the table, yes'

Naturally, the interpretation of this sentence implies the previous knowledge of the negative form of the proverb. However, because of this creative re-use of the negative structure, the negation element was not considered an obligatory core element of the proverb.

Class P2F2 consists of 71 proverbs, formed by two coordinated propositions. Many of them result from the sum of two simple proverbs with one proposition each, *e.g.* the proverb 5 results from the combination of the proverbs 5 and 5, so it is considered a proverb and not just a variant.

(15) *Quem casa não pensa, quem pensa não casa*
'Who gets married doesn't think, who think doesn't get married'

(16) *Quem casa não pensa*
'Who gets married doesn't think'

(17) *Quem pensa não casa*
'Who think doesn't get married'

In these cases, in which a proverb is formed by two clauses, but also admits that only one of the clauses be used independently, the proverb was inserted thrice: in P2F1 class or in P2F2 (two clauses), and in P1F3 or P1F4 classes (single clause classes).

[11] `http://www.pastoralis.com.br/pastoralis/html/modules/newbb/` [2014/03/23]
[12] `http://m.fanfiction.com.br/reviews/historia/58620/capitulo/439083` [2014/03/23]
[13] `http://www.segredototal.com.br/de/homem/` [2014/03/23]

6 Final Remarks

In this paper we presented a methodology for detecting proverbs automatically in running texts. Proverbs have a similar syntactic structure and contain the same lexicon as ordinary free sentences, but they must be interpreted as a single unit of meaning. However, they often lack the presence of introductory expressions, that signal them as quotations, or are recast (and reshaped) in the ordinary stream of discourse, so it is necessary to recognize them in texts as multiword meaning units at a sentential/clausal level.

The results of this study showed contributions both for theoretical linguistics and to automatic text processing. As linguistic contributions, we emphasize:

(i) the formal (syntactic) classification of proverbs in 12 classes; this classification may serve as a starting point for deeper analysis on each one of these proverbial structures, as it has been done for the Spanish, French and Italian [2, 10, 11, 22];

(ii) the identification of the core elements of each proverb; the methodology presented to extract keywords can be replicated for other different *corpora* in order to see if the results are consistent across the different text types and domains;

(iii) the definition of an adequate extent of a window for insertions (words and punctuation), which may vary depending on the formal class; and

(iv) the frequent occurrence of variation, including of transformational nature, such as the mirror-permutation, and the zeroing of negation elements.

As contributions for automatic processing of texts in natural language, we highlight:

(i) the evaluation of the task, which showed 60.15% of precision with a 0-5 words window and 73.35% when no insertion is allowed; and

(ii) the construction and application of reference graphs for automatic detection of the proverbs and their variants in large corpus.

Naturally, much is still to be done.

Acknowledgements. This work was partially supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under project PEst-OE/EEI/LA0021/2013 and by Capes/PDSE under Process BEX 12751/13-8. We would like to thank the comments of the anonymous reviewers, which helped to improve this paper.

References

1 Jorge Baptista, Anabela Correia, and Graça Fernandes. Léxico-gramática das frases fixas do portugués europeo. *Cadernos de Fraseoloxía Galega*, pages 41–53, 2005.

2 María Lucía Navarro Brotons. *Las paremias y sus variantes: análisis sintáctico, semántico y traductológico español/francés*. PhD thesis, Universidad de Alicante, Alicante, Spain, 2008.

3 M. Bruckschein, F. Muniz, J. G. C. Souza, J. T. Fuchs, K. Infante, M. Muniz, P. N. Gonçalez, R. Vieira, and S. M. Aluisio. Anotação linguística em xml do corpus pln-br. Série de relatórios do nilc, NILC – ICMC – USP, 2008.

4 Lars Bungum, Björn Gambäck, André Lynum, and Erwin Marsi. Improving word translation disambiguation by capturing multiword expressions with dictionaries. In *Proceedings of the 9th Workshop on Multiword Expression*, pages 21–30, Atlanta, Georgia, USA, June 2013.

5 Helena M. Caseli, Carlos Ramisch, Maria das Graças Volpe Nunes, and Aline Villavicencio. Alignment-based extraction of multiword expressions. *Language Resources and Evaluation – Special Issue on Multiword expression: hard going or plain sailing.*, pages 59–77, 2010.

6 Lucília Chacoto. A sintaxe dos provérbios – as estruturas quem/quien en portugués e español. *Cadernos de Fraseoloxía Galega*, pages 31–53, 2007.

7 Lucília Chacoto. Mais vale mais um gosto na vida que três vinténs na algibeira – las estructuras comparativas en los proverbios portugueses. *Aspectos formales y discursivos de las expresiones fijas*, pages 87–103, 2008.

8 Mirella Conenna. Acerca del tratamiento informático de los proverbios. *Léxico y fraseología*, pages 197–204, 1998.

9 Mirella Conenna. Sur un lexique-grammaire comparé de proverbes – les expressions figées. *Langages*, 90:99–116, 1998.

10 Mirella Conenna. Classement et traitement automatique des proverbes français et italiens. *Lexique, Syntaxe et Sémantique, Mélanges offerts à Gaston Gross à l'occasion de son soixantième anniversaire*, pages 285–294, 2000.

11 Mirella Conenna. Dictionnaire électronique de proverbes français et italiens. In *Actes du XXIIe Congrès International de Linguistique et de Philologie Romanes*, pages 137–145, Bruxelles, Juillet 2000.

12 Mirella Conenna. Principes d'analyse automatique des proverbes. *Syntax, Lexis & Lexicon-Grammar, Papers in honour of Maurice Gross*, pages 91–103, 2004.

13 Paul Cook and Graeme Hirst. Automatically assessing whether a text is cliched, with applications to literary analysis. In Valia Kordoni, Carlos Ramisch, and Aline Villavicencio, editors, *Proceedings of the 9th Workshop on Multiword Expression*, pages 52–57, Atlanta, Georgia, USA, June 2013. Association for Computational Linguistics.

14 Márcia de Carvalho Saliba. Unidades lexicais maiores que a palavra: descrição linguística, considerações psicolinguísticas e implicações pedagógicas. Master's thesis, Universidade Federal do Paraná, Paraná, 2000.

15 Ana Clara Gonçalves Alves de Meira. Uma análise da articulação de cláusulas hipotáticas adverbiais em provérbios do portugês brasileiro. In EDUFU, editor, *Anais do SILEL*, volume 1, Uberlândia-UFMG, 2009.

16 Ana Clara Gonçalves Alves de Meira. A articulação de orações em provérbios do portugês em uso: uma análise das relações retóricas. Master's thesis, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, 2011.

17 Glaucy Ramos Figueiredo. *O gênero proverbial na imprensa: usos e funções retóricas*. PhD thesis, Universidade Federal de Pernambuco, Recife-PE, 2012.

18 Maurice Gross. Une classification des phrases figées du français. *Révue Québécoise de Linguistique*, 11(2):151–185, 1982.

19 Raimundo Magalhães Jr. *Dicionário brasileiro de provérbios, locuções e ditos curiosos: bem como de curiosidades verbais, frases feitas, ditos históricos e citações literarias, de curso corrente na língua falada e escrita*. Documentário, Rio de Janeiro, 3 ed edition, 1974.

20 Valia Kordoni, Carlos Ramisch, and Aline Villavicencio, editors. *Proceedings of the ACL Workshop on Multiword Expressions: from Parsing and Generation to the Real World (MWE 2011)*, Portland, OR, USA, June 2011.

21 Valia Kordoni, Carlos Ramisch, and Aline Villavicencio, editors. *Proceedings of the 9th Workshop on Multiword Expression*, Atlanta, Georgia, USA, June 2013.

22 Cláudia B. Lacavalla. *Lexique-grammaire des proverbes en Quand/Quando – Comparaison français-italien et représentation par grammaires locales*. PhD thesis, Universitá degli Studi di Bari, Bari, Itália, 2007.

23 Éric Laporte, Preslav Nakov, Carlos Ramisch, and Aline Villavicencio, editors. *Proceedings of the COLING Workshop on Multiword Expressions: from Theory to Applications (MWE 2010)*, Beijing, China, August 2010.

24 Ana Cristina Macário Lopes. *Texto Proverbial Português – Elementos para uma análise semântica e pragmática*. PhD thesis, Universidade de Coimbra, Coimbra, 1992.

25 Maria Lucia Mexias-Simon. Para uma estrutura dos provérbios nas línguas românicas: uma experiência. *Mosaico – Revista Multidisciplinar de Humanidades*, 2(2):59–74, 2011.

26 Martha Palmer. Complex predicates are multi-word expressions. In *Proceedings of the 9th Workshop on Multiword Expression*, page 31, Atlanta, Georgia, USA, June 2013.

27 Sébastien Paumier. *De la reconnaissance des formes linguistiques à l'analyse syntaxique.* PhD thesis, Université de Marne-la-Vallée, 2003.

28 Sébastien Paumier. *Unitex 3.1 – Manuel d'Utilisation*, last edition, 2013.

29 Ciça Alves Pinto. *Livro dos provérbios, ditados, ditos populares e anexins.* Senac, São Paulo, 4 ed edition, 2003.

30 Ivan A. Sag, Timothy Baldwin, Francis Bond, Ann Copestake, and Dan Flickinger. Multiword expressions: A pain in the neck for NLP. In *Proc. of the 3rd International Conference on Intelligent Text Processing and Computational Linguistics (CICLing-2002)*, pages 1–15, 2001.

31 Ana Paula Gonçalves Santos. Análise da escolha lexical no estudo dos provérbios em LP. In *Anais do SIELP*, Uberlândia-UFMG, 2012. EDUFU.

32 Martha Steinberg. *1001 provérbios em contraste: provérbios ingleses e brasileiros.* Editora Ática, São Paulo, 1985.

33 José Teixeira. Mecanismos metafóricos e mecanismos cognitivos: Provérbios e publicidade. In Arco Libros, editor, *Actas del VI Congreso de Lingüistica General*, pages 2271–2280, Madri, 2007.

34 Nelson Carlos Teixeira. *O grande livro de provérbios.* Leitura, Belo Horizonte, 1942.

35 Oto Araújo Vale. *Expressões cristalizadas do português do Brasil: uma proposta de tipologia.* PhD thesis, Universidade Estadual Julio Mesquita Filho – UNESP, 2001.

Language Identification: a Neural Network Approach

Alberto Simões[1], José João Almeida[2], and Simon D. Byers[3]

1 **Centro de Estudos Humanísticos, Universidade do Minho**
Braga, Portugal
ambs@ilch.uminho.pt

2 **Departamento de Informática, Universidade do Minho**
Braga, Portugal
jj@di.uminho.pt

3 **AT&T Labs**
Bedminster NJ, US
headers@gmail.com

Abstract

One of the first tasks when building a Natural Language application is the detection of the used language in order to adapt the system to that language. This task has been addressed several times. Nevertheless most of these attempts were performed a long time ago when the amount of computer data and the computational power were limited. In this article we analyze and explain the use of a neural network for language identification, where features can be extracted automatically, and therefore, easy to adapt to new languages. In our experiments we got some surprises, namely with the two Chinese variants, whose forced us for some language-dependent tweaking of the neural network. At the end, the network had a precision of 95%, only failing for the Portuguese language.

1998 ACM Subject Classification I.2.7 Natural Language Processing: Language models

Keywords and phrases language identification, neural networks, language models, trigrams

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.251

1 Introduction

The problem of Language Identification has been addressed for a long time, usually as a language model, that validates how likely a text is modeled by a specific language model [4]. This task can be considered the base when building a natural language processing stack of tools, as before one can apply mostly any kind of language processing tool there is the need to know the text language or, at least, the text alphabet. Only after that identification is done we can apply a tool to the text being certain that it will know how to deal with the characters, the words, or the syntax.

Following the idea presented in the previous paragraph, we can divide the task of identifying a language in two main tasks: first, the alphabet identification (looking to which characters[1] are used) and second, the identification of the language itself.[2]

[1] We are aware that the notion of character change with different alphabets. In this article we refer to character as an entry in the Unicode table.

[2] Here we are simplifying, as there are some languages that can be written in two different alphabets. In this paper we will consider that each language has a preferred alphabet, and that is the one that will be used.

© Alberto Simões, José João Almeida, and Simon D. Byers;
licensed under Creative Commons License CC-BY

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 251–265

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

At first the identification of some languages can be seen as simple. If a human looks to some Chinese text, he might notice that characters seem different from the ones used in Korean or Japanese text. In the same manner, Hindi characters are not likely to be found in other languages. The truth is that it is not as simple as it might seem, as for example, Chinese, Korean and Japanese share a huge amount of characters.

For the next level there are yet more problems. Consider the large amount of languages that share the Latin characters, and the amount of languages with the same origin, like Portuguese, Galician or Spanish. It can get harder if one tries to distinguish between language variants, like English from United States or United Kingdom.

In this paper we present a tool for learning language identification from tagged corpora into a Neural Network. Although this idea is not new, previous work on this task was published more than 20 years ago, and a lot has changed. Nowadays we have large quantities of text, in most any language existing in the world, and enough computational power to train a Neural Network is a relatively large amount of parameters. Also, and although we are not using that knowledge, there are new studies on methods to train Deep Neural Networks [5, 1], that we are interested to research about.

So, to start with, our main objective was to use a simple Neural Network implementation, making it easy to implement a language identifier in any programming language given the neural network parameters Θ. Then, as this approach is working, we intend to apply new techniques, namely the referred deep neural networks.

At the moment, and as a proof of concept we developed the learning algorithm in Octave (an open-source implementation of the well known MATLab software), and implemented language identifiers in two different programming languages: Perl and Java.

First we will analyze the current language identification approaches, namely the ones already using Neural Networks, and compare them with our approach. This will be discussed in the next section.

The section 3 describes the entire process of training the Neural Network, starting with the dataset preparation. Then, we will discuss how the features were chosen, and which were used. Follows the description of the Neural Network architecture, and its implementation details.

Section 4 will analyze how well the Neural Network performs in different kind of texts and for different language pairs.

Finally, section 5 presents some conclusions regarding our work, and pointers in future directions.

2 Language Identification Approaches

When looking up for works on language identification one might be surprised to find out that most of the recent published work is devoted to spoken language identification. Although the task might be similar, as the main algorithm would be to detect features most common in some languages than in another, the fact is that the searched features are of different kind. In this work we will focus only in the works devoted to identify languages on written text.

