1st Symposium on Languages, Applications and Technologies

SLATE'12, June 21–22, 2012, Braga, Portugal

Edited by

Alberto Simões

Ricardo Queirós

Daniela da Cruz

Editors

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

Ricardo Queirós
Esc. Sup. Estudos Industriais e Gestão
Instituto Politécnico do Porto
ricardo.queiros@eu.ipp.pt

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

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

ISBN 978-3-939897-40-8

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/978-3-939897-40-8.

Publication date
June, 2012

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

Digital Object Identifier: 10.4230/OASIcs.SLATE.2012.i

ISBN 978-3-939897-40-8 **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.

ISSN 2190-6807

www.dagstuhl.de/oasics

To my parents.
Alberto Simões

To my wife, Márcia, and our daughter, Gabriela,
for their unconditional support and love.
Ricardo Queirós

To rocas, zinha and necas.
Daniela da Cruz

Contents

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Short Communications

Preface

This story has to be told. It was ready and mature in our minds for many years of discussions, research and fruitful developments. It is not the story of our group, or of our laboratory. It is about people and about the most primitive need everyone has to live in society: the communication. This process and the ideas around should be converted into an event like this 1^{st} international Symposium on Languages, Applications and Technologies, SLATe'2012, dedicated to everyone interested in the study of languages to dialog with, or through, computers.

As Human Beings invented many different alphabets and various rules to transmit messages among them, in the computers world a similar phenomena emerged. Creativity justifies one part of this variety, but the desire of novelty is not the reason for all. Environment also conditions the choice of vocabulary, transmission media and conventions to follow in the composition of symbols. Another strong influential reason is the community's literacy, education, and cultural heritage.

All these different forms of communication use languages, different languages, but that still share many similarities. In SLATe we are interested in discussing all these languages.

In a computer system, even nowadays, most relevant digital data still circulates in textual format: computer programs, specifications, documents and other information resources have their content in this format. In order to be read and interpreted by a machine, texts should be valid sentences of a known language to be possible to associate them some meaning. The path towards this meaning takes us to the formal definition of syntax and then to the formalization of semantics. So the broad area of *Language Processing*, that is precisely the SLATe framework, is concerned with approaches, methods and tools for the *specification of languages* and the *systematic development of their processors*.

Language syntax and semantics is formally defined by a Grammar; Language Processors are systematic and automatically built using tools, called Generators, that read a grammar and produce the analyzers/translators.

Although we keep the same approach, different kind of texts require specialized methods and tools. So, the area is split into three main research directions that will be covered by SLATe'2012:

1. completely structured texts, that belong to a formal language — we named it as *Processing Human-Computer Languages* (denoted by HCL Track);
2. semi-structured texts, that are compliant to an annotation language — we said *Processing Computer-Computer Languages* (denoted by CCL Track)
3. unstructured texts, that belong to a natural languages — we referred to as *Processing Human-Human Languages* (denoted by HHL Track).

Born from a group of Portuguese researchers that organized over a decade two different conferences — XATA, with interest in XML as the *de facto* language for computer interaction; and CoRTA, with interest in Compilers and related techniques to understand computer

1^{st} Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

languages — and share the fascination by the way languages work, being them natural or artificial, SLATe'2012 arrives as the generalization of the three directions into the abstraction of languages.

The strong motivation and the total enthusiasm of that group, the generous acceptance by Alberto Simões to co-chair the Symposium with Daniela da Cruz and Ricardo Queirós, the quick choice of a small but very effective local organizing team and the privilege of finding immediately, for the three tracks, a large international Program Committee gathering renown names in the area, made possible to bring SLATe to light here in Braga, hosted by Departamento de Informática da Universidade do Minho, which is proud to provide a birth for this first edition of a series of Symposiums on Languages, Applications and Technologies that will attract more and more people and become a great event in a while.

And it is my truly, my full, pleasure to write today this Preface for the SLATe'2012 Proceedings!

Before finishing and after expressing my sincere gratitude to all that make this dream a reality, from the Chairs and Committee Members to our Sponsors, I want to write a warm thank you to Alexander Paar and Josep Silva Galiana that accepted the invitation to enrich the meeting with two talks on *From Program Execution to Automatic Reasoning: Integrating Ontologies into Programming Languages* and *The New Generation of Algorithmic Debuggers*, respectively.

Braga, May 31st, 2012

Pedro Rangel Henriques

Committees

Scientific Committee

Ademar Aguiar
Universidade do Porto, Portugal

Alberto Simões
Universidade do Minho, Portugal

Alda Lopes Gançarski
Institut National des Télécommunications, France

Alexander Paar
TWT Science and Innovation GmbH

António Menezes Leitão
Universidade Técnica de Lisboa, Portugal

António Teixeira
Universidade de Aveiro, Portugal

Bastian Cramer
Universitãt Paderborn, Germany

Bostjan Slivnik
Univerza v Ljubljani, Slovenia

Casiano Rodriguez-Leon
Universidad de La Laguna, Spain

Cristina Mota
Linguateca, Portugal

Cristina Ribeiro
Universidade do Porto, Portugal

Daniela da Cruz
Universidade do Minho, Portugal

Gabriel David
Universidade do Porto, Portugal

Giovani Librelotto
Universidade Federal Santa Maria, Brazil

Hugo Oliveira
Universidade de Coimbra, Portugal

Ian Kóllar
Technical University Kosice, Slovakia

Ivan Lukovic
University of Novi Sad, Serbia

Jean-Cristophe Filliatre
Laboratoire de Recherche en Informatique, France

Jorge Baptista
Universidade do Algarve, Portugal

Josep Silva
Universidad Politécnica de Valencia, Spain

José Carlos Ramalho
Universidade do Minho, Portugal

José João Almeida
Universidade do Minho, Portugal

José Paulo Leal
Universidade do Porto, Portugal

João Correia Lopes
Universidade do Porto, Portugal

João Lourenço
Universidade Nova de Lisboa, Portugal

João Paiva Cardoso
Universidade do Porto, Portugal

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

João Saraiva
Universidade do Minho, Portugal

Jörg Tiedemann
Uppsala University, Sweeden

Lluís Padró
Universitat Politècnica de Catalunya, Spain

Luis Ferreira
Instituto Politécnico do Cávado e Ave, Portugal

Luísa Coheur
Universidade Técnica de Lisboa, Portugal

Maria João Varanda Pereira
Instituto Politécnico de Bragança, Portugal

Mario Berón
Universidad Nacional de San Luis, Argentina

Marjan Mernik
Univerza v Mariboru, Slovenia

Matej Crepinsek
Univerza v Mariboru, Slovenia

Miguel Ferreira
Universidade do Minho, Portugal

Mirjana Ivanovic
University of Novi Sad, Serbia

Nuno Rodrigues
Instituto Politécnico do Cávado e do Ave, Portugal

Pablo Gamallo
Universidade de Santiago de Compostela, Spain

Pedro Rangel Henriques
Universidade do Minho, Portugal

Philipp Koehn
University of Edinburgh, United Kingdom

Piek Vossen
Vrije Universiteit Amsterdam, The Netherlands

Rafael Linz
Universidade Federal de Pernambuco, Brasil

Ricardo Queirós
Instituto Politécnico do Porto, Portugal

Rogério Paulo
Efacec, Portugal

Rui Lopes
Universidade de Lisboa, Portugal

Salvador Abreu
Universidade de Évora, Portugal

Simão Melo de Sousa
Universidade da Beira Interior, Portugal

Tomaz Kosar
Univerza v Mariboru, Slovenia

Vasco Amaral
Universidade Nova de Lisboa, Portugal

Xavier Gómez Guinovart
Universidade de Vigo, Spain

Yorick Wilks
Florida Institute for Human and Machine Cognition, USA

Organization Committee

Alberto Simões
Universidade do Minho, Portugal

Daniela da Cruz
Universidade do Minho, Portugal

Nuno Carvalho
Universidade do Minho, Portugal

Nuno Oliveira
Universidade do Minho, Portugal

Pedro Henriques
Universidade do Minho, Portugal

Ricardo Queirós
Instituto Politécnico do Porto, Portugal

Sara Correia
Universidade do Minho, Portugal

Sara Fernandes
United Nations University (UNU-IIST),
Macau
Universidade do Minho, Portugal

List of Authors

Marc Alier
Services and Information Systems Engineering Department, UPC
Barcelona, Spain.
marc.alier@ipc.edu

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

Miguel Areias
CRACS & INESC-Porto LA
Faculty of Sciences, University of Porto, Portugal
miguel-areias@dcc.fc.up.pt

Mario Marcelo Berón
National University of San Luis, Argentina
mberon@unsl.edu.ar

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

María J. Casany
Services and Information Systems Engineering Department, UPC
Barcelona, Spain.
mjcasany@essi.upc.edu

Miguel A. Conde
Science Education Research Institute (IUCE)
GRIAL Research Group. University of Salamanca, Spain
mconde@usal.es

Ulisses Araújo Costa
VisionSpace Technologies, Portugal
ucosta@visionspace.com

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

Patricia Sotelo Dios
University of Vigo
Galicia, Spain
psotelod@uvigo.es

Brett Drury LIAAD-INESC, Portugal
Brett.Drury@gmail.com

João P. Fernandes
HASLab / INESC TEC
Universidade do Minho, Portugal
jpaulo@di.uminho.pt

Josep Silva Galiana
DSIC, Universidad Politécnica de Valencia, Spain
jsilva@dsic.upv.es

Alda Loves Gancarski
Institut Télécom,
Télécom Sudparis CNRS UMR Samovar, France
alda.gancarski@it-sudparis.eu

Francisco J. García-Peñalvo
Science Education Research Institute (IUCE)
GRIAL Research Group. University of Salamanca, Spain
fgarcia@usal.es

Xavier Gómez Guinovart
University of Vigo
Galicia, Spain
xgg@uvigo.es

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Peter Hancox
School of Computer Science
University of Birmingham, United Kingdom
pjh@cs.bham.ac.uk

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

Jaime Filipe Jorge
Instituto Superior Técnico, Portugal
jaime.f.jorge@ist.utl.pt

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

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

António Menezes Leitão
Instituto Superior Técnico, Portugal
antonio.menezes.leitao@ist.utl.pt

Mariano Luzza
National University of San Luis,
Argentina
mluzza@unsl.edu.ar

Jean-Noël Martin
All4Tex, France
jnm@all4tec.net

Pedro Martins
HASLab / INESC TEC
Universidade do Minho, Portugal
prmartins@di.uminho.pt

Damien Martin-Guillerez
Inria Bordeaux Sud-Ouest, France
damien.martin-guillerez@inria.fr

Carlos Monteiro
CRACS & INESC-Porto LA
Faculty of Sciences, University of Porto,
Portugal
carlosmonteiro@dcc.fc.up.pt

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

Alexander Paar
TWT Science and Innovation GmbH
alexpaar@acm.org

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

Jordi Piguillem
Services and Information Systems
Engineering Department, UPC
Barcelona, Spain.
jpiguillem@essi.upc.edu

Nikolaos Polatidis
formely of School of Computer Science
University of Birmingham, United Kingdom

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

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

Vânia Rodrigues
CRACS & INESC-Porto LA
Faculty of Sciences, University of Porto,
Portugal
c0416098@alunos.dcc.fc.up.pt

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

João Saraiva
HASLab / INESC TEC
Universidade do Minho, Portugal
jas@di.uminho.pt

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

Part I

Keynotes

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz

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

The New Generation of Algorithmic Debuggers

Josep Silva Galiana[1]

1 **DSIC, Universidad Politécnica de Valencia**
Camino de Vera, s/n. E–46022 Valencia, Spain
jsilva@dsic.upv.es

Abstract

Algorithmic debugging is a debugging technique that has been extended to practically all programming paradigms. Roughly speaking, the technique constructs an internal representation of all (sub)computations performed during the execution of a buggy program; and then, it asks the programmer about the correctness of such computations. The answers of the programmer guide the search for the bug until it is isolated by discarding correct parts of the program. After twenty years of research in algorithmic debugging many different techniques have appeared to improve the original proposal. Recent advances in the internal architecture of algorithmic debuggers face the problem of scalability with great improvements in the performance thanks to the use of static transformations of the internal data structures used. The talk will present a detailed comparison of the last algorithmic debugging techniques analyzing their differences, their costs, and how can they be integrated into a real algorithmic debugger.

1998 ACM Subject Classification D.2.5 Testing and Debugging

Keywords and phrases Debugging, Programming, Program Correction

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.3

Presentation type Keynote

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 3–3

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

From Program Execution to Automatic Reasoning: Integrating Ontologies into Programming Languages

Alexander Paar[1]

1 **TWT Science and Innovation GmbH**
alexpaar@acm.org

Abstract

Since their standardizations by the W3C, the Extensible Markup Language (XML) and XML Schema Definition (XSD) have been widely adopted as a format to describe data and to define programming language agnostic data types and content models. Several other W3C standards such as the Resource Description Framework (RDF) and the Web Ontology Language (OWL) are based on XML and XSD. At the same time, statically typed object-oriented programming languages such as Java and C# are most widely used for software development.

This talk will delineate the conceptual bases of XML Schema Definition and the Web Ontology Language and how they differ from Java or C#. It will be shown how XSD facilitates the definition of data types based on value space constraints and how OWL ontologies are amenable to automatic reasoning. The superior modeling features of XSD and OWL will be elucidated based on exemplary comparisons with frame logic-based models. A significant shortcoming will become obvious: the deficient integration of XSD and OWL with the type systems of object-oriented programming languages.