In the other hand, there are not many publications on language identification on written text. A reason for that might be the existence of two patents [9, 10], that explore the use of trigrams or generically n-grams for language identification. It is always interesting that some of these patents are granted when previous work like [6] already use this same kind of approach but for sound. Also, one year later, Nakagawa *et al.* [8] published work on language identification based on Hidden Markov Models that use n-grams as well.

In fact, Muthusamy already used neural networks for language identification. So, a question arises: what does our work do that was not done before? First, his main task was to identify language on spoken text. Other than that, in 1993 the amount of data available in text format was quite smaller than the amount of texts available nowadays, and the number of languages in which these texts exist is also very different. In another aspect, the computational power also changed drastically. Muthusamy used at most 10 languages, and not much more than 130 features, which were mostly chosen manually.

Our approach takes advantage of the amount of text available as well as the computational power to compute automatically what features to use. Our experiments gone up to more than one thousands features, having the network to took less than one day to train with a reasonable number of iterations.

3 Neural Network

Neural networks are being used for some time and their design and implementation for standard situations is well known [3]. Most of the work using neural networks aims at the classification of objects. In this case, the network works as a hypothesis function $h_\Theta(X)$ that, based on a set of matrices Θ previously computed on a training set, is able to classify an object based on a set of features extracted from that object. So, in the case of language identification, our training set is a set of texts manually classified in one language and an algorithm to extract features from them. These features are them fed in the neural network training algorithm that will compute the set of matrices Θ.

These matrices are then used to identify the language of new texts. For that, the features X are extracted from the text to be classified, and the hypothesis function is called. The resulting vector will include the probabilities of that text being identified as each one of the trained languages.

This section describes the main approach used to train our neural network. First we will discuss how the training texts were prepared. Follows the algorithm for extracting features from the training dataset, and the details on the neural network, explaining the architecture and the implementation details.

3.1 Dataset Corpora Preparation

In order to gather training data we used text from the TED conference website. This resulted in a core corpus of 105 different languages and language variants. These texts had very different sizes depending on the amount of data available in the TED website.

Given the technical nature of these texts, they include high proportion of technical terms, company and product names, and person names which are not translated. We will be referring to these as named entities [7], although some of them are not, at least in the usual definition of the term. This type of linguistic units is present to a varying degree in many language data sources.

This leads to the problem that text in a target language used for training might have snippets of another language appearing in it. This is exacerbated in translated text and in technical text. Also, in multilingual data on the same subject, particular word and character level features may appear in many languages despite being unrepresentative of most of them. When extracting n-grams, for example, it might happen that the most frequent are part of these terms. The result are features that are not language discriminant, although of high frequency.

In order to obtain clean training data we exploit the fact that the TED data form a multilingual parallel corpus. In particular the initial source language is English in this case. We extract the out-of-vocabulary words and some named entities from the English text using the *Hunspell*[3] spell checker and its default English dictionary. Note that we are extracting named entities that contain non-words, like proper names or trade marks. These words then, if they appear in the non-English tracks for that aligned text, should be removed due to their potential foreign origin.

This process allows us to obtain cleaner training text, where words are more likely to be purely of the tagged language. The drawback is that the resulting text no longer has correct sentences. Nevertheless, if we compute only character n-grams (and not word n-grams) that problem should not be relevant.

Finally, for the Portuguese language, we used the Lince [2] application in order to render the texts compliant to the 1990 Portuguese orthography reform, recently implemented. Given that this reform was established with the explicit goal of better unifying the orthography of the several variants of the Portuguese language worldwide, it is only natural that, in spite of the remaining differences, it has brought closer together the orthographies of European and Brazilian variants. Therefore, we might have considered Portuguese as an unique language and probably should have selected texts from only one of these variants. Nevertheless, we kept the two variants as distinct, and will discuss the obtained results later.

3.2 Feature Extraction

Our main goal, initially, was to use only n-gram features (namely trigrams) from the languages being used in the training process. Unfortunately, when using character trigrams, we are working with word trigrams for the Asiatic languages, like Korean, Chinese or Japanese, as each character represent (roughly) a word. This means that the amount of different trigrams for these languages is huge. To solve this problem we might enlarge the number of features extracted per language, thus making the training process prohibitive. Other option would be to change the number of trigrams for those specific languages. At the end we decided to create character dependent features (instead of some language-dependent features), regarding the number of characters used in some alphabets.

Therefore, currently we have two different levels of features: one related with the characters that are used, and another with the character trigram frequency information. All these features are extracted from 30 different texts for each one of the training languages.

Alphabet Features

As stated in the introduction, it is not possible to create an injective function from used characters to the written language neither from the language to the used characters.

For the Latin alphabet alone there are dozens of languages. For the Chinese, Japanese and Korean languages, they all use Chinese Kanji morphemic script, although Japanese script is sillabary, not an alphabet, and Korean uses a proper alphabet (phonologically based script). The situation gets worse when looking to the traditional and simplified Chinese versions that share most of their characters.

To compute features related to the used characters, we defined 10 different classes C_i:

[3] Details on the Hunspell spell checker and its dictionaries can be obtained from the project webpage at `http://hunspell.sourceforge.net/`

1. Latin characters, only `a-z`, without diacritics;
2. Cyrillic characters, containing Unicode characters in the intervals `0x0410-0x042F` and `0x0430-0x044F`;
3. Hiragana and Katakana characters (used for Japanese), containing Unicode characters between `0x3040-0x30FF`
4. The Hangul characters (used for Korean), from the Unicode classes `0xAC00-0xD7AF`, `0x1100-0x11FF`, `0x3130-0x318F`, `0xA960-0xA97F` and `0xD7B0-0xD7FF`;
5. Kanji characters (used in Japaneses, Korean and Chinese), from the Unicode class `0x4E00-0x9FAF`;
6. Simplified Chinese characters, a list of 2877 characters, hand-curated and available on GitHub[4];
7. Traditional Chinese characters, a list of 2663 characters, hand-curated and also availablefrom GitHub;
8. Arabic characters (used in Persian, Urdu, and different varieties of the Arabic language), in the Unicode class `0x0600-0x06FF`;
9. Thai characters, for the Unicode class `0x0E00-0x0E7F`;
10. Greek characters, in the Unicode classes `0x0370-0x03FF` and `0x1F00-0x1FFF`.

For the text segment being analyzed, the number of characters for each one of these classes are counted, and the relative frequency computed. After some experiments, and in order to reduce the entropy for the neural network, we decided to help by computing discrete values. Therefore, before using these ten values in the neural network a small set of rules make the values binary. When setting a class C_i, the result will have $C_i = 1$ and $C_j = 0, \forall j \neq i$.

Follows the list of rules used in this context:

$$
\begin{aligned}
set\ C_1 &\Leftarrow C_1 > 0.20 \\
set\ C_2 &\Leftarrow C_2 > 0.40 \\
set\ C_3 &\Leftarrow C_3 > 0.20 \\
set\ C_4 &\Leftarrow C_4 > 0.20 \\
set\ C_6 &\Leftarrow C_5 > 0.30 \wedge C_6 > C_7 \\
set\ C_7 &\Leftarrow C_5 > 0.30 \wedge C_6 < C_7 \\
set\ C_8 &\Leftarrow C_8 > 0.20 \\
set\ C_9 &\Leftarrow C_9 > 0.20 \\
set\ C_{10} &\Leftarrow C_{10} > 0.20
\end{aligned}
$$

These percentages were defined empirically. In fact, these rules are specially relevant for the Japanese, Korean and Chinese languages. Note that the two complicate rules are used to distinguish between the two Chinese variants. After running these rules, these features are used directly in the neural network.

Trigram Features

Regarding language information, we chose to store information about character trigrams. There are different reasons why we chose to use three characters:

[4] Check `https://github.com/jpatokal/script_detector`

```
Für mich war das eine neue Erkenntnis. Und ich denke, mit der Zeit, in den
kommenden Jahren, Wir haben Künstler, aber leider haben wir sie noch nicht
entdeckt. Der visuelle Ausdruck ist nur eine Form kultureller Integration.
Wir haben erkannt, dass seit kurzem immer mehr Leute
```

Figure 1 A sample text in the German language.

- bigrams would be too small when comparing very close languages like Portuguese and Spanish;
- tetragrams would be too big for Asiatic languages, where some gliphs represent words or morphemes;
- punctuation and numbers were removed, and spaces normalized, meaning that trigrams would be able to capture the end and beginning of two words that usually occur together, as well as to capture single character words that appear surrounded by spaces.

This task was performed using the Perl module `Text::Ngram`[5], which deals with the task of cleaning the text, normalizing spaces and computing n-grams. The obtained counts were then divided by the total number of trigrams found, thus computing their relative frequency.

As an example, Table 1 shows the result of computing trigrams on the text from Figure 1.

Table 1 Top 25 occurring trigrams from text shown in Figure 1.

en␣	0.02299	er␣	0.02682	␣de	0.01533	abe	0.01533	der	0.01149
hab	0.01149	ich	0.01149	ir␣	0.01149	it␣	0.01149	r␣h	0.01149
␣wi	0.01149	ben	0.01149	ch␣	0.01149	den	0.01149	wir	0.01149
␣ha	0.01149	ine	0.00766	ler	0.00766	lle	0.00766	n␣k	0.00766
mme	0.00766	ne␣	0.00766	nnt	0.00766	r␣l	0.00766	r␣m	0.00766

Features Merging

Although the alphabet features is a limited list of ten different alphabets, there is the need to merge the trigram features into just one list choosing only the more significant.

This process is performed in two stages, first for each language, then for the entire training set:

1. For each of the 30 training texts from a specific language we compute the 20 trigrams with higher frequency. The trigrams are then merged in an unique list that includes the most occurring trigrams from all the training texts in a specific language. Next, this list is reduced, preserving only the 20 trigrams that are present in most texts. Note that we are not interested in their frequency in each training text, but how often they appear in different texts.
2. Next, each group of 20 trigrams computed from a specific language are joined together in a big list of features.

So, the complete features list F includes the alphabet features (F_a) and the trigrams features (F_t): $F = F_a \cup F_t$. With this feature list we can compute the training data, in the form of a matrix. Each line of the matrix is the data collected from each one of the training

[5] Available from `https://metacpan.org/pod/Text::Ngram`.

Table 2 Training data matrix.

	Alphabet Features			Trigram Features						
	Latin	Greek	Cyril.	␣pa	ói␣	par	nia	ест	ати	ата
PT	1	0	0	0.0041	0	0.0038	0.0001	0	0	0
PT	1	0	0	0.0039	0	0.0036	0	0	0	0
RU	0	0	1	0	0	0	0	0.0020	0.0004	0.0003
RU	0	0	1	0	0	0	0	0.0026	0.0005	0.0002
UK	0	0	1	0	0	0	0	0.0003	0.0034	0.0001
UK	0	0	1	0	0	0	0	0.0003	0.0026	0.0001
VI	1	0	0	0	0.0028	0	0	0	0	0
VI	1	0	0	0	0.0029	0	0.0001	0	0	0

texts. Each column of the matrix corresponds to a different feature from F. Each cell of the matrix stores the value of a specific feature in a specific training text. Table 2 shows an excerpt from this matrix.

3.3 Network Architecture

A neural network is composed by a set of L layers, each one composed by a set or processing units. A processing unit is denoted by $a_i^{(l)}$ where l is the layer where it belongs, and i its order.

All units from a specific layer are connected to all units from the next layer. This connection is controlled by a matrix $\Theta^{(l)}$, for each layer l.

The first layer is known as the *input layer*. It has the same number of units as there are features to be analyzed (in our experiment, 565 units). Whenever the network hypothesis function is evaluated each cell $a_i^{(1)}$ is filled in with the values obtained by the features observation.

The next layer, $a_i^{(2)}$ is computed using the previous layer and the matrix $\Theta^{(1)}$, as will be explained in the next section. This process is done for every layer $l \leq L$.

The layer L is known as the *output layer*. There are as many units in this layer as the number of classes K in which the network will classify objects. Therefore, if the network is trained to detect 25 languages, then there are 25 units in the output layer. Each unit in the output layer will, optimally, get a value that is either 1 or 0, meaning that the object is, or is not, in the respective class. Usually, the result is a value in this range, that represent the probability of the object to be of that specific class.

The other layers, $1 < l < L$, are known as the *hidden layers*. There are as many hidden layers as one might want, but there is at least one hidden layer. Adding new layers will make the network return better results but it will take more time to train the network, and take more time to run the network hypothesis function. For our experiments we used only one hidden layer.

Regarding the number of units in the hidden layers, there are some rules of thumb: use the same number of units in all hidden layers, and use at least the same number of units as the maximum between the number of classes and the number of features. But there can be up to three times that value. Given the high number of features we opted to keep that same number of units in the hidden layer.

3.4 Training Details

This kind of neural network implementation is not complicated, but is susceptible to errors. Our neural network was implemented using the more common definition of a neural network [3].

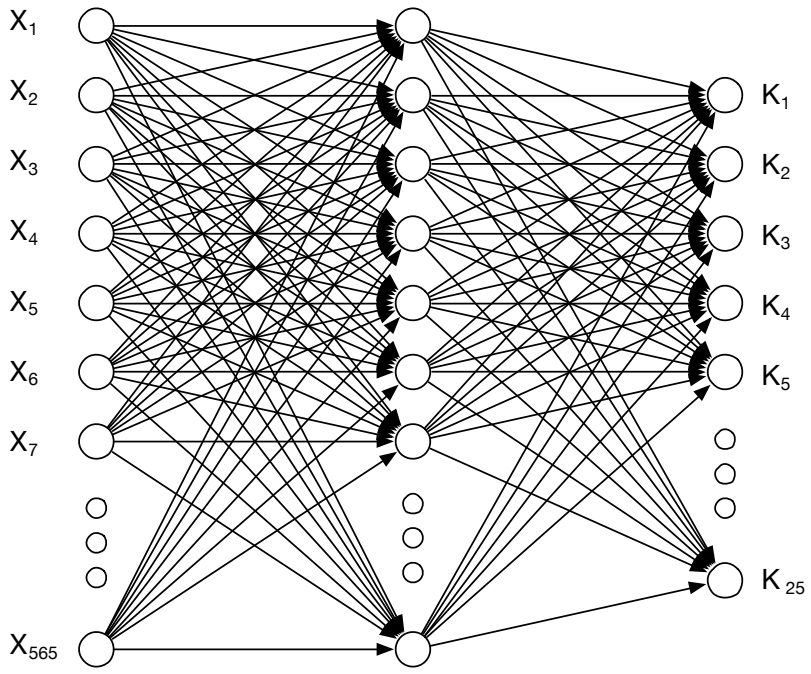


Figure 2 Neural network architecture.