Eventually, the Zhi# approach will be presented that integrates XSD and OWL into the C# programming language. In Zhi#, value space-based data types and ontological concept descriptions are first-class citizens; compile time and runtime support is readily available for XSD and OWL. Thus, the execution of Zhi# programs is directly controlled by the artificial intelligence inherent in ontological models: Zhi# programs don't just execute, they reason.

1998 ACM Subject Classification D.3 Programming Languages

Keywords and phrases Ontologies, OO programming languages, Automatic reasoning

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.5

Presentation type Keynote

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 5–5

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

Part II

Full Communications

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz

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

On Extending a Linear Tabling Framework to Support Batched Scheduling

Miguel Areias and Ricardo Rocha

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

Abstract

Tabled evaluation is a recognized and powerful technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. During tabled execution, several decisions have to be made. These are determined by the scheduling strategy. Whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. The two most successful tabling scheduling strategies are *local scheduling* and *batched scheduling*. In previous work, we have developed a framework, on top of the Yap system, that supports the combination of different *linear tabling strategies* for local scheduling. In this work, we propose the extension of our framework, to support batched scheduling. In particular, we are interested in the two most successful linear tabling strategies, the DRA and DRE strategies. To the best of our knowledge, no single tabling Prolog system supports both strategies simultaneously for batched scheduling.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Linear Tabling, Scheduling, Implementation

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.9

1 Introduction

The operational semantics of Prolog is given by SLD resolution [7], an evaluation strategy particularly simple that matches current stack based machines particularly well, but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. *Tabling* [3] is a proposal that overcomes those limitations. In a nutshell, tabling consists of storing intermediate solutions for subgoals so that they can be reused when a similar subgoal appears. Work on SLG resolution, as initially implemented in the XSB system [9], proved the viability of tabling technology for application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, Program Analysis, among others. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property* [3].

In a tabled evaluation, there are several points where we may have to choose between continuing forward execution, backtracking, consuming solutions from the table, or completing subgoals. The decision on which operation to perform is determined by the scheduling strategy. The two most successful strategies are *local scheduling* and *batched scheduling* [5]. Local scheduling tries to complete subgoals as soon as possible. When new solutions are found, they are added to the table space and the evaluation fails. Solutions are only returned when all program clauses for the subgoal at hand were resolved. Batched scheduling favors forward execution first, backtracking next, and consuming solutions or completion last. It thus tries to delay the need to move around the search tree by batching the return

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 9–24

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

of solutions. When new solutions are found for a particular tabled subgoal, they are added to the table space and the evaluation continues.

The main difference between the two strategies is that in batched scheduling, variable bindings are immediately propagated to the calling environment when a solution is found. However, for some situations, this behavior may result in creating complex dependencies between subgoals. On the other hand, since local scheduling delays solutions, it does not benefit from variable propagation, and instead, when explicitly returning the delayed solutions, it incurs an extra overhead for copying them out of the table.

Currently, the tabling technique is widely available in systems like XSB [12], Yap [10], B-Prolog [13], ALS-Prolog [6], Mercury [11] and Ciao [4]. In these implementations, we can distinguish two main categories of tabling mechanisms: *suspension-based tabling* and *linear tabling*. Suspension-based tabling mechanisms need to preserve the computation state of suspended tabled subgoals in order to ensure that all solutions are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. Linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points and for that they maintain a single execution tree without requiring suspension and resumption of sub-computations. While suspension-based mechanisms are considered to obtain better results in general, they have more memory space requirements and are more complex and harder to implement than linear tabling mechanisms.

In previous work, we have developed a framework, on top of the Yap system, that supports the combination of different linear tabling strategies for local scheduling [1, 2]. As these strategies optimize different aspects of the evaluation, they were shown to be orthogonal to each other for local scheduling. In this work, we propose the extension of our framework, to combine different linear tabling strategies, but for batched scheduling. In particular, we are interested in the two most successful linear tabling strategies, the DRA and DRE strategies [2]. To the best of our knowledge, no single tabling Prolog system supports both strategies simultaneously for batched scheduling. Extending our framework from local scheduling to batched scheduling should be, in principle, smooth but, as we will see, there are some relevant details that have to be considered in order to ensure a correct and efficient integration of the DRA and DRE strategies with batched scheduling.

The remainder of the paper is organized as follows. First, we briefly introduce the basics of tabling and describe the execution model for standard linear tabled evaluation using batched scheduling. Next, we present the DRA and DRE strategies and discuss how they can be used to optimize different aspects of the evaluation. We then provide some implementation details regarding the integration of the two strategies on top of the Yap system. Finally, we present some experimental results and we end by outlining some conclusions.

2 Standard Linear Tabled Evaluation

Tabling works by storing intermediate solutions for tabled subgoals so that they can be reused when a similar (or repeated) call appears[1]. In a nutshell, first calls to tabled subgoals are considered *generators* and are evaluated as usual, using SLD resolution, but their solutions are stored in a global data space, called the *table space*. Similar calls to tabled subgoals are considered *consumers* and are not re-evaluated against the program clauses because they can potentially lead to infinite loops, instead they are resolved by consuming

[1] Two subgoal calls are considered to be similar if they are the same up to variable renaming.

the solutions already stored for the corresponding generator. During this process, as further new solutions are found, we need to ensure that they will be consumed by all the consumers, as otherwise we may miss parts of the computation and not fully explore the search space.

A generator call C thus keeps trying its matching clauses until a fix-point is reached. If no new solutions are found during one round of trying the matching clauses, then we have reached a fix-point and we can say that C is completely evaluated. However, if a number of subgoal calls is mutually dependent, thus forming a *Strongly Connected Component (SCC)*, then completion is more complex and we can only complete the calls in a SCC together [9]. SCCs are usually represented by the *leader call*, i.e., the generator call which does not depend on older generators. A leader call defines the next completion point, i.e., if no new solutions are found during one round of trying the matching clauses for the leader call, then we have reached a fix-point and we can say that all subgoal calls in the SCC are completely evaluated.

We next illustrate in Fig. 1 the standard execution model for linear tabling using batched scheduling. At the top, the figure shows the program code (the left box) and the final state of the table space (the right box). The program defines two tabled predicates, *a/1* and *b/1*, each defined by two clauses (clauses *c1* to *c4*). The bottom sub-figure shows the evaluation sequence for the query goal *a(X)*. Generator calls are depicted by black oval boxes and consumer calls by white oval boxes.

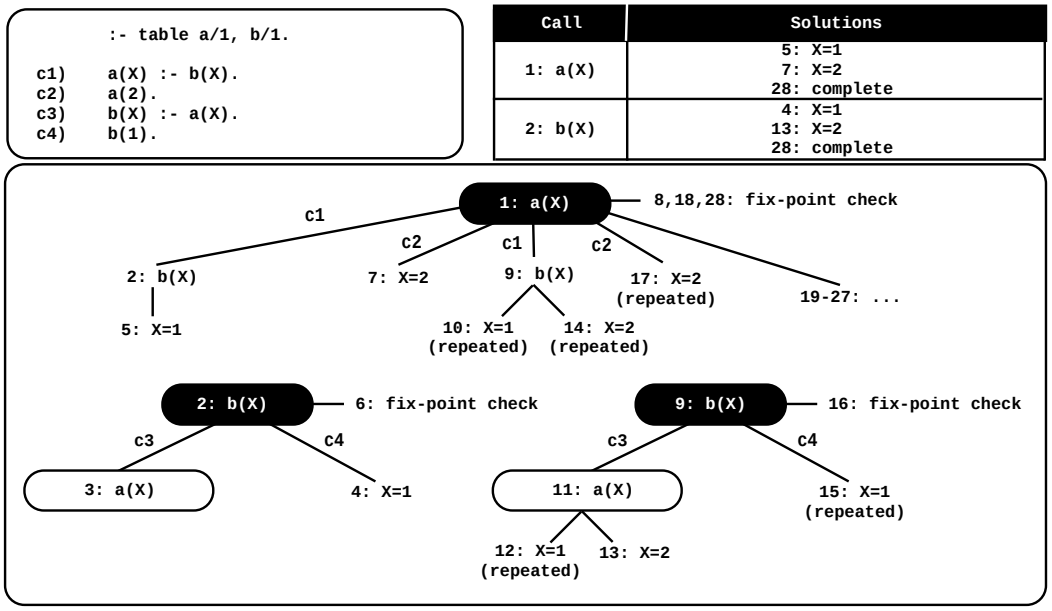


Figure 1 A standard linear tabled evaluation using batched scheduling.

The evaluation starts by inserting a new entry in the table space representing the generator call *a(X)* (step 1). Then, *a(X)* is resolved against its first matching clause, clause *c1*, calling *b(X)* in the continuation. As this is a first call to *b(X)*, we insert a new entry in the table space representing *b(X)* and proceed as shown in the bottom left tree (step 2). Subgoal *b(X)* is also resolved against its first matching clause, clause *c3*, calling again *a(X)* in the continuation (step 3). Since *a(X)* is a repeated call, we try to consume solutions from the table space, but at this stage no solutions are available, so execution fails.

We then try the second matching clause for *b(X)*, clause *c4*, and a first solution for *b(X)*, {*X=1*}, is found and added to the table space (step 4). We then follow a batched scheduling

strategy and the evaluation continues with *forward execution* [5]. With batched scheduling, new solutions are immediately returned to the calling environment, thus the solution for *b(X)* should now be propagated to the context of the previous call, which originates a first solution for *a(X)*, {*X=1*} (step 5). The execution then fails back to node 2 and we check for a fix-point (step 6), but *b(X)* is not a leader call because it has a dependency (consumer node 3) to an older call, *a(X)*. Remember that we reach a fix-point when no new solutions are found during the last round of trying the matching clauses for the leader call. Then, we try the second matching clause for *a(X)* and a second solution, {*X=2*}, is found and added to the table space (step 7). We then backtrack again to the generator call for *a(X)* and because we have already explored all matching clauses, we check for a fix-point (step 8). We have found new solutions for both *a(X)* and *b(X)* in this round, thus the current SCC is scheduled for re-evaluation.

The evaluation then repeats the same sequence as in steps 2 to 3 (now steps 9 to 11), but since we are following a batched scheduling strategy, we first consume the solutions already available for *b(X)* (this will be further explained later in section 4), which leads to a repeated solution for *a(X)* (step 10). Tabling does not store duplicate solutions in the table space. Instead, repeated solutions fail. This is how tabling avoids unnecessary computations, and even looping in some cases. Next, the evaluation jumps to the consumer call of *a(X)* (step 11). Solution {*X=1*} is first forwarded to it, which originates a repeated solution for *b(X)* (step 12) and thus execution fails. Then, solution {*X=2*} is also forward to it and a new solution for *b(X)* is found (step 13) and propagated to *a(X)*, which leads to a repeated solution for *a(X)* (step 14).

In the continuation, we find another repeated solution for *b(X)* (step 15) and we fail a second time in the fix-point check for *b(X)* (step 16). Again, as we are following a batched scheduling strategy, the solutions for *b(X)* were already all propagated to the context of *a(X)*, thus we can safely backtrack to the generator call for *a(X)*. Because we have found a new solution for *b(X)* during this last round, the current SCC is scheduled again for re-evaluation (step 18). The re-evaluation of the SCC does not find new solutions for both *a(X)* and *b(X)* (steps 19 to 27). Thus, when backtracking again to *a(X)* we have reached a fix-point and because *a(X)* is a leader call, we can declare the two subgoal calls to be completed (step 28).

3 Linear Tabling Strategies

The standard linear tabling mechanism uses a naive approach to evaluate tabled logic programs. Every time a new solution is found during the last round of evaluation, the complete search space for the current SCC is scheduled for re-evaluation. However, some branches of the SCC can be avoided, since it is possible to know beforehand that they will only lead to repeated computations, hence not finding any new solutions. Next, we present two different strategies for optimizing the standard linear tabled evaluation. The common goal of both strategies is to minimize the number of branches to be explored, thus reducing the search space, and each strategy tries to focus on different aspects of the evaluation to achieve it.

3.1 Dynamic Reordering of Alternatives

The key idea of the *Dynamic Reordering of Alternatives (DRA)* strategy, as originally proposed by Guo and Gupta [6], is to memoize the clauses (or alternatives) leading to consumer calls, the *looping alternatives*, in such a way that when scheduling an SCC for re-evaluation, instead of trying the full set of matching clauses, we only try the looping alternatives.

Initially, a generator call C explores the matching clauses as in standard linear tabled evaluation and, if a consumer call is found, the current clause for C is memoized as a looping alternative. After exploring all the matching clauses, C enters the *looping state* and from this point on, it only tries the looping alternatives until a fix-point is reached. Figure 2 uses the same program from Fig. 1 to illustrate how DRA evaluation works.

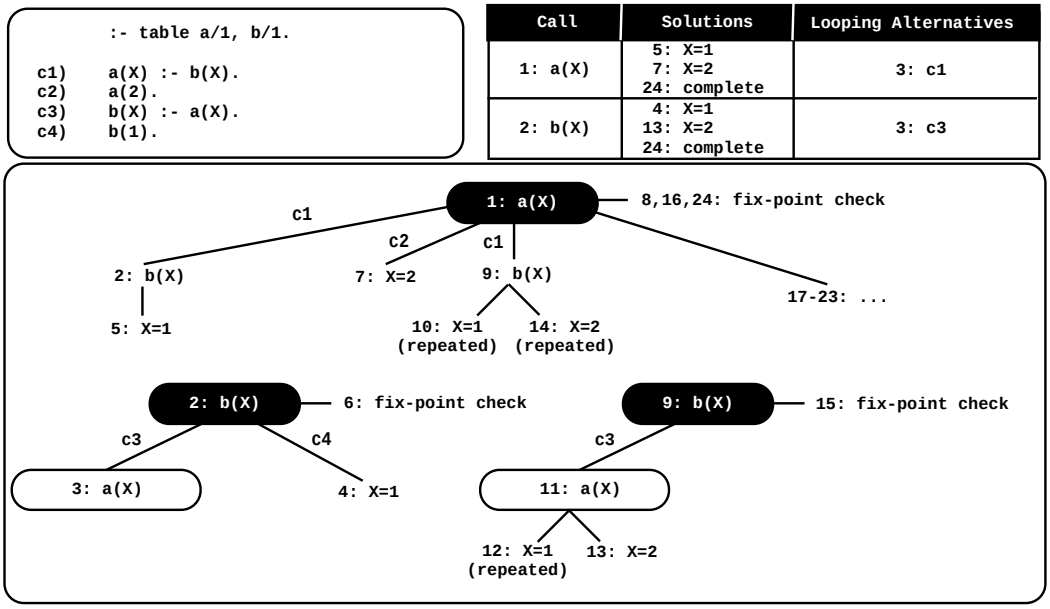


Figure 2 A linear tabled evaluation using batched scheduling with DRA evaluation.

The evaluation sequence for the first SCC round (steps 2 to 7) is identical to the standard evaluation of Fig. 1. The difference is that this round is also used to detect the alternatives leading to consumers calls. We only have one consumer call at node 3 for *a(X)*. The clauses in evaluation up to the corresponding generator, call *a(X)* at node 1, are thus marked as looping alternatives and added to the respective table entries. This includes alternative *c3* for *b(X)* and alternative *c1* for *a(X)*. As for the standard strategy, the SCC is then scheduled for two extra re-evaluation rounds (steps 9 to 15 and steps 17 to 23), but now only the looping alternatives are evaluated, which means that the clauses *c2* and *c4* are ignored.

3.2 Dynamic Reordering of Execution

The second strategy, that we call *Dynamic Reordering of Execution (DRE)*, is based on the original SLDT strategy, as proposed by Zhou et al. [14]. The key idea of the DRE strategy is to give priority to the program clauses and, for that, it lets repeated calls to tabled subgoals execute from the *backtracking clause of the former call*. A first call to a tabled subgoal is called a *pioneer* and repeated calls are called *followers* of the pioneer. When backtracking to a pioneer or a follower, we use the same strategy and we give priority to the exploitation of the remaining clauses. The fix-point check operation is still performed by pioneer calls. Figure 3 uses again the same program from Fig. 1 to illustrate how DRE evaluation works.

As for the standard strategy, the evaluation starts with (pioneer) calls to *a(X)* (step 1) and *b(X)* (step 2), and then, in the continuation, *a(X)* is called repeatedly (step 3). With DRE evaluation, *a(X)* is now considered a follower and thus we *steal* the backtracking clause

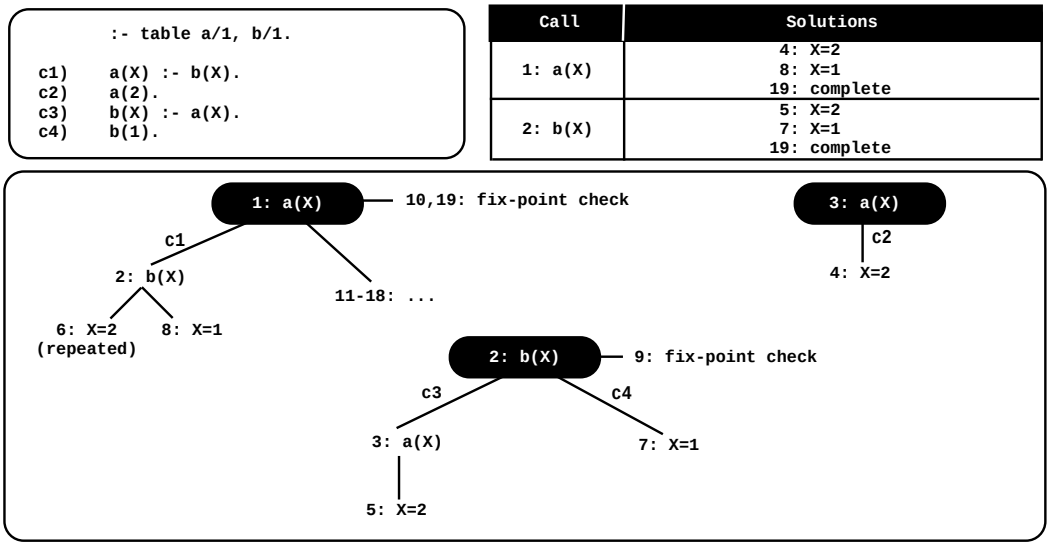


Figure 3 A linear tabled evaluation using batched scheduling with DRE evaluation.

of the former call at node 1, i.e., clause *c2*. The evaluation then proceeds as for a generator call (right upper tree in Fig. 3), which means that new solutions can be generated for *a(X)*. We thus try clause *c2* and a first solution for *a(X)*, $\{X{=}2\}$, is found and added to the table space (step 4). Then, we follow a batched scheduling strategy and the solution $\{X{=}2\}$ is propagated to the context of *b(X)*, which originates the solution $\{X{=}2\}$ (step 5), and to the context of *a(X)*, which leads to a repeated solution (step 6).

As both matching clauses for *a(X)* were already taken, the execution backtracks to the pioneer node 2. Next, we find a second solution for *b(X)* (step 7), which is then propagated, leading also to a second solution for *a(X)* (step 8). In step 9, we check for a fix-point, but *b(X)* is not a leader call because it has a dependency (follower node 3) to an older call, *a(X)*. We then backtrack to the pioneer call for *a(X)* and because we have already explored the matching clause *c2* in the follower node 3, we check for a fix-point. Since we have found new solutions during the last round, the current SCC is scheduled for re-evaluation (step 10). The re-evaluation of the SCC does not find any further solutions (steps 11 to 18), and thus the evaluation can be completed at step 19.

4 Propagation of Solutions in Re-evaluation Rounds

In the previous sections, one could observe that tabling does not store duplicate solutions in the table space and, instead, repeated solutions fail. This is how tabling avoids unnecessary computations, and even looping in some cases. However, since repeated solutions also fail in re-evaluation rounds, this means that, in fact, a solution is only propagated once, i.e., in the round it is first found, which might be not sufficient to ensure the completeness of the evaluation. To solve this problem, this is why, in a re-evaluation round, we start by propagating (consuming) the solutions already available for the subgoal call at hand. Alternatively, we could propagate the solutions at the end, after the fix-point check procedure, but by doing that some solutions will be propagated more than once in a single round, which is worthless.

In the previous examples, for simplicity of explanation, we have omitted some steps regarding the propagation of solutions since, for all the examples, one propagation per

solution was enough to correctly compute the corresponding evaluations. To better illustrate the importance of the propagation of solutions in re-evaluation rounds, Fig. 4 shows a new example, using again the same program from Fig. 1, but for the query goal *a(X), b(Y)*. For simplicity of explanation, we consider a standard linear tabled evaluation, i.e., without DRA and DRE support.

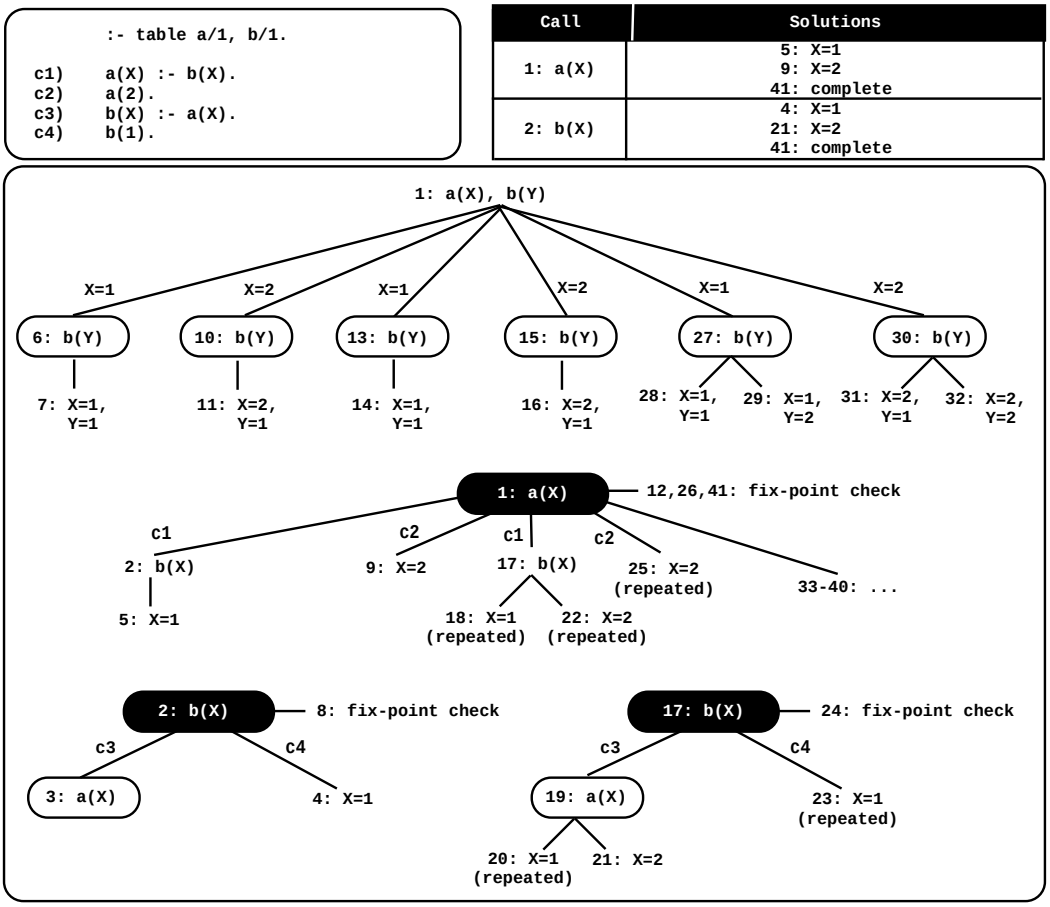


Figure 4 Propagation of solutions in re-evaluation rounds using batched scheduling.

In the first round of the evaluation (steps 1 to 12), the solutions found for *a(X)*, at steps 5 and 9, are propagated to the context of the top query goal and, in the continuation, *b(Y)* consumes (note that *b(Y)* is a variant call of *b(X)*) the available answer found for *b(X)* at step 4, which originates the solutions {*X=1, Y=1*} (step 7) and {*X=2, Y=1*} (step 11) for the top query goal.

Next, in the second round of the evaluation (steps 13 to 26), the generator node for *a(X)* starts by propagating its solutions (steps 13 to 16), but only repeated solutions are found for the top query goal (steps 14 and 16). Later, at step 21, a new solution, {*X=2*}, is found for *b(X)*. The combination of this new solution with the previous solutions for *a(X)* should originate new solutions for the top query goal. The solution for *b(X)* is then propagated to the context of *a(X)* (step 22) but, since this originates a repeated solution for *a(X)*, the computation fails.

By failing for *a(X)*, we cannot combine the new answer for *b(X)* with the previous answers for *a(X)* in this round. Hence, this fact, i.e., the fact that tabling fails for repeated

solutions, can lead to a collateral effect where it can be blocking forward execution. To solve this problem, this is why, in a re-evaluation round, we start by propagating all the available solutions. Therefore, in the third and final round of the evaluation (steps 27 to 40), the generator node for *a(X)* starts again by propagating its solutions (steps 28 to 32) and the solutions {*X=1, Y=2*} (step 29) and {*X=2, Y=2*} (step 32) are generated. Since this round does not find any new solution, the evaluation is finally completed at step 41.

5 Implementation Details

This section describes some implementation details regarding the extension of our framework to support batched scheduling, with particular focus on the table space data structures and on the tabling operations.

5.1 Table Space

To implement the table space, Yap uses *tries* which is considered a very efficient way to implement the table space [8]. Tries are trees in which common prefixes are represented only once. Tries provide complete discrimination for terms and permit look up and insertion to be done in a single pass. Figure 5 details the table space organization for the example used on the previous sections.

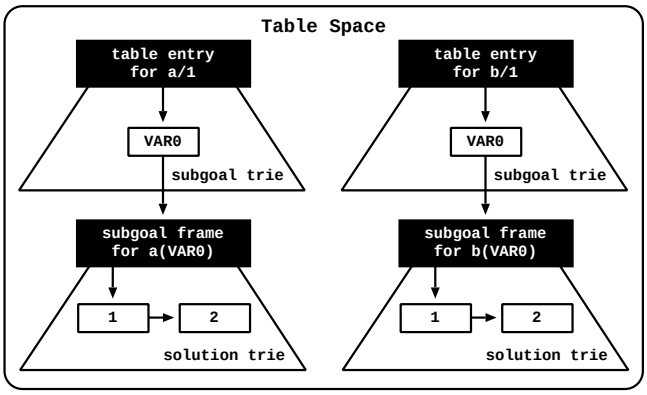


Figure 5 Table space organization.