The implementation of the neural network was based on the logistic function defined by $g(z)$. This function range is $[0, 1]$, and its result value can be considered a probability measure. The logistic function is defined as:

$$g(z) = \frac{1}{1 + \exp -z}$$

Our neural network hypothesis function, $h_{\Theta^{(l)}}(X)$ is defined by two matrices, $\Theta^{(1)}$ and $\Theta^{(2)}$. These matrices of weights are used to compute the network. The input values, obtained by the computed features, are stored in the vector X. This vector is multiplied by the first weight matrix, and the logistic function is applied to each value of the resulting vector. The resulting vector is denoted as $a^{(2)}$ and corresponds to the values of the second layer of the network (the hidden layer). It is then possible to multiply $a^{(2)}$ vector by the weights of $\Theta^{(2)}$ and, after applying the sigmoid function to each element of the resulting multiplication, we obtain $a^{(3)}$. This is the output layer, and each value of this vector corresponds to the probability of the document being analyzed to as being written in a specific language. This algorithm is known by *forward propagation* and is defined by:

$$\begin{aligned} &a^{(1)} = x \\ &\text{for } i = 2 \text{ to } L, \\ &\quad a^{(i)} = g\left(\Theta^{(i-1)} x\right) \end{aligned}$$

The main problem behind this implementation is how to obtain the weight values. For that the usual methodology is to define a cost function and try to minimize it, that is, finding the Θ values for which the hypothesis function has a smaller error for the training set.

The cost function with regularization is defined as:[6]

$$J(\Theta) = -\frac{1}{m}\left(\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log(h_\Theta(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_\Theta(x^{(i)}))_k)\right) + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{S_l+1}\left(\Theta_{j,i}^{(l)}\right)^2 .$$

The regularization is controlled by the coefficient λ which can be used to tweak how the Θ weights absolute value will increase. Although our implementation supports regularization the experiments performed did not use any regularization ($\lambda = 0$).

The minimization of the cost function $J(\Theta)$ is computed by an algorithm known as *Gradient Descent*. This algorithm uses the partial derivatives

$$\frac{\partial}{\partial \theta_{i,j}^{(l)}} J(\Theta)$$

to compute the direction to use to obtain the function minimum. The algorithm continues iterating until the difference between the obtained costs is very small, or until a limit number of iterations it met.

Gradient Descent can be implemented using an algorithm known as *Backwards Propagation* to compute efficiently the partial derivatives. Our implementation runs a number of iterations and save the Θ values. It is then possible to continue the training from those values. In the future this will allow us to create a test set and stop training when it has a sufficiently high precision. Nevertheless, at the moment we are performing tests with a fixed number of iterations (check next section).

4 System Evaluation

Our experiment used 25 languages: Arabic (AR), Bulgarian (BG), German (DE), Modern Greek (EL), Spanish (ES), Persian (FA), French (FR), Hebrew (HE), Hungarian (HU), Italian (IT), Japanese (JA), Korean (KO), Dutch (NL), Polish (PL), Portuguese (PT), Brazilian Portuguese (PT-BR), Romanian (RO), Russian (RU), Serbian (SR), Thai (TH), Turkish (TR), Ukrainian (UK), Vietnamese (VI) and, Traditional and Simplified Chinese (ZH-TW and ZH-CN).

The neural network was trained using these 25 languages and the corpora described in section 3.1. The next subsection explains the creation and characterizes the test set for these languages. Note that although the training corpora was cleaned, removing some words that are not likely to be in that language, the test corpora is noisy (namely including some words from other languages).

4.1 Test Set Characterization

For each language to be identified we collected 21 documents. Given we do not master all these languages we had some difficulties on collecting documents for some languages. To be sure of the languages of the test files we often resorted to other language identification software. All the texts were collected from on-line newspapers. Therefore, the texts have

[6] It goes beyond of focus of this article to discuss and explain what is the regularization and how it works. The same is true regarding the Gradient Descent or the Backwards Propagation algorithms.

Table 3 Training and test set statistic for each language. Values are in number of Unicode characters.

	Training Set				Test Set			
Language	Smaller	Larger	$\bar{x}$	σ	Smaller	Larger	$\bar{x}$	σ
AR	871921	969387	907562	21392	863	4618	2366	1210
BG	988450	1087435	1027581	23663	660	2099	1091	378
DE	588200	653508	618463	16475	677	3890	1554	842
EL	773265	885770	841203	22653	550	3297	1590	705
ES	578806	651240	617341	17637	897	3850	2342	935
FA	651807	766206	697212	28994	600	5221	1338	967
FR	639582	705675	673414	15377	936	4088	1879	689
HE	806098	877218	836222	20545	559	3649	1586	878
HU	406271	454506	431797	13131	729	6045	2175	1356
IT	588147	643252	616391	14348	1260	6607	2991	1370
JA	538033	606053	569956	18871	323	785	495	133
KO	737118	817651	773168	20550	530	1603	780	233
NL	533497	580313	557724	14033	552	1949	1115	381
PL	521184	591299	551259	17938	435	3092	1605	694
PT-BR	596158	643215	617734	14028	920	3189	1953	589
PT	338272	378872	355800	10605	486	5875	2031	1169
RO	592714	650375	616051	15442	718	3254	1438	695
RU	1019789	1144200	1069884	31232	662	2470	1444	526
SR	349389	433221	379344	20560	834	6493	1813	1263
TH	529484	601244	565082	18551	334	3242	1396	734
TR	494191	549998	524271	12774	332	5390	1559	1121
UK	370785	434683	395312	16641	299	15435	2430	3553
VI	470057	541930	510409	17246	680	6237	1555	1359
ZH-CN	536438	595027	562728	14457	495	6331	1695	1559
ZH-TW	514993	588860	542879	16000	270	1721	925	428

plenty of named entities (that our training corpus misses) and vary on size. In fact, in some situations the news texts were not copied completely, in order to have smaller texts. Unfortunately the task of collecting these texts was done ad-hoc, resulting in some very different sizes for different languages. Check Table 3 for some more information on the number of characters per test file.

Curiously, when building this test set we found some texts that were being wrongly identified because we collected them in the wrong language. Although this fact is not relevant, it was curious that a collected text in Catalan was identified as French. This means that the neural network is able to detect languages by proximity.

4.2 Accuracy

Our first experiments did not include the alphabet features. Although it worked relatively well for most languages, the trained neural network failed for the four Asiatic languages. The main reason for that is the large proportion of characters that are shared among these languages, while each one has a structurally different type of base script. This leads to a large amount of different trigrams and therefore the neural network would need many more features per language (or for these specific language).

Table 4 Accuracy on test set, when training with 1500 and 4000 iterations.

Language	1500 iters.	4000 iters.	Comments
AR	100%	100%	
BG	100%	100%	
DE	100%	100%	
EL	100%	100%	
ES	100%	100%	
FA	100%	100%	
FR	100%	100%	
HE	100%	100%	
HU	100%	100%	
IT	100%	100%	
JA	100%	100%	
KO	100%	100%	
NL	100%	100%	
PL	100%	100%	
PT	**5%**	**52%**	wrongly classifies as PT-BR
PT-BR	100%	**76%**	wrongly classifies as PT
RO	100%	100%	
RU	100%	100%	
SR	100%	100%	
TH	100%	100%	
TR	100%	100%	
UK	100%	100%	
VI	100%	100%	
ZH-CN	100%	100%	
ZH-TW	100%	100%	

After adding the alphabet features, we trained the neural network with two different number of iterations: 1500, and 4000. Table 4 presents accuracy values for each language when analyzing the test set. Globally, with 1500 iterations we were able to get 96% of precision, and with 4000 iterations it gets up to 97%.

Looking to the results' table one can see that the most problematic languages are the two Portuguese variants, for which many texts are being attributed to the Brazilian variant. This is probably the result from the 1990 orthographic reform, whose aim was, precisely, an orthographic unification of the Portuguese language across its variants, just like the tests demonstrate.

In order to compare our (bad) results we did some experiments with the Perl module `Lingua::Identify::Blacklists` [11] that uses lists of words that are blacklisted for some languages. The results for the Portuguese variants were 66% of accuracy for the European variant, and 100% accuracy for the Brazilian language.

Looking to this module blacklists we found out that, more than identifying the variant, the tool identifies the *topic* of the text. For example, the module states that if a text includes the word "Brasília" (the capital of Brazil) or "Pará" (a state from Brazil), then the text should be in the Brazilian variant. It happens in the inverse direction as well, with other proper names, like "Madaíl" (a controversial person in the Portuguese soccer) or "Louçã" (a left-wing deputy from the Portuguese parliament). Also, the module uses a list of words that changed in the 1990 orthographic agreement, meaning that for new Portuguese texts they are useless.

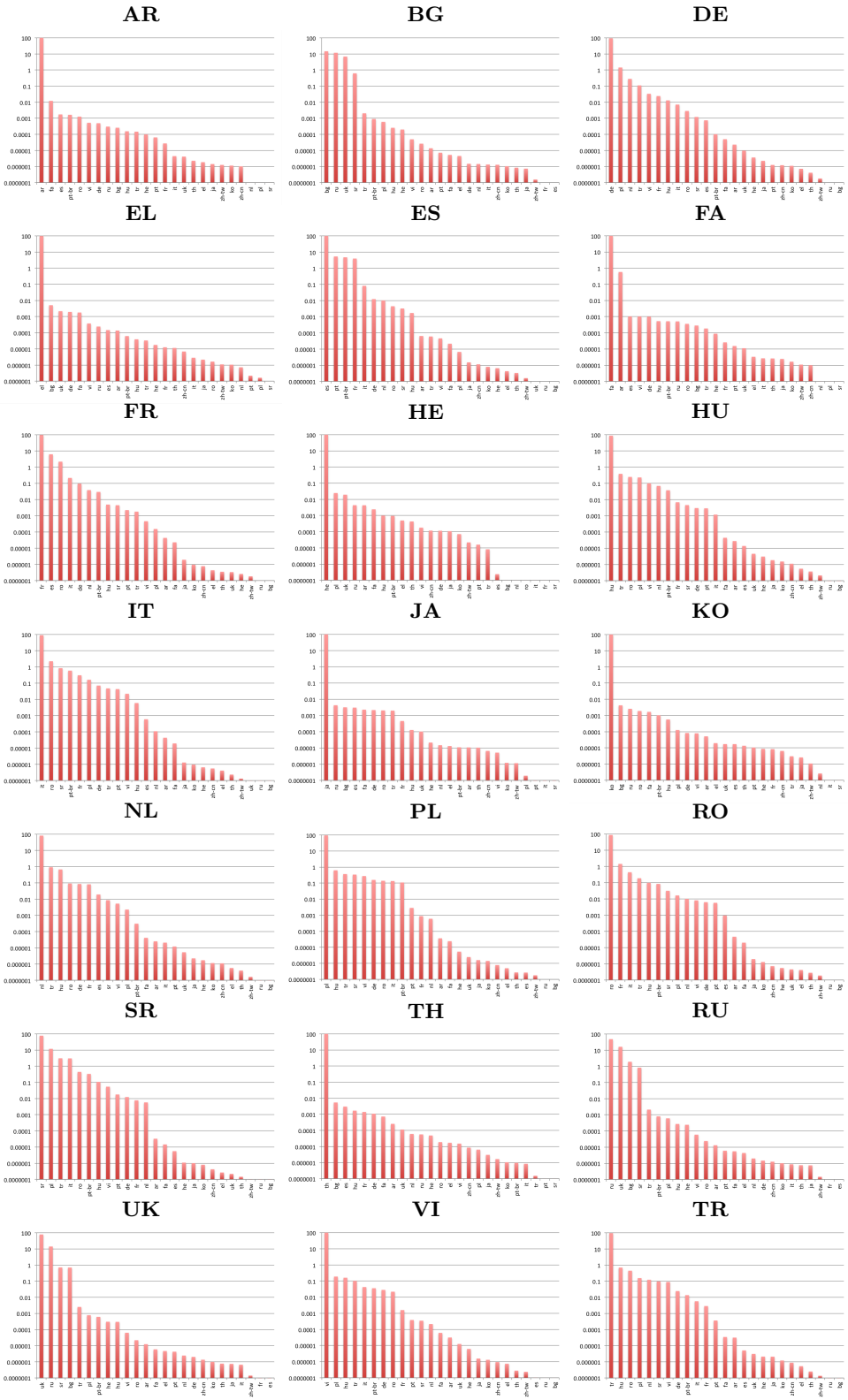


Figure 3 Language identification distribution.

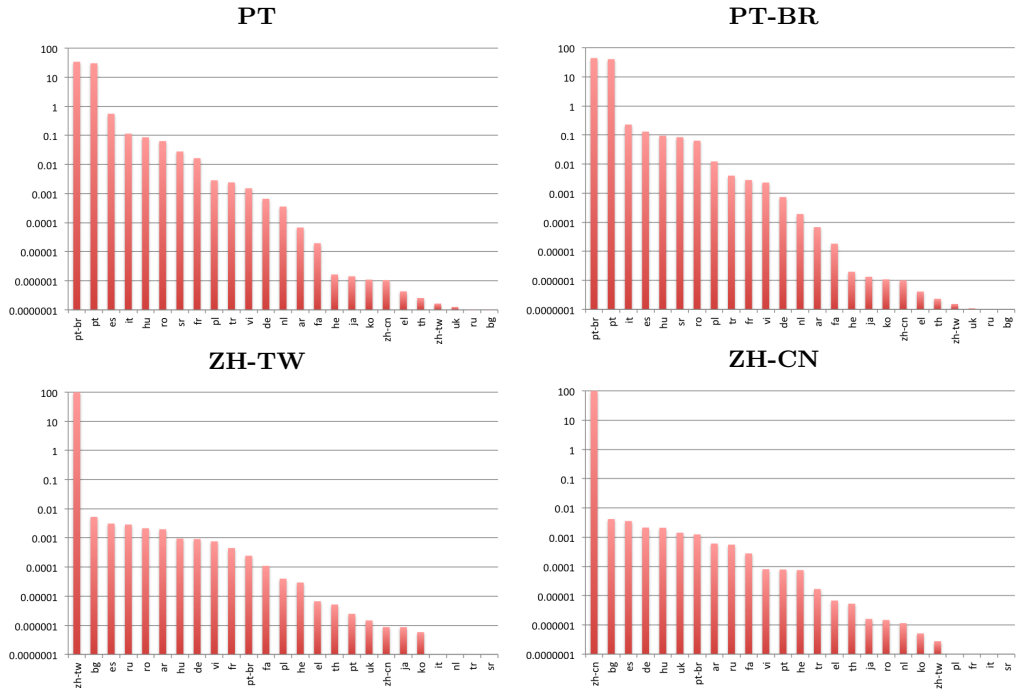


Figure 4 Language identification distribution for the two Portuguese and Chinese variants.

A good way to evaluate and compare the results from this module and our neural network would be the use of a good parallel corpus European/Brazilian Portuguese. This would allow us to evaluate the language identification and not the *topic* identification.

4.3 Probability Distribution

For each language we chose randomly one of the test files, and computed the language identification probabilities. Figure 3 show them for most languages.[7] Although the graphs are small and not readable, it is easy to notice that there is a big difference from the first language identified (the correct one) and the second choice. From these twenty one graphs the only relevant for analysis is the Bulgarian, which is very near Russian and Ukrainian.

Figure 4 presents the same graph for the remaining four languages, that include the two Portuguese and the two Chinese variants. Note that, for the Chinese variants, the difference from the first probability to the rest is very high. This is not a result of the trigrams features, but the fact that our alphabet identifier is working well to differentiate the two orthographies. Regarding the two Portuguese variants, it is clear the confusion between the European and Brazilian variants, with probabilities around 45%.

5 Conclusions and Future Work

In this article we present a neural network that is able to identify languages with 96% or 97% of accuracy, depending on the number of iterations performed during the training process.

[7] Note that graphs are using an exponential y axis.

For that we used two kind of features: one related with the language alphabet, and another related to the character trigrams with higher occurrence.

Given that we are able to use binary features to classify the alphabet (at the moment we have ten binary features) and they are mutually exclusive, the neural network is able to learn much faster to distinguish some collections of languages.

A problem with our approach is that it will perform badly on short snippets of text (like instant messages or mobile messages), because of the low number of trigrams selected by language. We are investigating how to deal with this problem without compromising the time needed to train the neural network.

Regarding the problem with the Portuguese variant we are mostly convinced to merge the two variants in a single one, given that with the so mentioned Orthographic Agreement it does not make sense to keep distinguishing between the two.

On using a neural network, we should be reminded that the result is not deterministic: the same number of iterations to train a network might yield different results, depending on the values used to initialize the Θ matrices.

Future Work

The next (certain) steps on this project would be (and probably, in this order):

1. Remove the Brazilian Portuguese and/or merge it with the European Portuguese variant;
2. Add the English language, that was not included at first because of some technical problems when preparing the training corpora;
3. Release the Perl and Java identification modules publicly;
4. Add more languages;
5. Go to point 3, and iterate.

Nevertheless, every time we train the neural network we find new experiments we would like to perform. These steps are likely to be done, but in any order:

- Try to reduce the number of trigrams per language and add some bigrams or one-grams. These tests' main rationale would be to reduce the number of features, as adding new languages are likely to include more features and make the training process slower.
- Compute distribution differences between near languages and, instead of using just the more occurring trigrams, use those that are most distinctive;
- In order to make the neural network smaller, train a different neural network for each alphabet. This will allow modularization when making the language identifier available. The user could then download only the modules relevant for her task.
- Our experiments with more than 4000 iterations gave worst results than the ones presented here. This happens because the algorithm is not using any regularization, and therefore the neural network is being biased by the training data and is unable to generalize. Further experiments are needed to study good values for the regularization coefficient.
- Neural networks are known to have difficulties to scale. Nevertheless, recent work in deep learning [5, 1], and deep neural networks might be relevant to analyze and use.

Acknowledgments. The authors would like to thank Catarina Sousa for the help compiling the test dataset, and the three reviewers, Lluís Padró, António Teixeira and Jorge Baptista, for their comments, insights and corrections.

References

1 Yoshua Bengio. Learning Deep Architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1):1–127, January 2009.

2 José Pedro Ferreira, António Lourinho, and Margarita Correia. Lince, an end user tool for the implementation of the spelling reform of Portuguese. In Helena de Medeiros Caseli, Aline Villavicencio, António J. S. Teixeira, and Fernando Perdigão, editors, *Computational Processing of the Portuguese Language - 10th International Conference (PROPOR)*, volume 7243 of *Lecture Notes in Computer Science*, pages 46–55. Springer, 2012.

3 Simon Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall PTR, 1994.

4 Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*. Prentice-Hall, second edition edition, 2009.

5 Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *J. Mach. Learn. Res.*, 10:1–40, June 2009.

6 Yeshwant Kumar Muthusamy. *A Segmental Approach to Automatic Language Identification*. PhD thesis, B. Tech., Jawaharlal Nehru Technological University, Hyderabad, India, October 1993.

7 David Nadeau and Satoshi Sekine. A survey of named entity recognition and classification. *Lingvisticae Investigationes*, 30(1):3–26, 2007.

8 Seiichi Nakagawa and Allan A. Reyes. An evaluation of language identification methods based on HHMs. *Studia Phonologica*, 28:24–26, 1994.

9 John C. Schmitt. Trigram-based method of language identification. US Patent Number 5.062.143, February 1990.

10 Bruno M. Schulze. Automatic language identification using both n-grams and word information. US Patent Number 6.167.369, December 1998.

11 Jörg Tiedemann and Nikola Ljubešić. Efficient discrimination between closely related languages. In *Proceedings of COLING 2012*, pages 2619–2634, Mumbai, India, December 2012. The COLING 2012 Organizing Committee.

LemPORT: a High-Accuracy Cross-Platform Lemmatizer for Portuguese

Ricardo Rodrigues, Hugo Gonçalo Oliveira, and Paulo Gomes

Centre for Informatics and Systems of the University of Coimbra
Pinhal de Marrocos, Coimbra, Portugal
{rmanuel,hroliv,pgomes}@dei.uc.pt

Abstract

Although lemmatization is a very common subtask in many natural language processing tasks, there is a lack of available true cross-platform lemmatization tools specifically targeted for Portuguese, namely for integration in projects developed in Java. To address this issue, we have developed a lemmatizer, initially just for our own use, but which we have decided to make publicly available. The lemmatizer, presented in this document, yields an overall accuracy over 98% when compared against a manually revised corpus.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases lemmatization, normalization, rules, lexicon

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.267

1 Introduction

Almost every task related to natural language processing (NLP) [10] must apply some kind of text normalization. Texts must be split into sentences, and sentences into tokens (words and punctuation). Words must also be further processed in order to facilitate their analysis: for instance, when searching for specific words, as it happens in many information retrieval systems (IR) [2], their inflections must also be considered in order to broaden the results.

The most common approaches for tackling this problem and collapsing morphological variants of the same word are: (i) stemming, which essentially consists of stripping off word endings; and (ii) lemmatization, where words with the same morphological root are identified, despite their surface differences.

Stemming is easier and faster to implement, but discards potentially useful information, by making it virtually impossible to distinguish a verb from a noun or an adjective in its stemmed form. Moreover, the stem is not necessarily a recognizable dictionary word.

Lemmatization, in contrast, considers the syntactic category of words, presenting, for instance, different lemmas for a noun or a verb (in the same word family). This nuance is practically lost in English, where the same lemma can assume multiple syntactic categories, but it is of paramount importance in romance languages, including Portuguese.

In the remaining sections of this document, we make a brief contextualization on lemmatization and related work, then proceed to describe our method, followed by the evaluation performed and results obtained, after what we end up drawing some conclusions and pointing future paths for eventual improvement.

2 A Brief Contextualization

In some situations, lemmatization and stemming operate in a similar way: given a set of affixes, for each word in a list (a phrase, a sentence or a text), check if the word ends with

© Ricardo Rodrigues, Hugo Gonçalo Oliveira, and Paulo Gomes;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 267–274

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

any of the affixes, and, if so, and apart from a few exceptions, remove the affix from the word. The problem is that this process is sometimes not enough to retrieve the dictionary form of a word, it is too disruptive and, in most cases, the stem is not the same as the lemma. Next, we point out how lemmatization should perform, and discuss related work.

2.1 The Lemmatization Process

In order to retrieve the lemma of a word, it is sometimes enough to remove the word's affix. This typically happens in *noun number normalization*, which is the case of `carros` losing the trailing `s` and becoming `carro` (`cars` → `car`). In other cases, such as in *noun gender normalization*, it is necessary to replace the affix. For instance, the dictionary form of `gata` (a female cat) is `gato` (a male cat), which requires replacing the feminine affix, `a`, for the masculine affix, `o`. The same applies to *verb normalization*: for instance, the lemmatized version of `[eu] estude`*`i`* (`[I] studi`*`ed`*) becomes `[eu] estud`*`ar`* (`[to] study`), replacing the verbal inflection affix with the associated conjugation infinitive affix (`ar`, `er`, or `ir`).

For Portuguese, lemmatization may include the following types of normalization: noun (gender, number, augmentative and diminutive), adjective (gender, number, augmentative, diminutive, and superlative), article (gender and number), pronoun (gender and number), proposition (gender and number), adverb (manner) and verb (regular and irregular). Proper nouns, numbers, interjections and conjunctions are usually ignored.

Determining what kind of normalization to apply depends on the syntactic category, or part-of-speech (POS) tag, of each word. The task of identifying the POS tag of a word is out of our scope, and we resort to a freely available solution: the OpenNLP Library[1], by the Apache Software Foundation, with some minor tweaks. So, a lemmatizer must take as input both word and POS tag to produce the coveted lemma.

In theory, knowing the syntactic category of a word and the rules to normalize it, lemmatization should be a straightforward process. In practice, although these rules cover the vast majority of the cases in any given text, exceptions to these same rules defeat the goal of reaching an accuracy close to 100%. Moreover, the exceptions usually happen to be found in the oldest and most used lexemes of any lexicon. For instance, the verbs *to be* and *to have* are highly irregular in every western language, including Portuguese and English.

The same happens in other syntactic categories, such as nouns. For instance, the singular form of `capitães` (`captains`) is `capitão` (and not `capitãe*`). There are also cases where the masculine form of a noun is quite different from the feminine version. And the most problematic case is perhaps when a word is already in its dictionary form but appears to be in an inflected form, when, in reality, it is not – as happens with `farinha` (`flour`), that seems to be a diminutive (due to its ending in `inha`, the most common diminutive affix).

Most lemmatization tools use a rule system (covering the vast majority of cases for each type of normalization), and specify exceptions to the rules, using, at some point, a lexicon for validating the lemmas produced, or for extracting rules (and their exceptions).

2.2 Related Work

Even though most researchers on Portuguese NLP must use some sort of lemmatization tool, those tools are hard to come by, at least as isolated lemmatizers, although there are suites of tools, including morphological analyzers, that produce lemmas as a part of the

[1] The OpenNLP toolkit can be found at `https://opennlp.apache.org`.

outcome. While useful, these suites and morphological analyzers leave to the users the task of post-processing the output in order to extract just the intended lemmas, imposing also some restrictions on previous processing, as it usually has to be done using the same tools.

Three such tools are known to exist, targeting specifically Portuguese: jSpell,[2] FreeLing,[3] and LX-Suite.[4] All have web interfaces, with jSpell and Freeling providing downloadable versions and source code. Moreover, jSpell is available as C and Perl libraries, as well as a MS Windows binary; and Freeling is available as a Debian package and also as a MS Windows binary, in addition to an API in Java and another in Python, along with the native C++ API.

Both jSpell and Freeling start with a collection of lemmas and use rules to create (all) inflections and derivations from those lemmas, alongside data such as number, syntactic category, or person (in the case of verbs) [12, 5]. It is the output of that process that is used to lemmatize words, matching them against the produced inflections and derivations, retrieving the originating lemma.

Regarding LX-Suite, though only a description of a nominal lemmatizer [3] has been found, which is believed to be the lemmatizer behind the LX-Suite [4] of NLP tools (belonging to the LX-Center), it uses a lexicon for the purpose of retrieving exceptions to the lemmatization rules: if the lexicon contains a word (therefore, a valid word) that would be processed by the rules (but should not), it is marked as an exception and added to the exception list. According to the authors, that lemmatizer achieved an accuracy of 97.87%, when tested against a hand annotated corpus with 260,000 tokens.

For jSpell and Freeling, although evaluations for the morphological analyzers do exist, no statement regarding specifically the accuracy of the respective lemmatizers was found.

3 Our Approach to Lemmatization

Our lemmatization method shares features with other approaches, such as the use of rules and, more recently, a lexicon. The way these resources are combined leads to a high accuracy value. In this section, we describe our method and its evolution.

3.1 The Use of Rules

In its earlier versions, our lemmatizer depended only on handmade rules and exceptions. Each of the normalization steps included in the lemmatization process had an associated set of rules. As such, there were rules for (and in this order):

1. manner (adverb) normalization;
2. number normalization;
3. superlative normalization;
4. augmentative normalization;
5. diminutive normalization;
6. gender normalization;
7. verb normalization (for regular and irregular verbs).

Each of these rules were associated to one or more POS tags. So, for instance, gender normalization could be applied to nouns and also to adjectives, while superlative normalization would only be applied to adjectives. The rules were defined by the target affix, the POS

[2] jSpell is freely available from `http://natura.di.uminho.pt/wiki/doku.php?id=ferramentas:jspell`.
[3] Freeling is freely available from `http://nlp.lsi.upc.edu/freeling/`.
[4] The LX-Suite can be found at `http://lxcenter.di.fc.ul.pt/services/en/LXServicesSuite.html`.

```
<replacement target="inha" tag="n|n-adj|adj" →
exceptions="azinha|...|farinha|...|sardinha|...|vizinha">a</replacement>
```

Figure 1 A rule for transforming a diminutive into its "normal" form.

```
<prefix>(a|ab|abs|...|sub|super|supra|...|vis).?\-?</prefix>
<suffix>[\wàáãâéêíóõôúç\-]*</sufix>
...
<replacement target="a[gj][aeiío]" tag="v|v-fin|v-ger|v-pcp|v-inf"> →
agir</replacement>
```

Figure 2 A rule for transforming a inflected verb into its infinitive form.

tags of the words they should be applied to, exceptions, and the replacement for the target affix. All rules were declared in XML files, illustrated by the example in Fig. 1. That specific rule would transform `malinha` (`little briefcase`) into `mala`, by replacing the affix `inha` for the affix `a`, but would leave `farinha` untouched, as it is one of that rule's exceptions.