As other tabling engines, Yap uses two levels of tries: one for the subgoal calls and other for the computed solutions. A tabled predicate accesses the table space through a specific *table entry* data structure. Each different subgoal call is represented as a unique path in the *subgoal trie* and each different solution is represented as a unique path in the *solution trie*. A key data structure in this organization is the *subgoal frame*. Subgoal frames are used to store information about each tabled subgoal call, namely: the entry point to the solution trie; the state of the subgoal (*ready*, *evaluating* or *complete*); support to detect if the subgoal is a leader call; and support to detect if new solutions were found during the last round of evaluation. The DRA and DRE strategies extend the subgoal frame data structure with the following extra information [2]:

DRA: support to detect, store and load looping alternatives; and two new states, *loop_ready* and *loop_evaluating*, used to detect, respectively, generator and consumer calls in re-evaluating rounds.

DRE: the pioneer call; and the backtracking clause of the former call.

To support the propagation of solutions, as discussed in section 4, an extra field, named *SgFr_last_consumed*, marks the last solution consumed in a pioneer or follower call.

5.2 Tabling Operations

We next introduce the pseudo-code for the main tabling operations required to support batched scheduling with DRA and DRE evaluation.

We start with Fig. 6 showing the pseudo-code for the *new solution* operation. Initially, the operation simply inserts the given solution *SOL* in the solution trie structure for the given subgoal frame *SF* and, if the solution is new, it updates the *SgFr_new_solutions* subgoal frame field to *TRUE*. Then, for batched scheduling, it adjusts the execution's environment and proceeds with forward execution.

```
new_solution(solution SOL, subgoal frame SF) {
  if (solution_check_insert(SOL,SF) == TRUE)                    // new solution
    SgFr_new_solutions(SF) = TRUE
  else                                                        // repeated solution
    fail()
  if (batched_scheduling_mode(SF))                           // batched scheduling
    proceed()
  else { ... }                                                 // local scheduling
}
```

Figure 6 Pseudo-code for the *new solution* operation.

Figure 7 shows the pseudo-code for the *tabled call* operation. New calls to tabled subgoals are inserted into the table space by allocating the necessary data structures. This includes allocating and initializing a new subgoal frame *SF* to represent the given subgoal call *SC* (this is the case where the state of *SF* is *ready*). In such case, the tabled call operation then stores a new generator node[2]; updates the state of *SF* to *evaluating*; and proceeds by executing the current alternative.

On the other hand, if the subgoal call is a repeated call, then the subgoal frame *SF* is already in the table space, and three different situations may occur. First, if the call is already evaluated (this is the case where the state of *SF* is *complete*), the operation consumes the available solutions by implementing the *completed table optimization* which executes compiled code directly from the solution trie structure associated with the completed call [8].

Second, if the call is a first call in a re-evaluating round (this is the case where the state of *SF* is *loop_ready*), the operation stores a new generator node; updates the state of *SF* to *loop_evaluating*; and resets the *SgFr_last_consumed* field to the first solution. Then, it executes the *consume_solutions_and_execute()* procedure in order to consume the available solutions before re-evaluate the matching alternatives. This procedure, consumes all the available solutions for the subgoal, starting from the first solution, and, when no more solutions are to be consumed, it starts with the evaluation of the first matching alternative, which for DRA is the first looping alternative.

Third, if the call is a repeated call (this is the case where the state of *SF* is *evaluating* or *loop_evaluating*), the operation first marks the current branch as a non-leader branch and, if in DRA, it also marks the current branch as a looping branch. Next, if DRE mode is enabled and there are unexploited alternatives (i.e., there is a backtracking clause for the

[2] Generator, consumer and follower nodes are implemented as regular WAM choice points extended with some extra fields related to the table space data structures.

```
tabled_call(subgoal call SC) {
  SF = subgoal_check_insert(SC)                    // SF is the subgoal frame for SC
  if (SgFr_state(SF) == ready) {                                    // first round
    store_generator_node()
    SgFr_state(SF) = evaluating
    goto execute(current_alternative())
  } else if (SgFr_state(SF) == complete) {                  // already evaluated
    goto completed_table_optimization(SF)
  } else if (SgFr_state(SF) == loop_ready) {              // re-evaluation round
    store_generator_node()
    SgFr_state(SF) = loop_evaluating
    if (batched_scheduling_mode(SF)){
      SgFr_last_consumed(SF) = SgFr_first_solution(SF)
      if (DRA_mode(SF))
        goto consume_solutions_and_execute(SF,first_looping_alternative())
      else
        goto consume_solutions_and_execute(SF,first_alternative())
    } else { ... }                                          // local scheduling
  } else if (SgFr_state(SF) == evaluating or                      // first round
             SgFr_state(SF) == loop_evaluating) {         // re-evaluation round
    mark_current_branch_as_a_non_leader_branch(SF)
    if (DRA_mode(SF))
      mark_current_branch_as_a_looping_branch(SF)
    if (DRE_mode(SF) && has_unexploited_alternatives(SF)) {
      store_follower_node()
      if (DRA_mode(SF) and SgFr_state(SF) == loop_evaluating)
        goto consume_solutions_and_execute(SF,next_looping_alternative())
      else
        goto consume_solutions_and_execute(SF,next_alternative())
    } else {
      store_consumer_node()
      goto consume_solutions(SF)
    }
  }
}
```

Figure 7 Pseudo-code for the *tabled call* operation.

former call), it stores a follower node and proceeds by consuming the available solutions before executing the next looping alternative or the next matching alternative, according to whether the DRA mode is enabled or disabled for the subgoal. Otherwise, it stores a new consumer node and starts consuming the available solutions.

To mark the current branch as a non-leader branch, we follow all intermediate generator calls in evaluation up to the generator call for frame *SF* and we mark them as non-leader calls (note that the call at hand defines a new dependency for the current SCC). To mark the current branch as a looping branch, we follow all intermediate generator calls in evaluation up to the generator call for frame *SF* and we mark the alternatives being evaluated by each call as looping alternatives.

Finally, we discuss in more detail how completion is detected with batched scheduling. Remember that after exploring the last matching clause for a tabled call, we execute the *fix-point check* operation. Figure 8 shows the pseudo-code for its implementation.

The fix-point check operation starts by verifying if the subgoal at hand is a leader call. If it is leader and has found new solutions during the last round, then the current SCC is scheduled for a re-evaluation. For batched scheduling, this is the same situation as for a first call in a re-evaluating round in the *tabled call* operation, i.e., it resets the *SgFr_last_consumed* field to the first solution and executes the *consume_solutions_and_execute()* procedure. If

```
fix-point_check(subgoal frame SF) {
  if (SgFr_is_leader(SF)){
    if (SgFr_new_solutions(SF)) {                                  // start a new round
      SgFr_new_solutions(SF) = FALSE
      for each (subgoal in current SCC)
        SgFr_state(subgoal) = loop_ready
      SgFr_state(SF) = loop_evaluating
      if (batched_scheduling_mode(SF)) {
        SgFr_last_consumed(SF) = SgFr_first_solution(SF)
        if (DRA_mode(SF))
          goto consume_solutions_and_execute(SF,first_looping_alternative())
        else
          goto consume_solutions_and_execute(SF,first_alternative())
      } else { ... }                                                // local scheduling
    } else {                                        // complete subgoals in current SCC
      for each (subgoal in current SCC)
        SgFr_state(subgoal) = complete
      if (batched_scheduling_mode(SF))
        fail()
      else { ... }                                                  // local scheduling
  } else {                                                         // not a leader call
    if (SgFr_new_solutions(SF))                                 // propagate new solutions
      SgFr_new_solutions(current_leader(SF)) = TRUE
    SgFr_new_solutions(SF) = FALSE                                 // reset new solutions
    if (batched_scheduling_mode(SF))
      fail()
    else { ... }                                                    // local scheduling
  }
}
```

Figure 8 Pseudo-code for the *fix-point check* operation.

the subgoal is leader but no new solutions were found during the current round, then we have reached a fix-point and thus, the subgoals in the current SCC are marked as completed and the evaluation fails.

Otherwise, if the subgoal is not a leader call, then it propagates the new solutions information to the current leader of the SCC and the evaluation fails. Note that, with batched scheduling, we can safely fail since all the solutions were already propagated to the context of the calling environment. Moreover, since the *SgFr_new_solutions* flag is propagated to the leader of the SCC, the leader will mark the SCC for a new evaluation round, which means that the current subgoal will be called again, and so it will start by consuming all its solutions.

6 Experimental Results

To the best of our knowledge, Yap is now the first tabling engine that integrates and supports the combination of different linear tabling strategies using batched scheduling. We have thus the conditions to better understand the advantages and weaknesses of each strategy when used solely or combined. In what follows, we present initial experiments comparing linear tabled evaluation with and without DRA and DRE support, using batched scheduling. To put our results in perspective, we have also included experiments for the B-Prolog linear tabling system [13] and for the YapTab suspension-based tabling system [10], both using batched scheduling. In fact, for B-Prolog, we used its *eager scheduling mode*, which is similar to batched scheduling. The environment for our experiments was a PC with a 2.83 GHz Intel(R) Core(TM)2 Quad CPU and 8 GBytes of memory running the Linux kernel

3.0.0-16-generic. We used B-Prolog version 7.5 and Yap version 6.0.7.

For benchmarking, we used three sets of programs. The **Model Checking** set includes three different specifications and transition relation graphs usually used in model checking applications. The **Path Right** set implements the right recursive definition of the well-known *path*/2 predicate, that computes the transitive closure in a graph, using three different edge configurations. Figure 9 shows an example for each configuration. We experimented the **Pyramid** and **Cycle** configurations with depths 1000, 2000 and 3000 and the **Grid** configuration with depths 20, 30 and 40. We chose this set of experiments because the *path*/2 predicate implements a relatively easy to understand pattern of computation and its right recursive definition creates several inter-dependencies between tabled subgoals. The **Warren** set is a variation of the left recursive definition of the path problem for a linear graph (see Fig. 9), where the *path/2* clauses are duplicated to be used with the labels *a* and *b*. This problem was kindly suggested by David S. Warren as a way to stress the performance of a linear tabling system. All benchmarks find all the solutions for the problem.

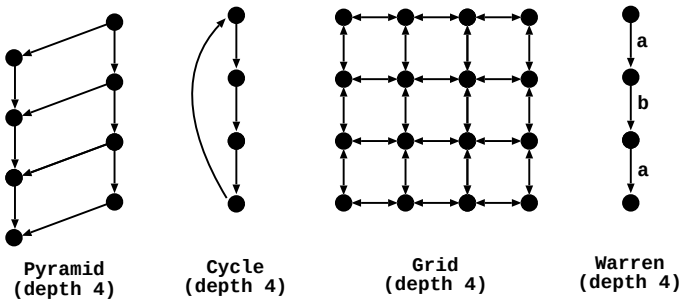


Figure 9 Edge configurations used with the second and third set of problems.

In Table 1, we show the execution time, in milliseconds, for standard linear tabling (column **Std**), DRA and DRE evaluation, solely and combined (column **DRA+DRE**), B-Prolog and YapTab, using batched scheduling, and the respective performance ratios when compared with standard linear tabling, for the **Model Checking**, **Path Right** and **Warren** sets of problems. Ratios higher than 1.00 mean that the respective strategy has a positive impact on the execution time. The ratio marked with *n.c.* for B-Prolog means that we are *not considering* it in the average results (for some reason, we failed in executing this benchmark). The results are the average of five runs for each benchmark.

In addition to the results presented in Table 1, we also collected several statistics regarding important aspects of the evaluation (not fully presented here due to lack of space). In Table 2, we show some of these statistics for standard linear tabling and the respective performance ratios when compared with the other models, for a subset of the benchmarks. We used the **Leader** specification for the **Model Checking** set, the configurations **Pyramid** and **Cycle** with depth 2000 and **Grid** with depth 30 for the **Path Right** set, and the configuration with depth 600 for the **Warren** set.

The statistics in Table 2 measure how the mixing with SLD (non-tabled) computations can affect the base performance of our benchmarks. For that, we extended the tabled predicates, at the beginning and at the end of each clause, with dummy SLD (non-tabled) predicates, which we named *sldi/0*, with $0 < i \leq 2n$, where n is the number of tabled clauses. For example, the extended definition for the *path/2* predicate is:

```
path(X,Z) :- sld1, edge(X,Y), path(Y,Z), sld2.
path(X,Z) :- sld3, edge(X,Z), sld4.
```

Table 1 Execution time, in milliseconds, for standard linear tabling, DRA and DRE evaluation, solely and combined, B-Prolog and YapTab, using batched scheduling, and the respective ratios when compared with standard linear tabling (for the linear tabling models, best ratios are in bold).