Although the rules for all types of normalization shared the same general structure, the rules regarding verbs were somewhat specific. (i) The rules for lemmatizing irregular verbs consisted of all the possible conjugations of the Portuguese irregular verbs. It was simpler to do this than to come up with rules that could address all existing variations. (ii) The rules for the regular verbs used as target the stem of each verb, always ending in a consonant, followed by the only vocals that could be appended to that specific stem (depending on the conjugation the verb belongs to), ending with any sequence of letters. Small variations that could occur – for instance, the substitution of a `g` for a `j` in the verb `agir` (`to act`) in some of its inflections – were also considered. (iii) When the two previous types of rules failed to be applied, a set of rules with verbal inflection affixes was used.

For this to be possible, regular expressions were used. Also, any of these rules could accept a list of prefixes, to broaden the list of addressed verbs. An example of rules for regular verbs is shown in Fig. 2, where it is also shown the list of prefixes that can be added to a verb, and the ending (suffix) of all the verb rules.

Eventually, in the selection of the rule that would be applied to a word in a given step, beyond its target and syntactic category, when more than one rule was eligible, the lengthier one was chosen. The length of a rule was computed by a weighted sum of the number of characters in the target, the exceptions and the POS tags, from a higher to a lower weight.

It is worth noticing that the rules are easily readable and customizable. Moreover, it is possible to select which kind of normalization steps should be performed, by specifying flags on the calls to the lemmatizer – when none is specified, it defaults to apply all the normalization steps. Both of these features make our lemmatizer flexible and easy to adapt to different situations and purposes.

3.2 The Addition of a Lexicon

The current version builds up on the previous (using rules), with the addition of a lexicon, namely the "LABEL-LEX-sw" lexicon,[5] version 4.1, produced by LabEL [7]. The specified lexicon contains over 1,500,000 inflected forms, automatically generated from about 120,000 lemmas, characterized by morphological and categorical attributes.

[5] The LABEL-LEX-sw lexicon is provided by LabEL, through `http://label.ist.utl.pt`.

gata,gato.N+z1:fs	gatita,gato.N+z1:Dfs
gatas,gato.N+z1:fp	gatitas,gato.N+z1:Dfp
gatinha,gato.N+z1:Dfs	gatito,gato.N+z1:Dms
gatinhas,gato.N+z1:Dfp	gatitos,gato.N+z1:Dmp
gatinho,gato.N+z1:Dms	gato,gato.N+z1:ms
gatinhos,gato.N+z1:Dmp	gatos,gato.N+z1:mp

Figure 3 An example of "LABEL-LEX-sw" lexicon entries.

Beyond using this lexicon for validating the lemmas produced by the lemmatizer, we have used the fact that each entry of the lexicon contains the inflected form, lemma, syntactic category, syntactic subcategory, and morphological attributes, that can be directly applied in the lemmatization process. Fig. 3 shows an example of these entries.

Using this lexicon provided an easy way of retrieving the lemma of any word, given its syntactic category. Also, the rules previously defined are now used only when a word is not found in the lexicon, with one advantage: virtually all exceptions to the rules are already present in the lexicon, so that the probability of a rule failing is extremely low. This comes from the fact that the exceptions are usually found in extremely frequent, ancient, and well known words of a lexicon, rather than in more recent, less used, or obscure words.

However, this does not mean that the lexicon could be used right out of the box. Some issues had to be addressed: (i) a mapping between the syntactic categories present in the lexicon and the ones used on the rest of the program (including the rules used in earlier versions); (ii) excluding all pronoun and determiner lexicon entries, as they present disputable normalization – for instance, `tu` (`you`) has `eu` (`I`) as its lemma; and (iii) making optional some gender normalizations, such as presenting `homem` (`man`) as the lemma of `mulher` (`woman`).

When a word is shared by multiple lemmas, the lemma with the highest frequency is selected. For this purpose, we used the frequency list of the combined lemmas present in all the Portuguese corpora available through the AC/DC project [11].[6]

The only drawback of the method is the time it takes, even if only a couple of seconds, to load the lexicon into memory. Other than that, it is quite performant, as the lexicon is stored in a hash structure, that is known to be a fast method for storing and searching on sets of elements. A lemma cache is also used, with each word that is found in the analysed text to be stored together with is syntactic category (POS tag) and lemma, at run-time, which avoids searching again the whole lexicon or selecting which rule to apply for a word already processed. The cache is a particular improvement to performance speed because, besides a set of words commonly used across different domains, texts on a specific topic tend to have their own set of words that are used multiple times over and over again. The basic structure of the currently used lemmatization algorithm is presented in Listing 1.

Regarding flexibilization, besides the customization of rules and selection of which normalization steps to apply, the current version of the lemmatizer allows the option to add new entries to a custom lexicon (if it fits best to do so in a lexicon, instead of specifying an exception in the rules, or both).

[6] The frequency lists of AC/DC are provided by Linguateca, through `http://www.linguateca.pt/ACDC`.

Listing 1 Overview of the lemmatization algorithm used.

```
load lexicon;
load rules;

lemmatize (token, tag) {
  if cache contains (token, tag) {
    return lemma of (token, tag);
  }
  if lexicon contains (token, tag) {
    add (token, tag) to cache;
    return lemma of (token, tag);
  }
  lemma = token;
  for each rule in (adverb, number, superlative, augmentative,
      diminutive, gender, verb) {
    lemma = normalize (lemma, tag, rule);
    if lexicon contains (lemma, tag) {
      add (token, tag) to cache;
      return lemma of (token, tag);
    }
  }
  return lemma;
}
```

4 Evaluation and Results

For the lemmatizer evaluation, we have used Bosque 8.0, the last version of a manually revised part of the Floresta Sintática treebank [1], by Linguateca.[7] Bosque contains around 120,000 tokens with annotations at various syntactic levels, for the Portuguese portion, and around 70,000 for the Brazilian portion.

Bosque was parsed in order to retrieve, for each word found in it, the inflected form, its syntactic category and corresponding lemma. The inflected form and syntactic category were fed to our lemmatizer, and the output was matched against the known lemma, as identified in Bosque.

In Table 1 we can see the overall results using rules, the lexicon, and both rules and lexicon (the current version of the lemmatizer), applied to the Portuguese and Brazilian parts of Bosque, with the current version reaching an accuracy value over 98%. The same table also presents the results broken down into three major syntactic categories: nouns, adjectives, and verbs.

It is possible to notice that using the lexicon greatly improves the normalization of adjectives (although other categories also benefit from it) against using only rules. However, the lexicon only by itself does not cover all the cases either, as it is virtually impossible for a lexicon, comprehensive as it may be, to cover all the lexemes, and associated syntactic categories, in any language. For instance, past participles in the plural form are not contemplated in the used lexicon – that may be one of the reasons rules perform better than the lexicon on verbs, beyond having a more extensive verb list.

[7] Floresta Sintática is freely available from `http://www.linguateca.pt/floresta/BibliaFlorestal/completa.html`.

Table 1 Overall and partial results in major categories.

Bosque	Only Rules	Only Lexicon	Both Rules and Lexicon
Overall PT	97.76%	95.06%	98.62%
Overall BR	97.67%	95.16%	98.56%
Nouns PT	96.94%	98.05%	98.30%
Nouns BR	96.40%	96.67%	97.86%
Adjectives PT	90.10%	95.39%	98.19%
Adjectives BR	88.77%	91.70%	97.23%
Verbs PT	98.04%	88.78%	98.79%
Verbs BR	98.34%	89.59%	99.15%

Table 2 Errors and discrepancies identified in both the lemmatizer and Bosque.

Type	Quantity	Example (*Form#POS:Bosque:LemPORT*)
Incorrect categorization	1.25%	`Afeganistão#N:afeganistão:afeganisto*`
Orthographic errors	3.75%	`ecxemplares*#N:exemplar:ecxemplar*`
Both lemmas acceptable	2.50%	`cabine#N:cabine:cabina`
LemPORT errors	43.75%	`presas#V-PCP:prender:presar`
Bosque errors	48.75%	`pais#N:pais:pai`

The accuracy could be even higher (probably slightly above 99%), as in a significant amount of the faulty cases the problem may actually be found in the Bosque annotation. In other discrepancies, different lemmas could be accepted, depending on the purpose. A brief analysis of a random sample of 10% (160) of the cases where the lemmatizer produced different lemmas on the Portuguese part of Bosque is presented in Table 2.

5 Conclusions and Future Work

We have presented a cross-platform lemmatization tool for Portuguese developed in Java that, through the use of simple rules in conjugation with a comprehensive lexicon, is able to have a very high overall accuracy, over 98%, making it suitable for use in many NLP tasks. For instance, its earlier versions have been used in a question generation system [6], and in the creation of the lexical-semantic resources CARTÃO [8] and Onto.PT [9]. We are also currently using this lemmatizer in a question-answering system under development.

Although the margin for improvement is narrow, we still hope to improve the lemmatizer by addressing some minor but troublesome issues, such as composed and hyphenated words, as well as multiword expressions. We already partially tackle one of these issues, by splitting the words at the hyphen, sharing the syntactic function. However, there are cases where elements of composed and hyphenated words, when put apart, belong to different categories.

Other issues may include the processing of oblique cases in pronouns. Bosque presents the oblique case and the pronoun that would be the corresponding lemma, but we usually process the oblique cases prior in the tokenization process.

We also intend, in the near future, to compare the lemmatizer against jSpell and Freeling, using the Bosque data as input, and processing the output of both tools in order to extract only the lemmas.

In order to be used by other members of the community, the presented lemmatizer is freely available from `https://github.com/rikarudo/LemPORT`, under the moniker "**LemPORT**".

Acknowledgements. This work was supported by the iCIS project (CENTRO-07-ST24-FEDER-002003), co-financed by QREN, in the scope of the Mais Centro Program and European Union's FEDER.

References

1 Susana Afonso, Eckhard Bick, Renato Haber, and Diana Santos. "Floresta Sintá(c)tica": a Treebank for Portuguese. In Manuel González Rodríguez and Carmen Paz Suárez Araujo, editors, *Proceedings of LREC 2002, the Third International Conference on Language Resources and Evaluation*, pages 1698–1703, Paris, 2002. ELRA.

2 Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, New York, USA, 1999.

3 António Branco and João Silva. A Suite of Shallow Processing Tools for Portuguese: LX-Suite. In *EACL'06 Proceedings of the Eleventh Conference of the European Chapter of the Association for Computational Linguistics: Posters & Demonstrations*, pages 179–182, Trento, Italy, 2006.

4 António Branco and João Silva. Very High Accuracy Rule-Based Nominal Lemmatization with a Minimal Lexicon. In *XXII Encontro Nacional da Associação Portuguesa de Linguística*, pages 169–181, 2007.

5 Xavier Carreras, Isaac Chao, Lluís Padró, and Muntsa Padró. FreeLing: An Open-Source Suite of Language Analyzers. In *Proceedings of the 4^{th} International Conference on Language Resources and Evaluation (LREC'04)*, pages 239–242, 2004.

6 Daniel Diéguez, Ricardo Rodrigues, and Paulo Gomes. Using CBR for Portuguese Question Generation. In *Proceedings of the 15^{th} Portuguese Conference on Artificial Intelligence*, EPIA 2011, pages 328–341, Lisbon, Portugal, October 2011. APPIA.

7 Samuel Eleutério, Elisabete Marques Ranchhod, Cristina Mota, and Paula Carvalho. Dicionários Electrónicos do Português. Características e Aplicações. In *Actas del VIII Simposio Internacional de Comunicación Social*, pages 636–642, 2003.

8 Hugo Gonçalo Oliveira, Leticia Antón Pérez, Hernâni Costa, and Paulo Gomes. Uma Rede Léxico-Semântica de Grandes Dimensões para o Português, Extraída a partir de Dicionários Electrónicos. *Linguamática*, 3(2):23–38, December 2011.

9 Hugo Gonçalo Oliveira and Paulo Gomes. ECO and Onto.PT: A Flexible Approach for Creating a Portuguese Wordnet Automatically. *Language Resources and Evaluation*, to be published (online September 2013), 2013.

10 Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Prentice Hall series in artificial intelligence. Prentice Hall, Pearson Education International, Englewood Cliffs, NJ, 2^{nd} edition, 2009.

11 Diana Santos and Eckhard Bick. Providing Internet Access to Portuguese Corpora: the AC/DC Project. In *Proceedings of 2^{nd} International Conference on Language Resources and Evaluation*, LREC 2000, pages 205–210, 2000.

12 Alberto Manuel Simões and José João Almeida. jSpell.pm – Um Módulo de Análise Morfológica para Uso em Processamento de Linguagem Natural. In *Actas da Associação Portuguesa de Linguística*, pages 485–495, 2001.

Expanding a Database of Portuguese Tweets

Gaspar Brogueira[1], Fernando Batista[1], João P. Carvalho[2], and Helena Moniz[3]

1 **Laboratõrio de Sistemas de Língua Falada – INESC-ID, Lisboa, Portugal**
ISCTE-IUL – Instituto Universitário de Lisboa, Lisboa, Portugal
gmrba@iscte.pt,fmmb@iscte.pt

2 **Laboratõrio de Sistemas de Língua Falada – INESC-ID, Lisboa, Portugal**
Instituto Superior Tẽcnico (IST), Lisboa, Portugal
joao.carvalho@inesc-id.pt

3 **Laboratõrio de Sistemas de Língua Falada – INESC-ID, Lisboa, Portugal**
FLUL/CLUL, Universidade de Lisboa, Lisboa, Portugal
helena.moniz@inesc-id.pt

Abstract

This paper describes an existing database of geolocated tweets that were produced in Portuguese regions and proposes an approach to further expand it. The existing database covers eight consecutive days of collected tweets, totaling about 300 thousand tweets, produced by about 11 thousand different users. A detailed analysis on the content of the messages suggests a predominance of young authors that use Twitter as a way of reaching their colleagues with their feelings, ideas and comments. In order to further characterize this community of young people, we propose a method for retrieving additional tweets produced by the same set of authors already in the database. Our goal is to further extend the knowledge about each user of this community, making it possible to automatically characterize each user by the content he/she produces, cluster users and open other possibilities in the scope of social analysis.

1998 ACM Subject Classification H.3.1 Content Analysis and Indexing: Linguistic processing

Keywords and phrases Twitter, corpus of Portuguese tweets, Twitter API, natural language processing, text analysis

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.275

1 Introduction

Twitter is one of the most widely used and well-known social networks worldwide. It allows for rapid communication and experience sharing among its users by providing an infrastructure for sending and receiving messages, containing 140 characters at most. About 646 million users produce approximately 400 million tweets everyday [12, 5]. Twitter is a source of information potentially useful for research in various fields, not only because of the amount of information produced, but also because the access to the data is facilitated for the scientific community by a number of APIs (Application Programming Interfaces).

This work aims at expanding a database of tweets that was collected over eight consecutive days, restricted to geolocated tweets produced in Portuguese regions, and written in Portuguese. The existing data was retrieved using the *statuses/filter* API that allows to fetch tweets with a low latency. The restriction to Portuguese regions was achieved by specifying geographic coordinates that define a number of rectangles covering the mainland and also the Portuguese archipelagos Azores and Madeira. The restriction to the Portuguese language was achieved by using the "*lang*" attribute that is automatically assigned by Twitter. An


© Gaspar Brogueira, Fernando Batista, João P. Carvalho, and Helena Moniz;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, Josẽ Paulo Leal, and Alberto Simões; pp. 275–282

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

overview of the dataset suggests that collected tweets are mostly produced by young people expressing very personal content, often describing family bonds and school activities and concerns. Furthermore, from a preliminary inspection to our 8-day sample we can also say that the data is fairly spontaneous, obviously coded in a written form.

This paper proposes a methodology that is now being used for expanding the database, thus allowing to further characterize the involved community in detail. It is our believe that the current database *per se* is an important resource to characterize part of the Portuguese Twitter community. The database can be used in several perspectives, including: geolocated analysis of users and content; characterization of the different Portuguese regions; age-identification and characterization; sociolinguistic studies; sentiment analysis; and among others. Due to their spontaneous nature, tweets may be eventually used for training spontaneous models for Automatic Speech Recognition (ASR) to cover the absence of models trained specifically with spontaneous data, since models are commonly trained on newspapers and broadcast news. Therefore, future work will tackle the use of tweets to train language models and to evaluate such models in different spontaneous speech domains. The extended version of the database will provide additional data that can be used for extending tasks, such as better assessment of age, twitter usage patterns over the time, vocabulary usage per author, amongst others.

This paper is organized as follows. Section 2 describes the related work in terms of twitter data collections, specially targeting the Portuguese language. Section 3 describes the existing corpus and presents a number of statistical elements concerning the tweet content. Section 4 proposes a methodology for extending the existing data and analyses the properties of the recently appended data. Finally, 5 presents the conclusions and overviews future trends.

2 Related Work

Previous studies involving Portuguese tweets are still scarcely found in the Literature. However, a considerable number of recent work have used Portuguese language Tweets in Sentiment Analysis related tasks [11, 3, 4]. [11] uses a database of 1700 tweets to evaluate the impact of different preprocessing techniques and negation modeling in the tweet sentiment classification. [3] also focuses on Sentiment Analysis, adapting state of the art approaches to Portuguese language. The author uses a collection of 300 thousand tweets, filtered according to the presence of certain verbs, such as “sentir”/feel. Portuguese twitter data was also used by [9] to predict Flu Incidence. In this recent study, the authors use about 14 million tweets originated in Portugal, together with a search engine query logs to estimate the incidence rate of influenza like illness in Portugal. Portuguese tweets are also currently being used for Machine Translation tasks. For example, [6] provides a link to databases of parallel corpora that also include Portuguese language[1].

Finally, an architecture for automatic collecting tweets with a predefined delimited geographic region is proposed by [7]. Their architecture uses a MySQL database for tweets storage and a Twitter Streaming API for accessing and collecting an unlimited number of tweets.

3 Data Analysis

The data here analyzed corresponds to an 8-day period and was collected between March 14th and 21th 2014 [2]. The stream API *statuses/filter* was configured for retrieving geolocated

[1] `http://www.cs.cmu.edu/~lingwang/microtopia/`

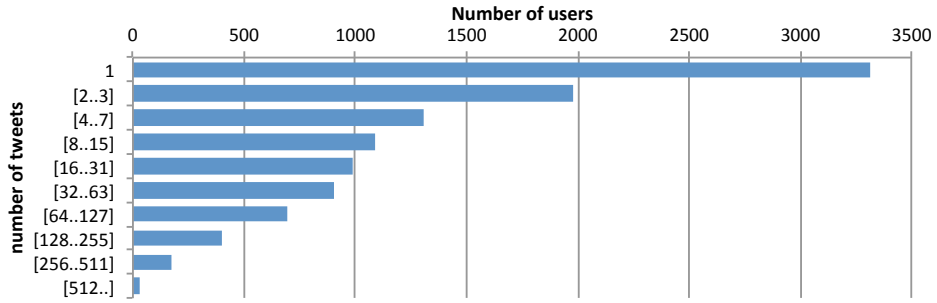


Figure 1 Number of users that have produced a certain quantity of tweets.

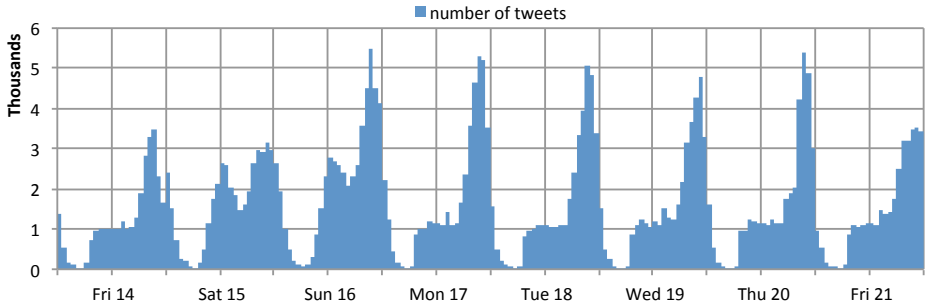


Figure 2 Distribution of tweets by hour.

tweets produced in Portugal. A total of 307K tweets were collected, corresponding to a daily average of about 48K tweets. The set of tweets being analyzed was produced by 11391 distinct users. Figure 1 represents the number of users that produced tweets in a given interval. Most of the tweets are, in fact, produced by distinct users. However, more than half of the users have produced more than one tweet during the period in analysis. The number of tweets varies considerable per user, ranging from 1 to around 1100 tweets in an 8-day period. This behavior justifies our approach described in Section 4 for extending our database, *i.e.*, a deeper knowledge of each user allows for a more suitable analysis of inter-users' traits.

The remainder of this section presents more detailed statistics concerning the number of produced tweets and their temporal distribution, and then focuses on characterizing the content of each tweet.

3.1 Temporal Distribution of the Collected Tweets

It is known that the number of produced tweets is not linear in time. Figure 2 presents the activity by hour, depicting that during friday evenings the number of tweets is lower than during the remainder of the evenings, suggesting that Twitter usage is mainly domestic during the evenings, *i.e.*, users do not usually tweet as much when going out with friends. In fact, users are less active during the day. However 43% of all the user activity is performed during that period which represents a considerable proportion. Taking that into account, we have made attempts to further characterize this community of users. The content we have observed suggested that tweets were mostly produced by teenagers. So, we have made an attempt to characterize the involved community in terms of age, which is not a trivial task because that information is not clearly provided within the tweet content. We have

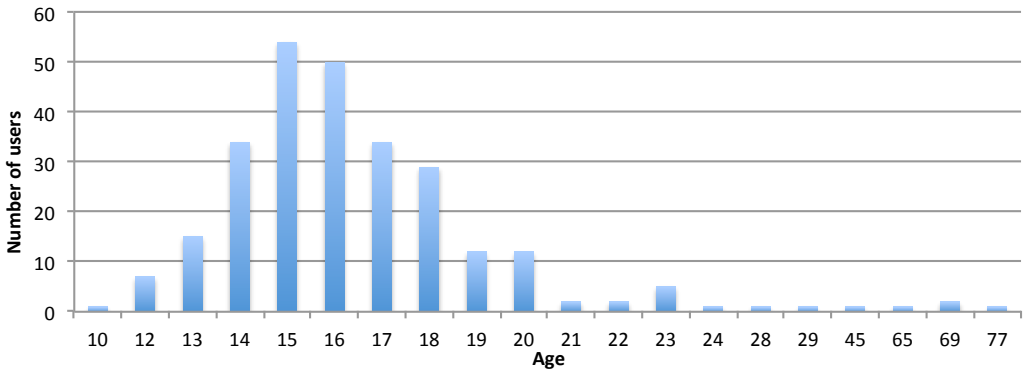


Figure 3 Distribution of the users age.

found that the user description associated to each user sometimes contains that information embedded in the text. Examples:

Paredes | 13 anos | Eminem
Ola tenho 14 anos e sõ sei dormir.
metro e meio de gente / 20 anos / ESTSP

In order to have an idea of the user age we have manually tagged about 265 users with their potential age, based on their own description. The resultant information is depicted in Figure 3, clearly revealing a predominance of teenagers and young adults, as expected by the previous analysis performed on the content. The extended version of the database described in Section 4 will provide additional data that can be used for better assessment of the age, twitter usage patterns over time, vocabulary usage per author, and age stylistic effects.

3.2 Content Analysis

One of the most interesting Twitter particularities is the 140 characters message length limit, leading to messages containing an expected low number of words. The data reveals a trend in the increase of the number of words per message along the day. The maximum value is attained around the hour of maximum Twitter activity.

Regarding tweets' content, Table 1 shows the most frequent trigrams from the 8-days in analysis. The trigrams' frequency is a key-factor for the understanding of the lexical selection

Table 1 Top trigrams.

Trigrams	Freq.
a minha mãe	1395
o meu pai	903
sei o que	746
o que ẽ	741
com a minha	704
tudo o que	634
toda a gente	621
com o meu	617
A minha mãe	601

Trigrams	Freq.
fim de semana	527
o que eu	459
todos os dias	450
a minha vida	444
que a minha	431
como ẽ que	428
porque ẽ que	424
que o meu	415
tenho de ir	412

Trigrams	Freq.
acho que vou	394
que ẽ que	386
a dizer que	382
dia do pai	376
ẽ que eu	373
ir para a	368
como Assunto do	353
e a minha	350
o meu irmão	345

Table 2 Top hashtags ordered by number of users that refer them.

Hashtag	users	Freq
#lt	631	1555
#lrt	616	1403
#np	529	1882
#twitteroff	238	439
#portugal	185	377
#carregabenfica	175	617
#lisbon	125	249
#lisboa	110	231
#love	102	132
#friends	94	132
#selfie	90	98
#nw	83	116

Hashtag	users	Freq
#sun	79	97
#somosporto	69	171
#me	61	103
#night	61	67
#porto	61	105
#beach	56	66
#happy	56	68
#valetudo	55	89
#sunset	54	57
#benfica	50	146
#excluidadasociedade	50	58
#sad	48	57

used by tweeters. We can say that Portuguese tweets are mostly focused on personal messages based on family bonds, as illustrated in the selection of words from the same semantic field – "mãe"/mother; "pai"/father; "irmão"/brother; "irmã"/sister. Moreover, there is also a semantic field associated with school, encompassing vocabulary such as "teste"/test; "a minha turma"/my class; "aula"/lesson (not displayed in Table 1 for legibility issues), suggesting a strong activity of teenagers/young adults.

Another lexical clue to the characterization of the personal component of the tweets is the use of first person pronouns (subject "eu"/I; object "me"/me; and possessives "minha"/my; "meu"/my). In what regards verbal forms, either than being inflected in the first person, the selection of epistemic verbs ("sei"/know, "acho"/think) is a crucial indicator of the way speakers communicate their doubts and certainties, mostly associated with values of like/dislike. Clefts ("ẽ que"/it's ... that) are another very frequent structure selected by tweeters. These structures are used to focus particular constituents, as an emphatic structure.

From the above set of lexical-syntactic options of the tweeters we are able to characterize the 8-days sample tweets as very personal data, written in first person, communicating believes and emotions.

In line with the personal trait of tweets is the use of emoticons, which are pictorial representations associated with distinct emotions, allowing for the expression of feelings in e-contexts. Even though some authors claim that emoticons are used mainly by teenagers and young adults [1, 8] we feel that nowadays emoticons' use is widespread among age groups and no conclusions can be taken from this fact. Emoticons, such as :), :-), :3, ;), and :)) express joy, while emoticons such as :(, :$, :/, :((, :-(expressed sadness. The set of emoticons found in our database is similar to the ones reported by [10] for English tweets.

Only a small number of tweets include hashtags (about 4.3%). This is a rather low number when comparing to other Twitter data collections. [14] reports about 11% of tagged tweets for Portuguese, which includes not only European Portuguese but also Brazilian Portuguese. Moreover, Portuguese is one of the languages that uses fewer hashtags from the 8 languages analyzed. Finally, in a similar database of about 1.5 Million tweets written in Portuguese (including all varieties of Portuguese), collected during the same time period without restricting the location, and where most of the tweets are written in Brazilian Portuguese, such value corresponds to 10.2%.

The top most frequent hashtags are expressed in Table 2 and include #lt (Last or Latest Tweet), #lrt (Last or Latest retweet), #np (Now Playing) used whenever someone is listening

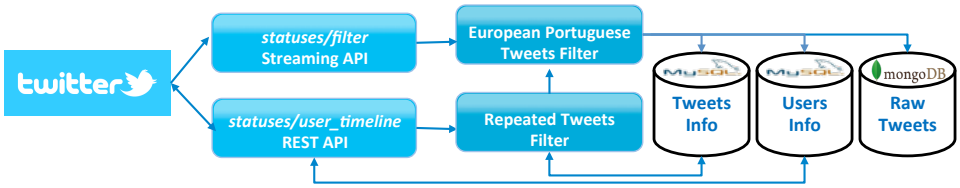


Figure 4 Diagram of the proposed data collection infra-structure.

to a song and wants to share it, and #twitteroff (Enough tweets for today). Nevertheless, we observed that the frequency of such hashtags was relatively low in the similar database mentioned in the previous paragraph of 1.5 Million tweets. #tl (position 69), #lrt (position 146), #np (position 109), #twitterof (position 643).