Bench	Std	DRA		DRE		DRA+DRE		B-Prolog		YapTab	
Model Checking											
IProto	2,874	2,879	**(1.00)**	5,766	(0.50)	3,098	(0.93)	8,087	(0.36)	1,201	(2.39)
Leader	5,355	5,288	**(1.01)**	13,423	(0.40)	5,430	(0.99)	40,624	(0.13)	1,891	(2.83)
Sieve	35,218	35,350	**(1.00)**	76,927	(0.46)	38,048	(0.93)	217,155	(0.16)	11,046	(3.19)
	Average		(1.00)		(0.45)		(0.95)		(0.22)		(2.80)
Path Right - Pyramid											
1000	983	526	**(1.87)**	1,102	(0.89)	658	(1.49)	948	(1.04)	517	(1.90)
2000	3,897	2,071	**(1.88)**	4,380	(0.89)	2,611	(1.49)	5,630	(0.69)	2,013	(1.94)
3000	9,043	4,740	**(1.91)**	10,110	(0.89)	5,920	(1.53)	—	(*n.c.*)	4,561	(1.98)
Path Right - Cycle											
1000	687	539	**(1.27)**	713	(0.96)	563	(1.22)	540	**(1.27)**	362	(1.89)
2000	2,793	2,198	**(1.27)**	2,891	(0.97)	2,286	(1.22)	3,079	(0.91)	1,534	(1.82)
3000	6,048	4,681	**(1.29)**	6,343	(0.95)	4,949	(1.22)	8,678	(0.70)	2,956	(2.05)
Path Right - Grid											
20	221	166	**(1.33)**	227	(0.97)	174	(1.27)	202	(1.09)	105	(2.10)
30	1,344	1,015	**(1.32)**	1,362	(0.99)	1,036	(1.30)	1,318	(1.02)	605	(2.22)
40	4,578	3,508	**(1.31)**	4,697	(0.97)	3,630	(1.26)	5,995	(0.76)	1,958	(2.34)
	Average		**(1.50)**		(0.94)		(1.33)		(0.93)		(2.03)
Warren											
400	2,673	2,632	(1.02)	42	**(64.26)**	42	**(64.26)**	7,861	(0.34)	21	(126.09)
600	9,496	9,564	(0.99)	109	**(87.28)**	109	**(87.28)**	27,302	(0.35)	58	(162.61)
800	23,163	23,086	(1.00)	205	(112.88)	198	**(116.98)**	67,049	(0.35)	107	(216.88)
	Average		(1.00)		**(87.93)**		**(89.51)**		(0.35)		(168.53)

The rows in Table 2 show the number of times each dummy SLD predicate is called for the corresponding benchmark. We can read these numbers as an estimation of the performance ratios that we will obtain if the execution time of the corresponding SLD predicate clearly overweights the execution time of the other computations. Note that the odd SLD predicates (such as **sld1** and **sld3**) correspond to re-executions of a clause and that the even SLD predicates (such as **sld2** and **sld4**) correspond to new answer operations. In our experiments, the **sld2** predicate (placed at the end of the first tabled clause) is the one that can potentially have a greater influence in the performance ratios as it clearly exceeds all the others in the number of times it is called (see Table 2).

Analyzing the general picture of Table 1, the results show that DRA evaluation is able to reduce the execution time for the **Path Right** problem set (1.50 times faster, on average) but has no impact for the other two sets, when compared with standard evaluation. The results also indicate that DRE evaluation has a negative impact in the execution time for the **Model Checking** and **Path Right** sets but, on the other hand, it can significantly reduce the execution time for the **Warren** set (more than 80 times faster, on average). We next discuss in more detail each strategy.

Table 2 Number of calls to the dummy SLD predicates for standard linear tabling and the respective ratios when compared with DRA and DRE evaluation, solely and combined, B-Prolog and YapTab, using batched scheduling (for the linear tabling models, best ratios are in bold).

Bench	Std	DRA	DRE	DRA+DRE	B-Prolog	YapTab
Model Checking - Leader						
sld1	3	1.00	0.75	1.00	1.00	3.00
sld2	1,153,026	1.00	0.40	1.00	1.00	2.00
sld3	3	**3.00**	0.75	**3.00**	**3.00**	3.00
sld4	3	**3.00**	0.75	**3.00**	**3.00**	3.00
Path Right - Pyramid 2000						
sld1	7,999	**2.00**	1.00	**2.00**	**2.00**	2.00
sld2	37,951,017	**2.38**	0.86	1.73	**2.38**	2.38
sld3	7,999	**2.00**	1.00	**2.00**	**2.00**	2.00
sld4	23,988	**2.00**	1.00	**2.00**	**2.00**	2.00
Path Right - Cycle 2000						
sld1	6,002	1.00	1.00	1.00	1.00	3.00
sld2	18,003,000	**1.29**	1.00	**1.29**	**1.29**	2.25
sld3	6,002	**3.00**	1.00	**3.00**	**3.00**	3.00
sld4	10,000	**2.50**	1.00	**2.50**	**2.50**	2.50
Path Right - Grid 30						
sld1	2,702	1.00	1.00	1.00	0.18	3.00
sld2	13,851,534	1.29	1.00	**1.30**	0.30	2.21
sld3	2,702	**3.00**	1.00	1.02	**3.00**	3.00
sld4	17,400	**2.50**	1.00	1.27	**2.50**	2.50
Warren - 600						
sld1/sld3	302	1.00	**100.67**	**100.67**	1.00	302.00
sld2/sld4	18,044,650	1.00	66.98	**100.42**	1.00	201.17
sld5/sld7	302	**302.00**	100.67	**302.00**	**302.00**	302.00
sld6/sld8	90,600	**302.00**	100.67	**302.00**	**302.00**	302.00

DRA: the results for DRA evaluation show that the strategy of avoiding the exploration of non-looping alternatives in re-evaluation rounds is quite effective in general and does not add extra overheads when not used. The results also show that, for the **Path Right** set, DRA is more effective for programs without loops, like the **Pyramid** configurations, than for programs with larger SCCs, like the **Cycle** and **Grid** configurations. On Table 2, we can observe that the number of dummy SLD computations is, in fact, effectively reduced with DRA evaluation.

DRE: for the **Model Checking** set, DRE evaluation is around two times slower than standard evaluation and, for the **Path Right** set, DRE has no significant impact for all the configurations. Table 2 confirms that, the strategy of allocating follower nodes, adds an extra complexity to the evaluation for the **Model Checking** set (the number of dummy SLD calls is higher) and that it has no impact for the **Path Right** set (the number of dummy SLD calls is identical to standard evaluation). For the **Warren** set, DRE evaluation produces the most interesting results. Note that, this is the set of benchmarks where suspension-based tabling (the YapTab system) is far more faster than standard linear tabling (168.53 times faster, on average) and the difference increases as the depth of the problem also increases. However, DRE evaluation is able to reduce this

huge difference to a minimum. On average, DRE evaluation is 87.93 times faster than standard evaluation and the scalability, as the depth of the problem increases, is similar to the one observed for YapTab. Table 2 confirms this behavior for DRE and YapTab evaluations (the number of dummy SLD calls is clearly lower than standard evaluation).

Regarding the combination of both strategies (DRA+DRE), our experiments show that, in general, the best of both worlds is always present in the combination. The results in Table 1 show that, by combining both strategies, DRA is able to avoid DRE behavior for the **Model Checking** and **Path Right** sets. Still, the results for DRA+DRE are slightly worst than DRA used solely. For the **Warren** set, the results show that, by combining both strategies, it is possible to reduce even further the execution time when compared with DRE used solely. In particular, one can observe that, for depths 400 and 600, the execution times are the same but, for depth 800, DRA+DRE evaluation outperforms DRE used solely.

The statistics on Table 2 confirm that, in general, the best of both worlds is always present in the combination. The exceptions are the **sld2** predicate, for the **Pyramid 2000** configuration, and the **sld3** and **sld4** predicates, for the **Grid 30** configuration. On the other hand, for the **Warren 600** configuration, the **sld1/sld3** predicates are executed the same number of times as for DRE used solely, the **sld5** to **sld8** predicates are executed the same number of times as for DRA used solely, and the **sld2** and **sld4** predicates are executed less times than both strategies used solely, which is explained by the fact that the fix-point is achieved in less rounds (statistics not shown here due to lack of space).

Regarding the comparison with the B-Prolog linear tabling system, the results in Table 2 suggest that B-Prolog implements a DRA-based evaluation strategy since the statistics for B-Prolog and DRA evaluation are all the same, except for the **sld1** and **sld2** predicates in the **Grid 30** configuration. However, the execution times in Table 1 show that our DRA implementation is always faster than B-Prolog in these experiments and that, for almost all configurations, the ratio difference shows a generic tendency to increase as the depth of the problem also increases.

For all experiments, the results obtained for the YapTab suspension-based system clearly outperform the standard linear tabled evaluation but, for our DRA+DRE implementation, they are globally comparable. On average, YapTab is around 2 times faster than DRA+DRE evaluation, including the **Warren** problem set, where YapTab shows a huge difference for standard linear tabling. The results also indicate that our implementation scales as well as YapTab when we increase the depth of the problem being tested.

7 Conclusions

We have presented a new linear tabling framework that integrates and supports batched scheduling with DRA and DRE evaluation, solely or combined. We discussed how these strategies can optimize different aspects of a tabled evaluation and we presented the relevant implementation details for their integration on top of the Yap system.

Our experimental results were very interesting and very promising. In particular, the combination of DRA with DRE showed the potential of our framework to effectively reduce the execution time of the standard linear tabled evaluation. When compared with YapTab's suspension-based mechanism, the commonly referred weakness of linear tabling of doing a huge number of redundant computations for computing fix-points was not such a problem in our experiments. We thus argue that an efficient implementation of linear tabling can be a good and first alternative to incorporate tabling into a Prolog system without such support.

Further work will include adding new strategies/optimizations to our framework, and exploring the impact of applying our strategies to more complex problems, seeking real-world experimental results, allowing us to improve and consolidate our current implementation.

Acknowledgements This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO/112158/2009).

References

1 M. Areias and R. Rocha. An Efficient Implementation of Linear Tabling Based on Dynamic Reordering of Alternatives. In *International Symposium on Practical Aspects of Declarative Languages*, number 5937 in LNCS, pages 279–293. Springer-Verlag, 2010.

2 M. Areias and R. Rocha. On Combining Linear-Based Strategies for Tabled Evaluation of Logic Programs. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 11(4–5):681–696, 2011.

3 W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

4 P. Chico, M. Carro, M. V. Hermenegildo, C. Silva, and R. Rocha. An Improved Continuation Call-Based Implementation of Tabling. In *International Symposium on Practical Aspects of Declarative Languages*, number 4902 in LNCS, pages 197–213. Springer-Verlag, 2008.

5 J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in LNCS, pages 243–258. Springer-Verlag, 1996.

6 Hai-Feng Guo and G. Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 181–196. Springer-Verlag, 2001.

7 J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

8 I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

9 K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

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

11 Z. Somogyi and K. Sagonas. Tabling in Mercury: Design and Implementation. In *International Symposium on Practical Aspects of Declarative Languages*, number 3819 in LNCS, pages 150–167. Springer-Verlag, 2006.

12 T. Swift and D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1 & 2):157–187, 2012.

13 Neng-Fa Zhou. The Language Features and Architecture of B-Prolog. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):189–218, 2012.

14 Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a Linear Tabling Mechanism. In *Practical Aspects of Declarative Languages*, number 1753 in LNCS, pages 109–123. Springer-Verlag, 2000.

Mode-Directed Tabling and Applications in the YapTab System

João Santos and Ricardo Rocha

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

Abstract

Tabling is an implementation technique that solves some limitations of Prolog's operational semantics in dealing with recursion and redundant sub-computations. Tabling works by memorizing generated answers and then by reusing them on similar calls that appear during the resolution process. In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. Traditional tabling systems are thus very good for problems that require finding all answers. Mode-directed tabling is an extension to the tabling technique that supports the definition of selective criteria for specifying how answers are inserted into the table space. Implementations of mode-directed tabling are already available in systems like ALS-Prolog, B-Prolog and XSB. In this paper, we propose a more general approach to the declaration and use of mode-directed tabling, implemented on top of the YapTab tabling system, and we show applications of our approach to problems involving Justification, Preferences and Answer Subsumption.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Tabling, Mode Operators, Applications

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.25

1 Introduction

Logic programming languages, such as Prolog, provide a high-level, declarative approach to programming. The operational semantics of Prolog is given by SLD resolution, an evaluation strategy particularly simple that matches stack based machines particularly well, but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. Tabling [1] is a recognized and powerful implementation technique that solves such limitations in a very elegant way. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property*[1]. Work on tabling, as initially implemented in the XSB system [11], proved its viability for application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, Program Analysis, among others. Currently, tabling is widely available in systems like XSB, Yap, B-Prolog, ALS-Prolog, Mercury and Ciao.

In a nutshell, tabling consists of saving and reusing the results of sub-computations during the execution of a program and, for that, the calls and the answers to tabled subgoals are stored in a proper data structure called the *table space*. The tabling technique can be viewed as a natural tool to implement dynamic programming algorithms. Dynamic programming

[1] A logic program has the bounded term-size property if there is a function $f : N \rightarrow N$ such that whenever a query goal Q has no argument whose term size exceeds n, then no term in the derivation of Q has size greater than $f(n)$.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 25–40

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

is a general recursive strategy that consists in dividing a problem in simple sub-problems that, after solved, will constitute the final solution for the main problem. Often, many of these sub-problems are really the same. The dynamic programming approach seeks to solve each sub-problem only once, hence reducing the number of computations. Tabling is thus suitable to use with this kind of problems since, by storing and reusing intermediate results while the program is executing, it avoids calculating the same computation several times.

In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. When a new answer is not a variant[2] of any answer that is already in the table space, then it is always considered for insertion. Therefore, traditional tabling systems are very good for problems that require finding all answers. However, with dynamic programming, usually, the goal is to dynamically calculate optimal or selective answers as new results arrive. Writing dynamic programming algorithms can thus be a difficult task without further support. *Mode-directed tabling* [4] is an extension to the tabling technique that supports the definition of selective criteria for specifying how answers are inserted into the table space. The idea of mode-directed tabling is to use *mode operators* to define what arguments should be used in variant checking in order to select what answers should be tabled. The features added by these operators can then be elegantly applied, not only to dynamic programming problems, but also to problems related to Machine Learning [15], Justification [10, 7], Preferences [2, 5, 6], Answer Subsumption [8], among others.

In a traditional tabling system, to evaluate a predicate using tabling, we just need to declare it as '*table* p/n', where p is the predicate name and n its arity. With mode-directed tabling, tabled predicates are declared using statements of the form '*table* $p(m_1, ..., m_n)$', where p is the tabled predicate and the m_i's are the mode operators for the arguments. Implementations of mode-directed tabling are already available in systems like ALS-Prolog [4], B-Prolog [15] and XSB [14]. In this paper, we propose a more general approach to the declaration and use of mode operators, implemented on top of the YapTab tabling system [9], and we show applications of our approach to problems involving Justification, Preferences and Answer Subsumption.

The remainder of the paper is organized as follows. First, we introduce some background concepts about tabling. Next, we describe the mode operators that we propose and we show small examples of their use. Then, we address the use of such operators in three application areas. We start by discussing how these operators can be used in the generation of justifications for the answers of a program. Next, we discuss the set of transformations that allow preference problems to be implemented using mode operators and, last, we discuss the implementation of an answer subsumption mechanism using mode operators.

2 Tabled Evaluation

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Similar calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all similar calls. Within this model, the nodes in the search space are classified as either: *generator*

[2] Two terms are considered to be variant if they are the same up to variable renaming.

nodes, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to similar calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

Figure 1 illustrates the execution of a tabled program. The top left corner of the figure shows the program code and the top right corner shows the final state of the table space. The program defines a small directed graph, represented by two *edge/2* facts, with a relation of reachability, given by a *path/2* predicate, that computes the transitive closure in a graph. The declaration *:- table path/2* specifies that *path/2* should be evaluated using tabling.

The bottom of the figure shows the evaluation sequence for the query goal *path(a,Z)*. Note that traditional Prolog would immediately enter an infinite loop because the first clause of *path/2* leads to a variant call to *path(a,Z)*. In contrast, if tabling is applied then termination is ensured. Next, we describe in more detail how this query is solved with tabled evaluation.

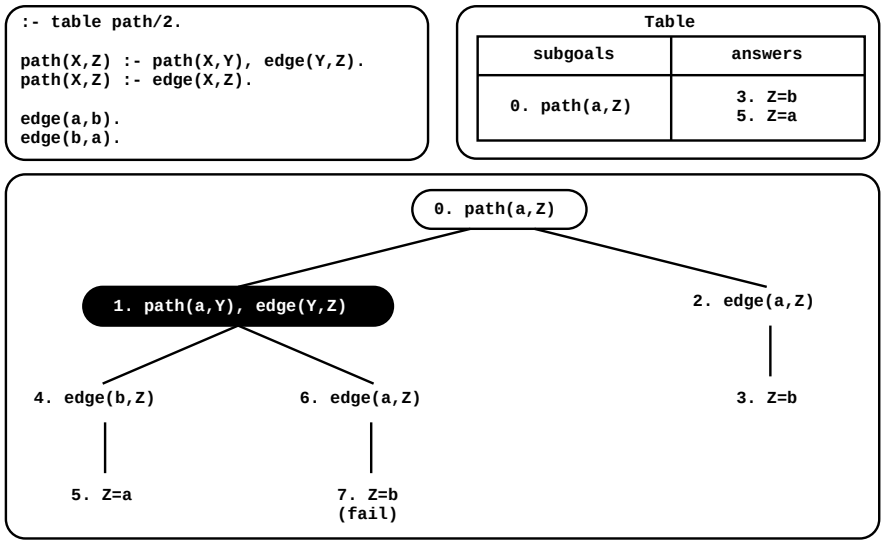


Figure 1 An example of a tabled evaluation.

The execution starts with the subgoal call *path(a,Z)*. First calls to tabled subgoals correspond to generator nodes (generator nodes are depicted by white oval boxes) and, for first calls, a new entry, representing the subgoal, is added to the table space (step 0). Next, *path(a,Z)* is resolved against the first matching clause in the program calling, in the continuation, *path(a,Y)* (step 1). Since *path(a,Y)* is a variant call to *path(a,Z)*, we do not evaluate the subgoal against the program clauses, instead we consume answers from the table space. Such nodes are called *consumer nodes* (consumer nodes are depicted by black oval boxes). However, at this point, the table does not have answers for this call, so the computation is suspended[3]

The only possible move after suspending is to backtrack and try the second matching clause for *path(a,Z)* (step 2). This originates the answer $\{Z{=}b\}$, which is then stored in the table space (step 3). At this point, the computation at node 1 can be resumed with the newly found answer (step 4), giving rise to one more answer, $\{Z{=}a\}$ (step 5). This second

[3] For the sake of simplicity, we are assuming a *suspension-based tabling* mechanism [11], where a tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. Alternatively, *linear tabling* mechanisms [16, 3] use iterative computations to compute fix-points and for that they maintain a single execution tree without requiring suspension and resumption of sub-computations.

answer is then also inserted in the table space and propagated to the consumer node (step 6), which originates the answer {*Z=b*} (step 7). This answer had already been found at step 3. Tabling does not store duplicate answers in the table space. Instead, repeated answers *fail*. This is how tabling avoids unnecessary computations, and even looping in some cases. A new answer is inserted in table space only if it is not a variant of any answer that is already there. Since there are no more answers to consume nor more clauses left to try, the evaluation ends and the table entry for *path(a,Z)* can be marked as *completed*.

3 Mode Operators

This section describes the mode operators that we are proposing and how they can be used in our implementation done on top of YapTab [9], a tabling system that extends Yap's engine [12] to support tabled evaluation for definite programs.

As mentioned before, in a traditional tabling system, to evaluate a predicate using tabling, we just write at the top of the program '*table p/n*'. With mode-directed tabling [4], tabled predicates are declared using statements of the form '*table* $p(m_1, ..., m_n)$', where the m_i's are mode operators for the arguments. We have defined 6 different mode operators: *index*, *min*, *max*, *first*, *last* and *all*. Arguments with modes *min*, *max*, *first*, *last* or *all* are assumed to be output arguments and only *index* arguments are considered for variant checking. After an answer be generated, the system tables the answer only if it is *better*, accordingly to the meaning of the output arguments, than some existing variant answer. Next, we describe in more detail how these mode operators work and we show some small examples of their use in the YapTab system.

3.1 *First* Mode Operator

Starting from the example in Fig. 1, consider now that we modify the program so that it also calculates the number of edges that are traversed in a path. Figure 2 illustrates the execution of this new program. As we can see, even with tabling, the program enters an infinite loop. Such behavior occurs because there is a path with an infinite number of edges starting from *a*, thus not verifying the bounded term-size property necessary to ensure termination. In particular, the answers found in steps 3 and 7 and in steps 5 and 9 have the same answer for variable *Z* ({*Z=b*} and {*Z=a*}, respectively), but they are both inserted in the table space because they are not variants for variable *N*. For programs with an infinite number of answers, traditional tabling is thus not enough, since we may need to specify a selective criteria to restrict the number of answers to a finite set.

As we will see next, by using tabling with mode operators, termination can be ensured for programs with an infinite number of answers. The example in Fig. 2 can be solved by using the *index* and *first* mode operators. As already mentioned, the mode *index* means that only the given arguments must be considered for variant checking. The mode *first* means that only the first answer for those arguments must be stored.

Knowing that the problem with the program in Fig. 2 resides on the fact that the third argument generates an infinite number of answers, we can thus define this argument to have mode *first* and the others to have mode *index*. Figure 3 illustrates this modification and the new execution tree and table space. If we compare both programs, the only difference is the declaration of the *path/3* predicate that is now *path(index,index,first)*. By observing Fig. 3, we can see that the answer {*Z=b, N=3*} (step 7) is no longer inserted in the table. That happens because, with the *first* mode on the third argument, the answer {*Z=b, N=1*} found at step 3 is considered a variant of the answer {*Z=b, N=3*} found at step 7.

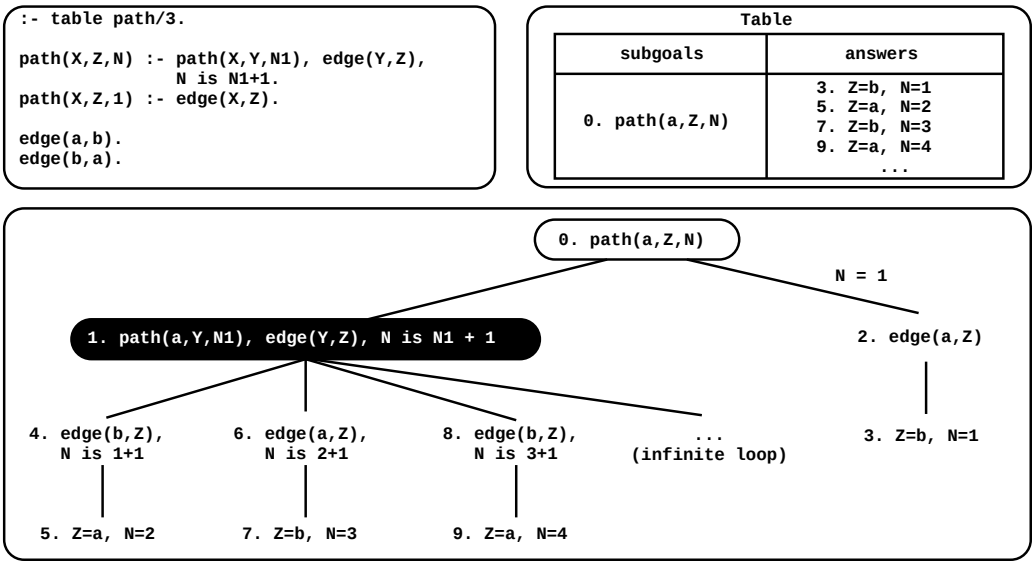


Figure 2 A tabled evaluation with an infinite number of answers.

3.2 *Min*/*Max* Mode Operators

The mode operator *first* only allows to store the first answer found for the respective argument. It would be interesting if we can also define other criteria to specify which answers we would like to store. For that we introduce two new mode operators, *min* and *max*, that allow to store, respectively, the minimal and the maximal answers found for an argument. To better understand the effect of these operators, let us consider the example in Fig. 4.

The program defines again a small directed graph with a relation of reachability, but now the edge facts include a third argument with a cost. The program's goal is to compute the paths with the lowest costs. To do that, the *path/3* predicate is declared as *path(index,index,min)*, meaning that the third argument should store only the minimal answers related with the first two arguments.

By observing the example in Fig. 4, we can see that the execution tree follows the normal evaluation of a tabled program and that the answers are stored as they are found. The most interesting part happens at step 8, where the answer $\{Z{=}d,\ C{=}3\}$ is found. This answer is a variant of the answer $\{Z{=}d,\ C{=}5\}$ found at step 6. In the previous example, with the *first* operator, the old answer would have been kept in the table. Here, as the new answer is minimal on the third argument, the old answer is replaced by the new answer.

The *max* mode operator works similarly to the *min* mode operator, but stores the maximal answer instead. In any case, we must be careful when using these two mode operators as they may not ensure termination for programs without the bounded term-size property. For instance, this would be the case if, in the example of Fig.4, we used the *max* mode instead of the *min* mode operator.

3.3 *All* Mode Operator

Another mode operator that can be useful is the *all* operator, that allows us to store all the answers for a given argument. Consider, for example, the program in Fig. 5, that is a mix of the programs in Fig. 3 and Fig. 4. In this new program, the path predicate is declared as *path(index,index,min,all)* meaning that, for each path, we want to store the lowest cost of

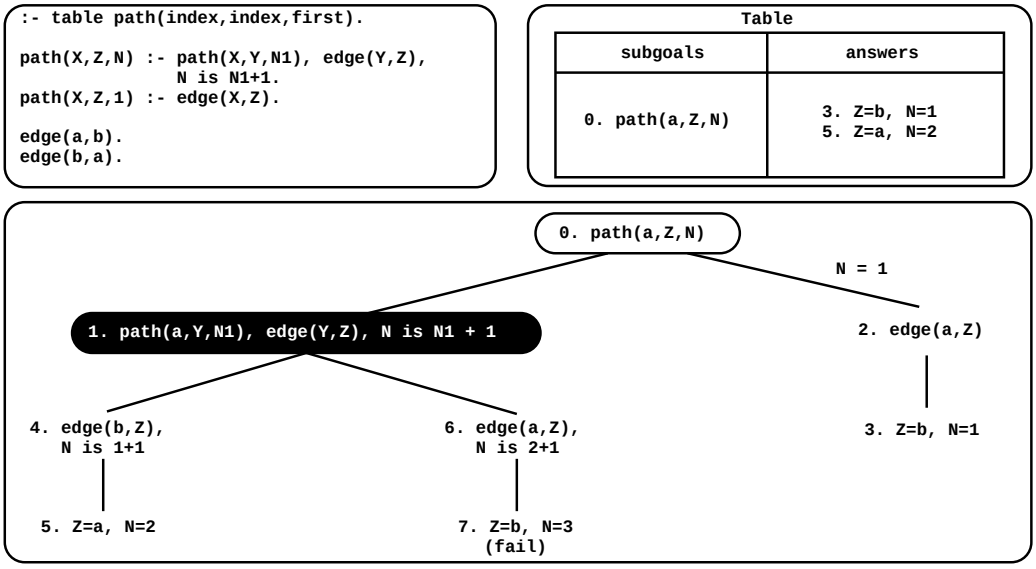


Figure 3 Using tabling with the mode operator *first*.

the path (the third argument) and, at the same time, we want to store the number of edges traversed, for all paths with minimal cost (the fourth argument).