Another interesting fact concerns the number of retweets (RT), which is very low in our current database, corresponding to roughly 0.1% of the tweets. About 26% of the tweets from the 1.5Million tweets previously mentioned database are retweets. Two possibilities exist concerning this point. The first possibility is that geolocated tweets do not usually include retweets. Another hypothesis is related with the fact that tweets from our database are mostly personal are therefore not usually retweeted.

4 Database Expansion

Twitter APIs give free access to millions of tweets and can be classified into two broad categories, according to their design and access methods. The streaming API provides a continuous data flow of public tweets, where the data is continuously updated immediately after a request until an interrupt request. The REST API is based on the concept of *Representational State Transfer* APIs for web design permits to collect data for specific users. These two APIs accept different parameters and are suitable for quite different purposes. They are also equally restricted in terms of the type of data and the amount of data they can provide, so that a straight use of the available APIs is often not enough to obtain the required information. We propose a methodology that allows circumventing this restriction for collecting and storing Portuguese tweets

The proposed methodology is illustrated in Figure 4 and provides a way of collecting tweets produced in a given region and then expanding this initial set by collecting additional tweets, produced by the same set of users. It comprises two stages: i) collect and filter tweets containing geographic information, which are available through the streaming API statuses/filter. Tweets are filtered by several rectangles, defined by coordinates, corresponding to Portuguese regions, including the Portuguese archipelagos Azores and Madeira. At this point, tweets must also meet the restriction of being written in Portuguese, which is performed by checking the language attribute assigned by Twitter. For each successfully validated tweet, the tweet *user id* is extracted and stored in a MySQL database together with other tweet specific information, such as *tweet id*, and *creation date*; ii) for each of the stored *user id*s use the REST API for retrieving user timelines, allowing to retrieve at most the last 3200 tweets of each user [13].

The second stage entirely depends on the information stored about users and tweets in the first stage. However, the process of retrieving the latest tweets from a user timeline is currently limited to 45 requests per hour, each one returning a maximum of 200 tweets from that specific user. Consequently, fetching an entire user timeline may take 16 queries, which are also limited to 180 per 15 minutes windows [5]. Concerning this process, a number of

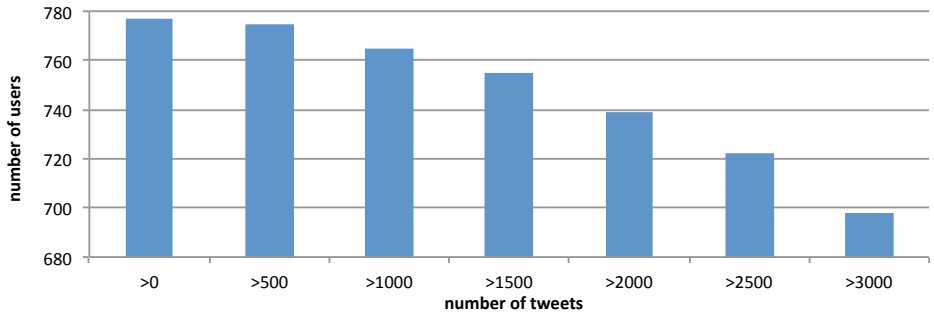


Figure 5 Number of users for which more than a certain amount of tweets was retrieved.

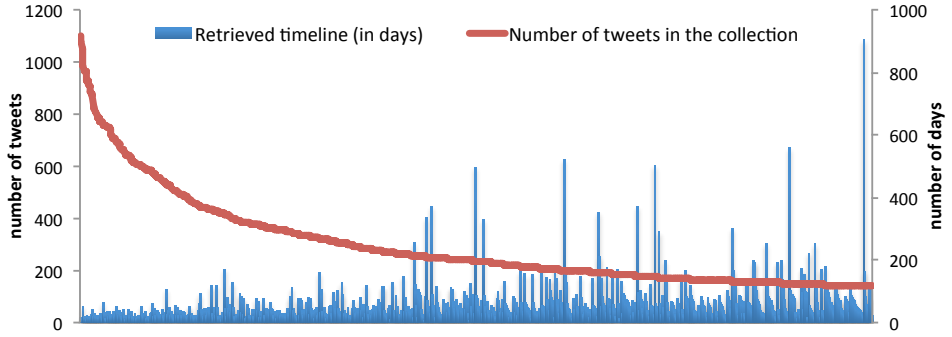


Figure 6 Relation between the time period covered by the retrieved timeline and the activity of a user during the 8-day period.

optimizations have been performed in order to avoid unnecessary queries. All information is stored in a MongoDB database, especially suitable for storing unstructured information, where duplicated entries are detected.

4.1 Analysis of the Recently Collected Data

By the time this document was written we had collected the user timeline of the 777 most active users during the 8-day period. In average, we have retrieved about 3167 tweets per user, totaling about 2.3 Million additional tweets. Figure 5 shows the number of users for whom the number of retrieved tweets achieved a given threshold.

The time period covered by the retrieved timeline varies according to the user activity, as expected. Therefore, the timeline of a very active user can go back as much as 8 days, while the timeline for a less active user can go back more than 900 days. Figure 6 illustrates such relation and supports the idea that more information about a less active user can nevertheless be retrieved by using our proposed approach.

5 Conclusions and Future Work

The information produced by a community through a social network provides means to characterize such community over a vast number of perspectives. The interactions between the community members provide information that until now were very difficult to discover. On Twitter, the interaction between users is carried out by small messages that can be used to express everything, from personal feelings to serious news. We have described a database

of collected geolocated tweets produced in Portugal. Our current database, containing tweets from 8 consecutive days aggregates about 300 thousand messages written in Portuguese and produced in Portugal. We have characterized the collected data and we think it is a valuable resource for studying part of the Portuguese community that is now using social networks. We have found that such small community is mostly composed of young users who use this social network to exchange personal messages. The paper describes a method for expanding the database with related tweets, produced by the same community of users, by combining different Twitter APIs. The proposed methodology is now being applied, and by now allowed to collect 10 times more tweets than the original ones, corresponding to less than 10% of all the users in the database.

In a near future, we expect to complete retrieve the user timeline of our existing users, in order to further characterize the community of users. We also aim at studying different time periods, such as vacations where scholar subjects would not be so frequent.

Acknowledgments. This work was supported by national funds through FCT – Fundação para a Ciência e Tecnologia under projects PTDC/IVC-ESCT/4919/2012 (MISNIS) and PEst-OE/EEI/LA0021/2013, and under Grant SFRH/BPD/95849/2013.

References

1 A. Brito. O discurso da afetividade e a linguagem dos emoticons. *Revista Eletrônica de Divulgação Científica em língua Portuguesa, Linguística e Literatura*, 9, 2008.

2 G. Brogueira, F. Batista, J. P. Carvalho, and H. Moniz. Towards a characterization of tweets geolocated in Portugal. In *PROPOR 2014*, 2014 (submitted).

3 Tiago Daniel Sá Cunha. Sentiment analysis on Twitter's Portuguese language. Technical report, Faculdade de Engenharia da Universidade do Porto, 2013.

4 Eduardo Santos Duarte. Sentiment analysis on Twitter for the Portuguese language. Master's thesis, Faculdade de Ciências e Tecnologia, UNL Lisboa, 2013.

5 Shamanth Kumar, Fred Morstatter, and Huan Liu. *Twitter Data Analytics*. Springer, New York, NY, USA, 2013.

6 Wang Ling, Guang Xiang, Chris Dyer, Alan Black, and Isabel Trancoso. Microblogs as parallel corpora. In *Proceedings of the 51st Annual Meeting on Association for Computational Linguistics*, ACL'13. Association for Computational Linguistics, 2013.

7 M. Oussalah, F. Bhat, K. Challis, and T. Schnier. A software architecture for twitter collection, search and geolocation services. *Knowledge-Based Systems*, 37(0):105–120, 2013.

8 Yanghui Rao, Qing Li, Xudong Mao, and Liu Wenyin. Sentiment topic models for social emotion mining. *Information Sciences*, 266(0):90–100, 2014.

9 José Carlos Santos and Sérgio Matos. Predicting flu incidence from Portuguese Tweets. In Ignacio Rojas and Francisco M. Ortuño Guzman, editors, *IWBBIO*, pages 11–18. Copicentro Editorial, 2013.

10 Tyler Schnoebelen. Do you smile with your nose? Stylistic variation in Twitter emoticons. *Working Papers in Linguistics*, 18(14), 2012.

11 Marlo Souza and Renata Vieira. Sentiment analysis on twitter data for portuguese language. In *Computational Processing of the Portuguese Language*, volume 7243 of *Lecture Notes in Computer Science*, pages 241–247. Springer Berlin Heidelberg, 2012.

12 Statistic Brain. Twitter statistics `http://www.statisticbrain.com/twitter-statistics/`, 2014.

13 Twitter. Documentation, `https://dev.twitter.com/docs/`, 2013.

14 Wouter Weerkamp, Simon Carter, and Manos Tsagkias. How people use twitter in different languages. In *Proceedings of the ACM WebSci'11*, Koblenz, Germany, 2011.

MLT-prealigner: a Tool for Multilingual Text Alignment

Pedro Carvalho and José João Almeida

Departamento de Informática, Universidade do Minho
Braga, Portugal
{pedrocarvalho,jj}@di.uminho.pt

Abstract

Parallel text alignment is a key procedure in the automated translation area. A large number of aligners have been presented along the years, but these require that the target resources have been pre-prepared for alignment (either manually or automatically). It is rather normal to encounter mixed language documents, that is, documents where the same information is written in many languages (Ex: manuals of electronic devices, touristic information, PhD thesis with dual language abstracts, etc).

In this article we present MLT-prealigner: a tool aimed at helping those that need to process mixed texts in order to feed alignment tools and other related language systems.

1998 ACM Subject Classification I.7.2 Document Preparation

Keywords and phrases parallel corpora, multilingual text alignment, language detection, Perl, automated translation

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.283

1 Introduction

With the rising importance of multilingualism in language industries, parallel corpora, which consists of source texts along with their translations into other languages, have become a key resource on natural language processing and, in special, the translation area.

Parallel text alignment is a necessary step so that we can generate all sorts of data based on these texts. Although we can easily find a number of tools that already tackle this subject (like Hunalign [10] and Giza++ [5]), it is still rather difficult to automatically align texts in two or more different languages when they are all contained in the same document.

While working on the Per-Fide project [1] we stumbled upon various sources where files like the ones referred above were encountered and were considered useless unless some kind of preprocessing was made on them. In many of these files the texts and their translations where divided in similar fashion. That lead us to catalog a number of patterns that could be easily found and dealt with. After this identification and cataloging process, we reached the conclusion that a generic solution could be proposed to each and every one of these patterns. That was the main motivation that lead to the creation of this tool.

MLT-prealigner is intended to be used as a preprocessor that can consume documents like the ones referred above and render them usable by any classical aligner. It is intended to separate these documents into multiple files and also to tag the output, according to their language, paving the way for any further processing.

This tool was made to tackle the documents that follow the patterns that we will thereafter present, and although it is not a *one size fits all* solution we think that it is a much needed help to multilingual text alignment and to the translation area in particular.


© Pedro Carvalho and José João Almeida;
licensed under Creative Commons License CC-BY
3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 283–290

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Outline. In the following pages we state the problem more clearly in Section 2. The tool's architecture will be presented in Section 3. Case Studies and results can be viewed in Section 4. Finally a brief discussion will be made in Section 5.

2 Background

The problem of language identification has been address (with good results) in a large number of articles and tools. However some specific problem constraints and details raise the necessity of building new solutions or new variants. In this world we can find problems like: (i) classification of small units (e.g. one line sentences); (ii) the use of minority, or poorly covered, languages (e.g. Tetum and its variants); (iii) documents with (large amounts of) language independent words (e.g. Entities, Person names, bibliography). This paper will focus on a specific problem, multilingual documents.

When gathering resources for multilingual text alignment for our different projects, documents with multiple translations contained in them were a common sight. Usually, these documents followed certain patterns and it was easy to see that dealing with them was a repetitive task. As such, these patterns were cataloged, and served as a starting point for MLT-prealigner's core idea. The most common patterns were:

- Documents where the languages are all in sequential sections that do not intertwine (example: abstracts in thesis, articles, etc.), but there is no clear separation point between them. This pattern is called *contiguous sections*.

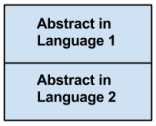


Figure 1 Contiguous sections in different languages pattern.

- Documents where the languages are separated by clear markers. The two most common markers were pages and paragraphs. So, we cataloged two cases, *zipped pages* and *zipped paragraphs*.

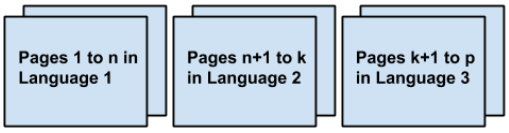


Figure 2 Zipped paragraphs/pages pattern.

- Documents where the languages are contained in columns, different languages in different columns (we call this *zipped columns*).

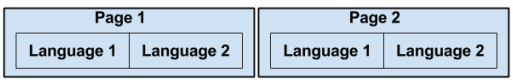


Figure 3 Zipped columns pattern.

Some of these examples are discussed in Section 4.

In the present, we have access to publicly available good quality language identifying tools, such as: (i) command line tools that have a very good response for most cases, provided they have a minimum document length (e.g. 15 words); (ii) web services (SOAP, REST) such as Open Xerox [11]; (iii) libraries for all the popular programming languages, (e.g. `Lingual::Identify::CLD` [9] for Perl); (iv) works like King et al. [4] presented promising results on mixed-language documents.

Although we use generic tools (such as `Lingua::Identify::CLD`) to do some language classification activities, we realized that it was useful to build a project-specific module in order to have better control on the configuration details. That lead to the creation of our tool, presented in the following section.

3 MLT-prealigner Architecture

We took a dictionary based approach using the `Text::Aspell` Perl module, and public available dictionaries from the Aspell project and similar. Our intention was to create a tool that was able not only able to separate languages that are within multilingual documents in an automated manner (with a high level of precision), but also easily customizable, so that the user has more control.

Problems regarding document format will be disregarded in this paper. We have built some inside-solutions, but the focus of this paper will be upon the extracted text and its manipulation. Experience showed that in some situations, it is useful to use this tool in collaboration with cleaning tools (ex bookcleaner [7, 8]) to remove some document noise (ex: page numbers, headers and footers, etc).

3.1 Work-flow

The process is sequential. Our input file passes first through our *splitter* component, which divides the file's content into *smaller chunks* and then redirect the file to the *tagger* component. That component will analyze each chunk and try to identify which language it is written in, it will then tag the chunk accordingly. The next step is creating the output files based on the *tagged chunks*, that job is done by the *unzipping* component, which will create a new file for every language that has been identified, pasting in it the chunks that belong to that language.

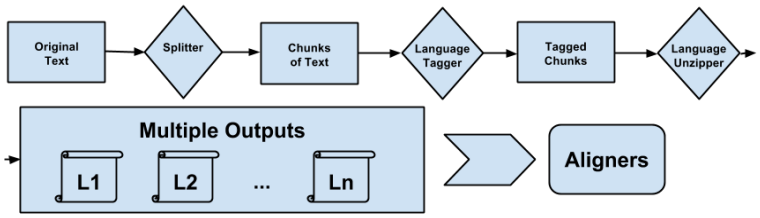


Figure 4 MLT-prealigner's work-flow.

3.2 Splitting

We observed that the splitting logic could be cataloged in two major families. Firstly we have the case where just applying out splitting action would lead to an estimate of the languages of our resulting chunks. Good examples for this kind of splitting are documents with zipped

Require: text chunk and dictionaries list
1: break the chunk in words
2: **for all** words **do**
3: **for all** dictionaries **do**
4: **if** word is contained in dictionary **then**
5: increment dictionary's counter
6: **end if**
7: **end for**
8: **end for**
9: **if** a dictionary has a definitive advantage over the others **then**
10: **return** dictionary's language
11: **else**
12: call tiebreaker algorithm
13: **end if**

Figure 5 Main algorithm for language detection.

pages or zipped columns, if we know beforehand that page 1 is written in language L_1 and page 2 is in language L_2, then splitting both pages should be sufficient to extract both translations. This means that in the next step of our work-flow only a language verification is needed instead of trying to identify it from scratch. In these cases, usually, the separation point between languages is very clear and should be used as our *splitting target*. Our second splitting family is, naturally, the antitheses of our first case. In cases where there is no possibility to estimate the languages of the resulting chunks (there is no clear separation point between languages) then a more generic split has to be applied, and a full language identification has to be made. At the moment, the splitting strategy is manually defined.

3.3 Language Detection and Tagging

MLT-prealigner uses a dictionary based approach to language detection. The main principle is cross checking words against dictionaries; these dictionaries may be user defined and it is possible to build a dictionary from a list of words, this case is very useful for minority languages, like Tetum, where it is very difficult to obtain a robust spell-checker dictionary. Our implementation of this component uses `Text::Aspell` [2] Perl module.

3.3.1 Main Algorithm for Language Detection

As said above, the main principle of the algorithm is cross checking words against dictionaries. When a dictionary has a high enough number of words contained in it, it's language will be selected as the language of the chunk of text that is being processed. In the case that no specific dictionary was able to breakthrough then another algorithm will be applied, to these algorithms we gave the name *Tiebreaker algorithms*. At the time this article was written it was still a bit unclear at which point we could safely say that a specific dictionary had *broken through*, or even if this threshold should be user defined. A rough sketch of the algorithm is presented in Figure 5.

3.3.2 Tiebreaker Algorithms

Because there is the possibility that the main algorithm may not be able to detect, with a high degree of confidence, the language of all chunks of text, we had to envision certain ways that we could do a deeper inspection of the text so that a decision could be reached. Several tiebreaker algorithms were proposed and as we found out, the success percentage of these algorithms directly depended on the language division pattern of the document that was being processed. So one algorithm could be more suitable for one type of problem, but when in another context another algorithm would be more suited for that specific problem. By

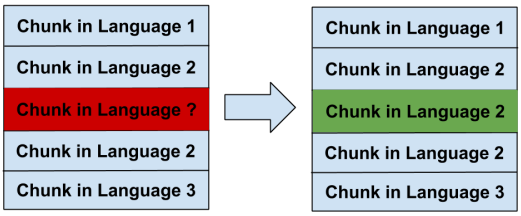


Figure 6 Language scheme after the algorithm.

giving the user direct control on what tiebreaker algorithm to apply, we are also giving the tool a new level of flexibility.

Neighborhood Algorithm. In certain patterns, the fact that a chunk of text is surrounded by other chunks, that share the same language between them, is a clear indicator that this chunk in itself may be written in that language as well. So if we can detect that the immediate predecessor and successor of the chunk in question are both written in the same language, then we assume that the chunk itself belongs to that language (Figure 6).

This type of reasoning is very successful in cases where we know before hand that different languages won't mix, instead there are specific sections for each language in the document that is being processed. One such case will be addressed in Section 4.1.

Reversed Neighborhood Algorithm. Just as the name indicates, this algorithm is the exact opposite of the former one. In cases where we previously know that the chunks of text will be intercalated according to their language, we can detect that the immediate predecessor and successor are of *Language 1* and so the chunk of text that is being analyzed is of *Language 2*. This reasoning is suitable for cases like *zipped paragraph*. A case study that tackles this pattern will be presented in Section 4.3.

Dimension Algorithm. In some cases it is expected that the resulting chunks in each identified language do not vary a lot in size. To verify if this similarity in size between the chunks is as expected a metric was defined. Considering σ as our *size function*.

$$max(\sigma(L_1, L_2)) \geq min(\sigma(L_1, L_2)) \times \delta$$

where δ is the *size coefficient* that is defined as our threshold, for example, if $\delta = 2$ then the biggest chunk was expected to be, at most, twice as big as the smallest one.

This kind of reasoning can very well be applied in cases as *zipped pages* and *zipped columns*, as long as the number of the target pages or columns is more or less equal. When the output does not match the expected dimensions, then a finner analysis has to be made, this means reducing the set of active languages and/or decreasing the size of the text chunks.

3.4 Output Generation

After the original text has been split and all the chunks tagged, the next step is to produce the output files. MLT-prealigner allows for two types of output generation, the process that we call *unzipping* and also the creation of a *tagged file*.

We call *language unzipping* to the creation a new output file for each language that has been detected. The text chunks are distributed accordingly to the tag that has been previously inserted.

The other type of output generation that is available is the creation of a file with all the tagged chunks. This is specially useful when our aim is not text alignment in itself, instead we may want to do another kind of processing, like obtaining statistics about the chunks. A small example of one of this type of file can be seen below.

> `[en]` The classic crown ethers are macrocyclic polyethers that contain between 3 and 20 oxygen atoms, separated from each other by two or more carbon atoms. `[/en]`
>
> `[pt]` Os éteres coroa clássicos são poliéteres macrocíclicos que contêm 3 a 20 átomos de oxigénio, separados entre si por dois ou mais átomos de carbono. `[/pt]`

We ended up using this notation instead of XML due to potential problems that could arise due to the appearance of nested XML tags inside our documents. To identify each chunk we used the same nomenclature as the Aspell dictionaries. For instance, standard Portuguese will be tagged with `[pt_PT]` and Brazilian Portuguese with `[pt_BR]`[1].

4 Case-studies and Results

Because this tool was created pretty much out of necessity, as we encountered several text alignment issues when trying to undertake certain projects, it seemed interesting and relevant to present some of the problems we faced and were solved using MLT-prealigner. In the next subsections we will discuss these problems and how we proceeded to solve them.

4.1 Abstracts of Rcaap Thesis

The Portuguese Open Access Scientific Repository (Rcaap)[6] is home to a large number of academic thesis. This type of document usually has an abstract translated in one or more languages. Knowing this, we undertook a small project with the intention of creating a *parallel text corpus* based on the collection of abstracts available on the repository, which were very rich in terms of technical terminology, however approximately 31% of these abstracts were contained in single documents with all the translations mashed up together, one after another, just as in Figure 1.

To be able to use these documents we had to find a way to split the chunks of text according to its languages, originating a new document for each language detected, in the case of the example above, 3 documents would be produced. The main barrier was to find the separation point between languages so we could divide the text into multiple documents.

We chose to split the texts by *sentence*, and because we knew before hand that the languages would come one after another, it seemed that this was a pattern suitable to use the *neighborhood algorithm* as the tiebreaker.

We made a small analysis on the outputs, 50 random files were chosen, and realized that 1043 sentences were processed and 13 of them were mistakenly tagged with the wrong language, this gives us a 98.75% success percentage. A big part of those errors were due to citations inside the text, very common in academic texts like this, where, for example, the expression “et al”, which is very common in citations, would almost always make MLT-prealigner think the sentence was in French.

[1] The full list of dictionaries can be found in `ftp://ftp.gnu.org/gnu/aspell/dict/0index.html`

4.2 Multilingual Manuals of Electrical Appliances

In this case-study we describe the separation of a set of PDF multilingual manuals of electrical appliances of Teka[2]. In near 80% of the available manuals, we found several different languages in the same PDF document. In some situations we had just partial translations.

We noticed the existence of multilingual title-pages after processing the entire corpus. Some problems were created by imperfect PDF to text conversion of texts that included images and other non textual elements.

Observing just the Portuguese Spanish pair, we obtained 273 in each language of length varying from 2 to 39 pages. After the alignment process, we obtained 43 alignments marked as "bad-alignment" and 230 that passed the automatic quality check assessment of the project.

4.3 PT-Tetum Bilingual Version of "O anjo de Timor"

In this case-study we discuss the separation of a bilingual text: "O anjo de Timor" [3]. The book is a bilingual document (PT, Tetum). Tetum is a minority language, and therefore not covered by `Lingua::Identify::CLD` [9], neither Aspell dictionaries. The multilingual chunks were separated into paragraphs. Also, pages included headers and footers.

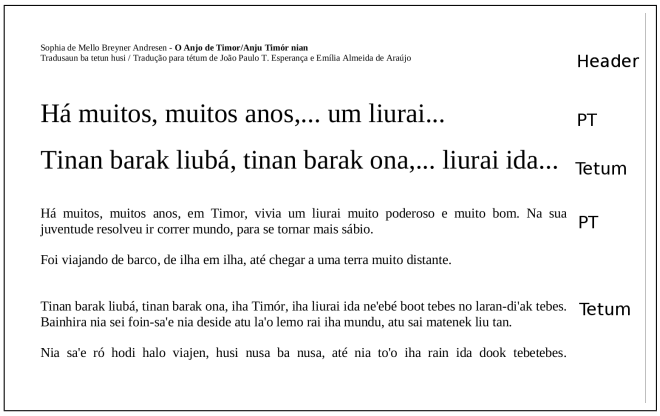


Figure 7 Example bilingual document.

4.4 Results

This tool is in a very preliminary phase, and so evidence of it's utility is still scarce. However, building on the aforementioned case studies, some results were calculated, and our first results are very encouraging. At the moment this paper was written, unfortunately, results on the *Anjo de Timor* case study were not calculated.

5 Conclusions

We believe that this tool will have a big impact in the gathering of new resources, a task that is critical in multilingual text processing, especially in the translation area.

[2] `http://www.teka.com`

Table 1 Preliminary results.

Case-Study	Dimension	Precision
Rcaap	1043 sentences	98.75%
Teka (PT-ES)	2800 pages	(very good lang detection 98%; some PDF problems)
Anjo de Timor	to be calculated	

As new patterns are identified, and new solutions are engineered to deal with them, the tool will grow and become even more useful, that is why one of the main qualities of MLT-prealigner is that it's tiebreaker algorithms set is easily extensible.

We envision that the one of the next steps in the development of the tool will be to build some kind of multilingual text classifier to automatically detect patterns.

Another key point of improvement, is alternating it's dictionary based approach for language detection with an approach based in `Lingua::Identify::CLD`. Because both approaches have their *pros* and *cons*, we believe we may be able to take the best of both worlds.

Finally, the possibility to create robust dictionaries for minority languages, based for example on word lists, is something that should be a great asset to the translation area.

References

1 José João Almeida, Sílvia Araújo, Nuno Carvalho, Idalete Dias, Ana Oliveira, André Santos, and Alberto Simões. The Per-Fide corpus: A new resource for corpus-based terminology, contrastive linguistics and translation studies. In Tony Berber Sardinha and Telma São-Bento Ferreira, editors, *Working with Portuguese Corpora*, chapter 9, pages 177–200. Bloomsbury Publishing, April 2014.

2 Kevin Atkinson. Aspell spell checker, 2011. `http://aspell.net/`.

3 Sophia de Mello Breyner Andresen. *Anjo de Timor, Anju Timór nian.* Instituto Camões, 2003. Bilingual edition Tetum-Portuguese, translated by J. Esperança and E. Araújo.

4 B. King and S. Abney. Labeling the languages of words in mixed-language documents using weakly supervised methods. In *NAACL-HLT*, 2013.

5 F. J. Och and H. Ney. Improved statistical alignment models. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pages 440–447. Association for Computational Linguistics, 2000. Giza++.

6 Rcaap. Project Rcaap – repositório científico de acesso aberto de portugal. (home page), FCT, 2010. `http://www.rcaap.pt/`.

7 André Santos and José João Almeida. Text::Perfide::BookCleaner, a perl module to clean and normalize plain text books. In *Actas del XXVII Congreso de la Sociedad Española para el Procesamiento del Lenguaje Natural*, 2011.

8 André Santos, José João Almeida, and Nuno Carvalho. Structural alignment of plain text books. In Nicoletta Calzolari et al., editors, *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey, may 2012. European Language Resources Association (ELRA).

9 Alberto Simões and Google Chrome Team. `Lingua::Identify::CLD` – Perl interface to Google Chrome Language Detection library, 2013. `http://search.cpan.org/dist/Lingua-Identify-CLD/`.

10 D. Varga, P. Halácsy, A. Kornai, V. Nagy, L. Németh, and V. Trón. Parallel corpora for medium density languages. *Selected Papers from RANLP 2005*, pages 590–596, 2005.

11 Xerox. Open xerox language identifier system, 2013. `http://open.xerox.com/Services/LanguageIdentifier`.