The execution tree for the program in Fig. 5 is similar to the previous ones. The most interesting part happens when the answer {*Z=b, C=2, N=2*} is found at step 8. This answer is a variant of the answer {*Z=b, C=2, N=1*} found at step 3 and although both have the same minimal value (*C=2*), the new answer is still inserted in the table space since the number of edges (fourth argument) are different.

Notice that when the *all* operator is used in conjunction with another mode operator, like the *min* operator in the example, it is important to keep in mind that the aggregation of answers made for the *all* argument depends on the corresponding answer for the *min* argument. Consider, for example, that in the previous example we had found one more answer {*Z=b, C=1, N=4*}. In this case, the new answer would be inserted and the answers {*Z=b, C=2, N=1*} and {*Z=b, C=2, N=2*} would be deleted because the new answer corresponds to a shorter path, as defined by the value *C=1* in the *min* argument.

3.4 *Last* Mode Operator

Finally, we introduce the *last* mode operator. The *last* operator implements the opposite behaviour of the *first* operator. In other words, it always stores the last answer being found and deletes the previous one, if any. As we will see in the next sections, this operator is very useful for implementing problems involving Preferences and Answer Subsumption.

3.5 Related Work

The ALS-Prolog [4] and B-Prolog [15] systems also implement mode-directed tabling using a very similar syntax. Some operators, however, have different names in those systems. For example, the operators *index*, *first* and *all* are known as *+*, - and *@*, respectively. B-Prolog has an extra operator, named *nt*, to indicate that a given argument should not be tabled and, thus, not considered to be inserted in the table space.

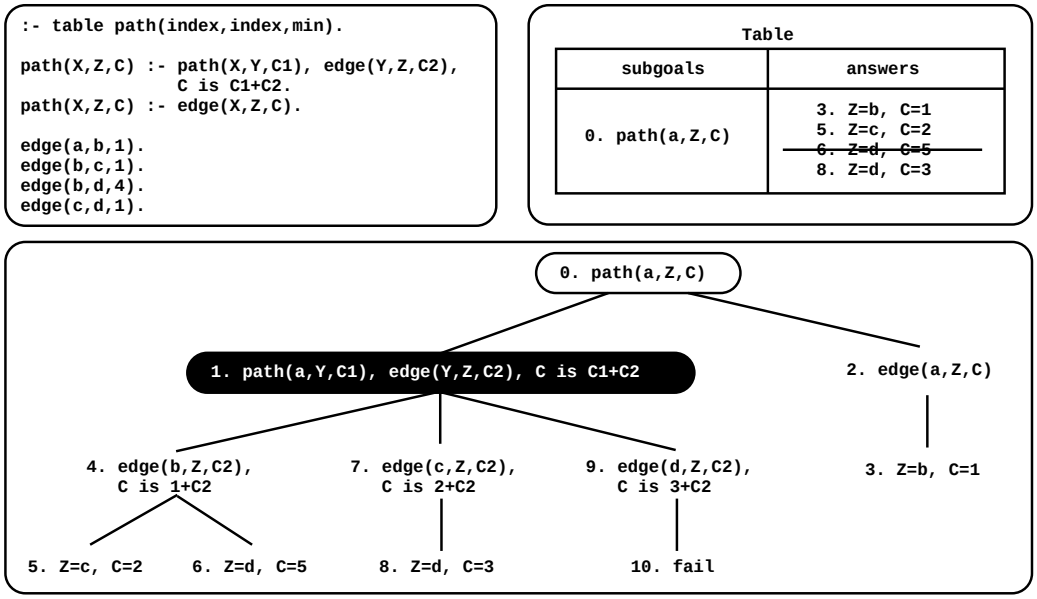


Figure 4 Using tabling with the mode operator *min*.

B-Prolog also extends the mode-directed tabling declaration to include a cardinality limit C that allows to define the maximum number of answers to be stored in the table space:

$:- \ table\ p(m_1, ..., m_n) : C$

Until the C limit be reached, all the answers generated are inserted in the table. After that, if a preferable answer is generated, it replaces the less preferable answer in the table. As we will see, this behavior can be recreated by using answer subsumption.

4 Mode-Directed Tabling and Justification

When the execution of a program generates a certain amount of answers, it could be interesting to have access to *proofs* showing that those answers are indeed correct. The process of generating proofs to answers is called *Justification* [10, 7].

Consider again the path problem, the justification for an answer could be, for example, the list of nodes traversed. Despite this apparent simplicity, generating justifications could be a very difficult task.

There are two main approaches for generating justifications. The first is called *post-processing justification* [10]. It has the advantage of being totally independent of the execution of the program but has the disadvantage of needing the program to be run twice: one to generate the answers and another to generate the justifications. The second approach is called *on-line justification* [7] and is done during program's execution, so it is much faster than the post-processing approach. Next, we discuss on-line justification in more detail.

Pemmasani et al. [7] proposed a technique for on-line justifications that involves the transformation of the original program. Figure 6 shows the result of applying such transformation to the program in Fig. 1. As we can see, the main difference between both programs is the inclusion of a new predicate, *store_evid/2*, at the end of the *path/2* clauses. This new predicate stores the justifications and guarantees that repeated answers are stored only once, thus avoiding infinite loops.

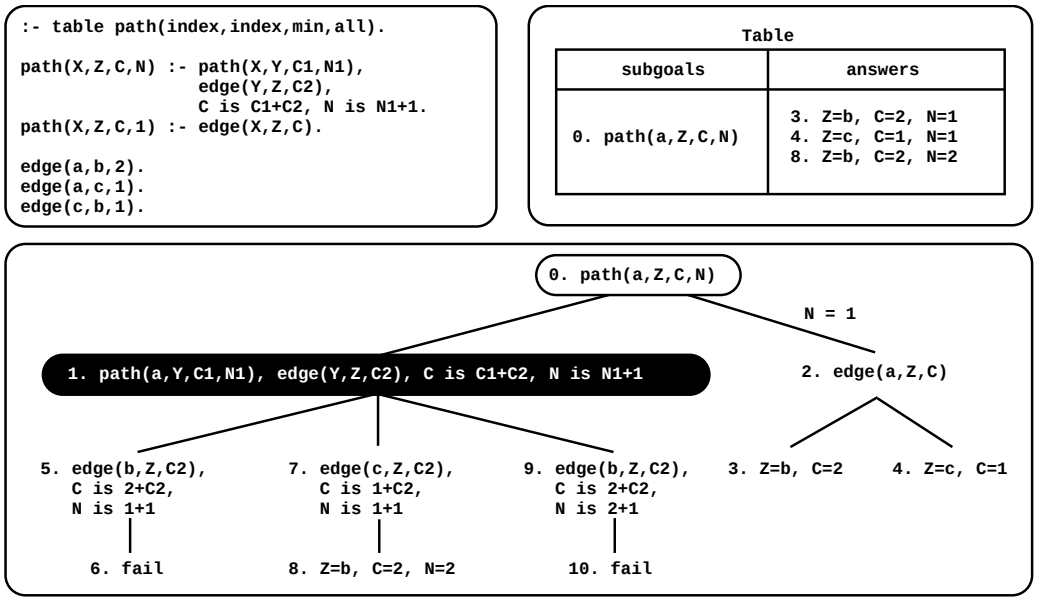


Figure 5 Using tabling with the mode operator *all*.

```
path(X,Z) :- path(X,Y), edge(Y,Z), store_evid(path(X,Z),
                        [((path(X,Y),true),ref(path(X,Y))),((edge(Y,Z,true),[])]).
path(X,Z) :- edge(X,Z), store_evid(path(X,Z),[((edge(X,Z),true),[])]).

edge(a,b).
edge(b,a).
```

Figure 6 On-line justification with the *store_evid/2* predicate.

Mode-directed tabling can also be used to generate on-line justifications [4]. Figure 7 shows how the program in Fig. 1 can be modified to implement on-line justifications using mode-directed tabling in the YapTab system.

The modifications are minimal. The path predicate includes an extra argument to represent the justifications and its declaration is modified to use mode operators. The built-in predicate *append/3* is also used to generate the list of nodes defining a path. With this, the task of avoiding infinite loops and controlling the generation of justifications is now delegated to the operators *index* and *first*, since they only store the first answer found for a certain path. For example, executing the query *path(a,Z,J)*, we would get the answers {*Z=b, J=[(a,b)]*} and {*Z=a, J=[(a,b),(b,a)]*}, where variable J represents the justification for the corresponding path.

```
:- table path(index,index,first).

path(X,Z,J) :- path(X,Y,J1), edge(Y,Z), append(J1,[(Y,Z)],J).
path(X,Z,[(X,Z)]) :- edge(X,Z).

edge(a,b).
edge(b,a).
```

Figure 7 On-line justification with mode-directed tabling.

5 Mode-Directed Tabling and Preferences

Logic programming is commonly used to solve optimization problems. When we write that kind of programs, it could be difficult to define the solution as a simple maximization or minimization problem. Preference Logic Programming (or Preferences) tries to solve this issue by dividing the program into the *specification of the problem* and the *definition of the optimal solution*. A very simple and declarative syntax for Preferences is the one proposed by Govindarajan et al. [2]. To better understand its syntax, let us consider the example in Fig. 8.

```
% problem specification
path(X,Z,C,D) :- path(X,Y,C1,D1), edge(Y,Z,C2,D2), C is C1+C2, D is D1+D2.
path(X,Z,C,D) :- edge(X,Z,C,D).

edge(a,b,1,4).
edge(b,a,1,3).

% preference clauses
path(X,Z,C1,D1) < path(X,Z,C2,D2) :- C2<C1, !.
path(X,Z,C1,D1) < path(X,Z,C2,D2) :- C2=C1, D2<D1.
```

Figure 8 Example of a program using Preferences.

The program finds the shortest path between two nodes in a graph and, if there are two or more paths with the same cost, the tie is undone by selecting the answer with the less distance. The program is divided in two parts: the problem specification (the top part in Fig. 8) and the preference clauses that are responsible for selecting the optimal answers for the problem (the bottom part in Fig. 8). The symbol $<$, used in the preference clauses, can be read as *"is less preferred than"*.

To efficiently implement Preferences, Guo et al. [5, 6] proposed the use of mode-directed tabling. For that, they introduced slightly modifications to Govindarajan's original syntax together with a *program transformation* that takes advantage of mode-directed tabling. For example, if we consider the program in Fig. 8, we only need to declare the predicate arguments subjected to preferences which, in this case, corresponds to include the declaration *path(index,index,<,<)* at the top of the program, meaning that the first two arguments are indexed and the last two are subjected to preferences. Starting from this declaration, we next describe Guo's program transformation:

- The first $<$ argument in the declaration of the tabled predicate is replaced by the mode operator *last*. The following $<$ arguments, if any, are replaced by the mode operator *first*.
- The head of the tabled predicate is modified by adding the word *New* to the name of the predicate.
- A new clause, with the same name as the original tabled predicate, is added to the program in order to control the execution of the program and the correct generation of the preferred answers.

For our example, this corresponds to the code in Fig. 9.

However, Guo's approach has one major limitation: all indexed arguments should be instantiated when calling a transformed tabled predicate. The problem resides in the new clause introduced by the third transformation item described above. For example, if we call the open query *path(X,Y,C,D)*, the new *path/4* clause will then call *pathNew(X,Y,C,D)*, in order to generate answers, after what variables X and Y will be instantiated with an answer

```
:- table path(index,index,last,first).

path(X,Z,C,D):-
   pathNew(X,Z,C,D),                    % generate answers
   (path(X,Z,C1,D1) ->                  % if old answers ...
      path(X,Z,C1,D1) < path(X,Z,C,D)   % ... test if the new answer is preferable
   ;
      true                              % otherwise accept the new answer
   ).

pathNew(X,Z,C,D) :- path(X,Y,C1,D1), edge(Y,Z,C2,D2), C is C1+C2, D is D1+D2.
pathNew(X,Z,C,D) :- edge(X,Z,C,D).
```

Figure 9 Preferences with mode-directed tabling.

(for instance {*X=x, Y=y*}). Next, when calling *path(X,Y,C1,D1)* (now *path(x,y,C1,D1)*), to get old answers from the table and check if the new answer is preferable, the problem arises since the first call was *path(X,Y,C,D)* and the new call is *path(x,y,C,D)*. Remember that tabling only consumes answers from the table space when it detects similar calls, which is not the case here. To solve this problem, we next propose some modifications to Guo's program transformation in order to allow calling open queries. Figure 10 shows our proposal along with the execution tree and table space for the same example and query goal *path(X,Y,C,D)*.

Initially, the original variables are first copied, using the built-in predicate *copy_term/2*, and only then we call the *pathNew/4* predicate (step 1). This call then generates a first intermediate answer (for illustration purposes, we are simplifying some steps here) for the path between *a* and *b* with cost 1 and distance 4 ({*X=a, Z=b, C=1, D=4*}). After that, we call the new *pathPrefer/2* predicate, introduced by our transformation, which is used to control the correct generation of the preferred answers. The *pathPrefer/2* predicate receives two arguments, one with the new path found and another with the copy of the variables in the original call (step 2).

In the first clause of *pathPrefer/2*, the global variable *match* is set to 0 (we will explain the usefulness of this step next) and then we call *path/4* with the copied variables of the original call (step 3). By doing that, we guarantee that a similar call will be done, hence avoiding Guo's problem. Tabling will thus access the table space and get one by one the available answers. In this case, there are no answers in table, so execution suspends and the second clause for *pathPrefer/2* is tried.

The second clause starts by executing a cut operation (!). This action is done in order to prevent the computation to be later resumed to the first *pathPrefer/2* clause (step 3). Next, it verifies if the global variable *match* is set to 0. We should note that the variable *match* is only set to 1 in the preference clauses, which means that there is at least one matching (but not necessarily preferable) answer in the table space. More, the cut operation performed at the end of the first *pathPrefer/2* clause ensures that, if we are considering the second *pathPrefer/2* clause, it is because the first clause did not succeed in finding a preferable answer. Thus, if the global variable *match* is set to 0 in the second clause, it is because there are no matching answers for the path we are considering, so the new answer can be safely inserted in the table (step 4). If the variable *match* was set to 1, that would mean that, at least, one matching answer existed in the table, and thus the current new answer must be discarded.

In the continuation, we backtrack to node 1 and a second answer is found for the path between *a* and *b* with cost 1 and distance 3 ({*X=a, Z=b, C=1, D=3*}). Again, the first

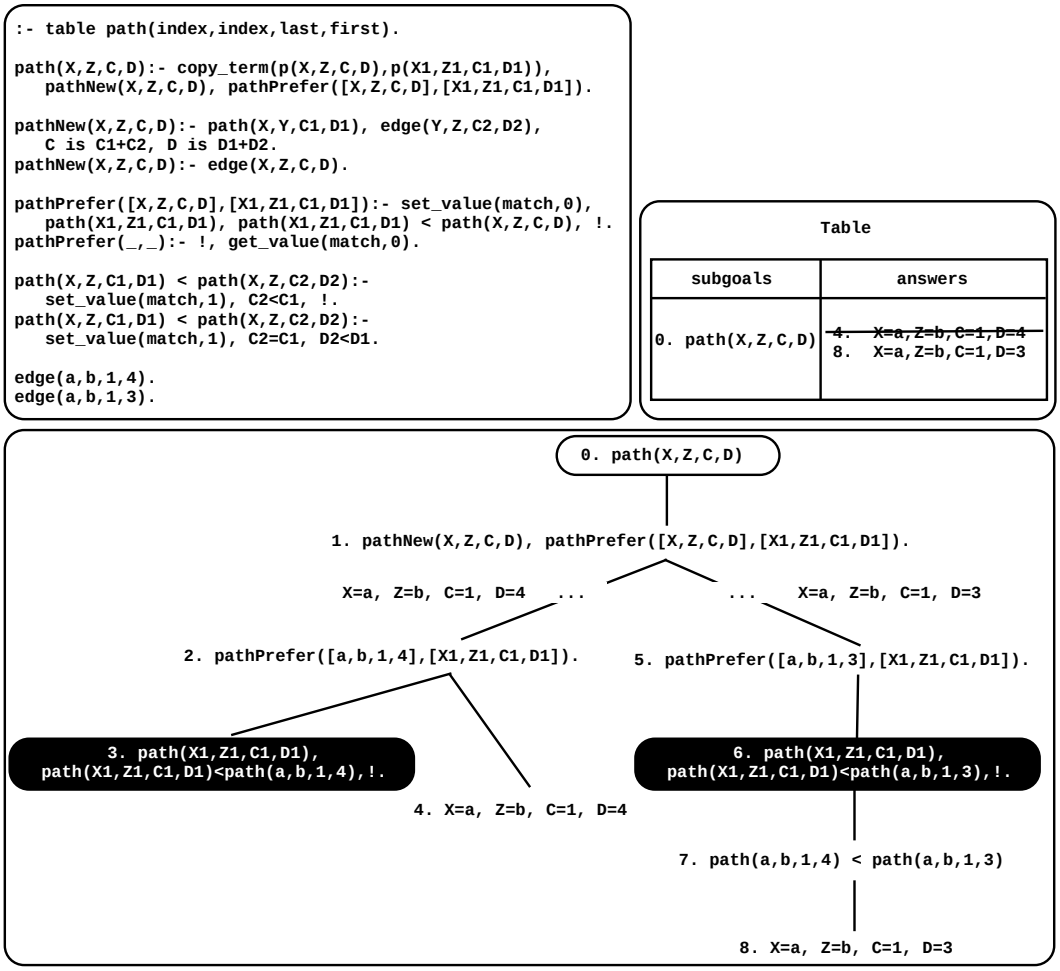


Figure 10 Using our transformation for Preferences with mode-directed tabling.

clause for *pathPrefer/2* is called (step 5) but now, there is an answer in the table space, the one found at step 4. Using the preference clauses (step 7), it is concluded that the new answer is preferable than the previous one. The predicate thus succeeds and the new answer is inserted in the table space replacing the older one (step 8).

6 Mode-Directed Tabling and Answer Subsumption

Traditional tabling only inserts an answer in table space if it is not a variant of one already there. In this paper, we presented mode operators that allow changing this behavior, giving us more control over the process of answer insertion. In particular, in the previous section, we discussed how to use mode-directed tabling for defining our own criteria of optimization through preferences. Preferences could be seen as a sub-case of Answer Subsumption, a technique that we will now introduce.

Answer subsumption allows a newly found answer to modify the set of answers present in the table. Plus, it can generate a third answer based on such set, instead of choosing among the old and new answers, like we ought to do with Preferences. This possibility allows us to construct, for example, counters and aggregates.

XSB Prolog has two answer subsumption mechanisms [13]. The first one is called *partial order answer subsumption* and is comparable, in terms of functionality, with Preferences. Consider that we want to use it with the program that finds the shortest path in a graph:

```
path(X,Z,C):- path(X,Y,C1), edge(Y,Z,C2), C is C1+C2.
path(X,Z,C):- edge(X,Z,C).
```

then, we should declare the *path/3* predicate as:

```
:- table path(_,_,po(</2))
```

meaning that the third argument, related to the cost of the path, will be evaluated using partial order answer subsumption, where the preference predicate *</2* implements the partial order relation. The other arguments are considered to be index arguments. The preference predicate is then used to decide when a new answer is preferable to an answer already stored in the table space.

The second XSB's mechanism is more powerful and is called *lattice answer subsumption*. With this mechanism, we can write a preference predicate that can generate a third answer starting from the new answer and from the answer stored in the table. To use it with the last example, we only need to change the declaration of the *path/3* predicate to:

```
:- table path(_,_,lattice(min/3))
```

Note that the *min/3* predicate must have three arguments. This is necessary since this mechanism can generate a third answer from two others. For example, for the shortest path problem, the preference predicate *min/3* could be something like:

```
min(Old,New,Result) :- Old<New -> Result=Old ; Result=New.
```

Consider now that we run the open query goal *path(A,B,C)* and that we obtain the answers {*A=a, B=b, C=6*} and {*A=a, B=c, C=7*}. Next, if we find a third answer {*A=a, B=Y, C=1*}, it would make sense to also generate the answers {*A=a, B=b, C=1*} and {*A=a, B=c, C=1*} to replace the previous ones. Instead, with XSB, we get the answers {*A=a, B=b, C=6*} and {*A=a, B=c, C=1*}, meaning that XSB is only able to use the new answer found, {*A=a, B=Y, C=1*}, with just one matching answer present in the table.

Before describing our proposal for answer subsumption, that solves XSB's limitation, we present its behavior. For that, we consider the same *min/3* example and we present, in Fig. 11, an example showing how a sequence of new answers being found alters the table space. The left column shows the new answer being found and the right column shows the table state after considering that answer. In the middle column, we show the list of matching answers for the new answer at hand.

At step 0, we have the simplest case: the table space is empty, thus the new answer {*X=1, Y=2, C=4*} is inserted. Next, at step 1, we have the answer {*X=1, Y=2, C=3*}. This answer is a variant of the previous one and, since it is a preferable answer (3<4), it is inserted in the table and the previous one is deleted. At step 2, we have an answer with a free variable in the index arguments, {*X=X, Y=2, C=2*}. This answer matches with the previous answer and, since the new answer is preferable (2<3), the answer in the table space is updated to {*X=1, Y=2, C=2*}. In fact, by updated we mean that the previous answer is deleted and that this new one is inserted. Then, the original answer {*X=X, Y=2, C=2*} is also inserted in the table space, since it is not a variant of any other answer. At step 3, we have the opposite case: the new answer, {*X=4, Y=2, C=4*}, matches with the answer with

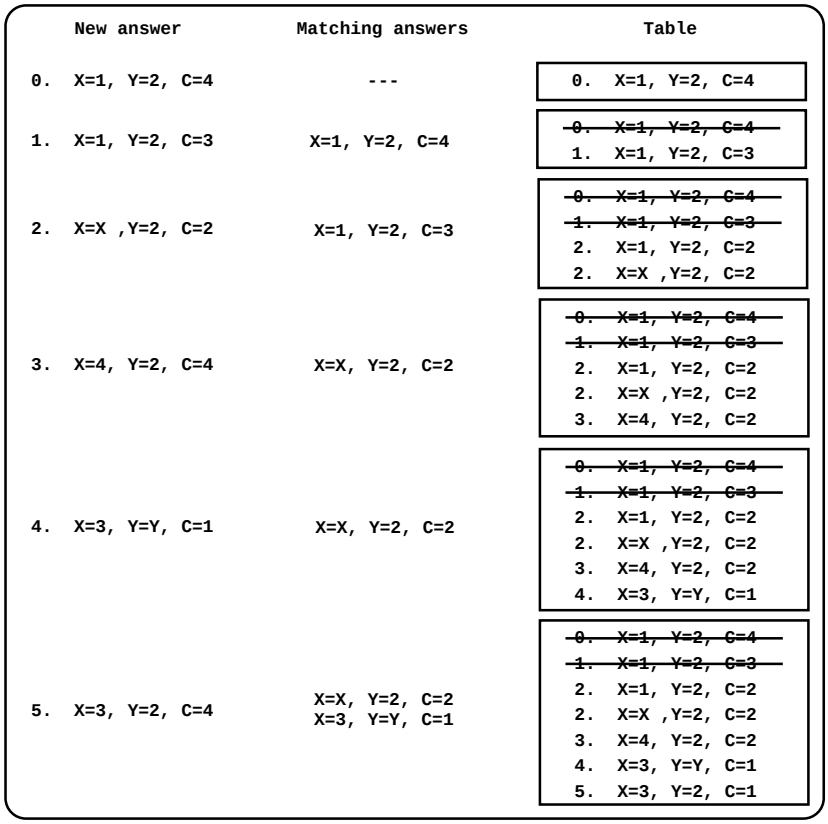


Figure 11 How our proposal for Answer Subsumption changes the table space.

free variables. Since the matching answer is a preferable answer (2<4), then a third answer, $\{X=4,\ Y=2,\ C=2\}$, is created and inserted in the table space.

At step 4, we illustrate a situation where the new answer $\{X=3,\ Y=Y,\ C=1\}$ unifies with a matching answer, $\{X=X,\ Y=2,\ C=2\}$, but they cannot be considered compatible. If we consider them compatible, then the matching answer will be updated to $\{X=X,\ Y=2,\ C=1\}$, which can lead to wrong answers. For example, if the answer $\{X=1,\ Y=2,\ C=4\}$ is then found, it will be turned into $\{X=1,\ Y=2,\ C=1\}$, which will be a wrong answer since the answer $\{X=3,\ Y=Y,\ C=1\}$ does not unifies with it. Our approach prevents those situations and, for such cases, the answer being found is simply inserted in the table space without modifying any previous answer.

The last step presents a situation where we have more than a matching answer. Since both matching answers are preferable to the new answer, $\{X=3,\ Y=2,\ C=4\}$, we must be careful as otherwise we can incorrectly create the answers $\{X=3,\ Y=2,\ C=2\}$ and $\{X=3,\ Y=2,\ C=1\}$. Since the answer $\{X=3,\ Y=2,\ C=1\}$ is preferable (1<2), we must consider only it to be inserted in the table space.

Next, we discuss, in more detail, the implementation of our answer subsumption proposal. We use a mode-directed tabled predicate named *answer_subsumption/3*. The first argument is the template representing the predicate name and the index arguments of the predicate to be answer subsumed. The second argument is the name of the preference predicate (as explained for XSB, this predicate must have three arguments). The third argument is the answer subsumption argument for the template in the first argument. Figure 12 uses again

```
path(X,Z,C) :- answer_subsumption(pathNew(X,Z),min,C).

pathNew(X,Z,C) :- path(X,Y,C1), edge(Y,Z,C2), C is C1+C2.
pathNew(X,Z,C) :- edge(X,Z,C).

min(Old,New,Result) :- Old<New -> Result=Old ; Result=New.
```

Figure 12 The shortest path example with Answer Subsumption.

the shortest path program to illustrate how the new *answer_subsumption/3* predicate can be used to implement answer subsumption in the YapTab system.

As you may have already noticed, for the *answer_subsumption/3* call in the body of the *path/3* predicate, the *pathNew/3* predicate appears with the answer subsumption argument, variable *C*, moved to the last argument of the *answer_subsumption/3* call. This is necessary since, for variant checking when calling a *answer_subsumption/3* subgoal, we should have the index arguments for *pathNew/3* separate from the non-index arguments. This is also the reason for the *answer_subsumption/3* predicate be declared as *answer_subsumption(index,index,last)* in our implementation. Figure 13 shows, in more detail, the implementation for the *answer_subsumption/3* predicate.

Initially, the third argument variable *Result* is first copied, using the built-in predicate *copy_term/2*, to two auxiliary variables, *CallResult* and *CopyResult*. The goal of this action is to avoid the original variable *Result* to become early bound. Next, using the built-in predicate *call/2*, we call the predicate to be answer subsumed in order to find a new answer. For example, to the shortest path example this corresponds to call *pathNew(X,Y,CallResult)*.

Then, we execute the *answer_subsumption_matching_answers/4* predicate. This predicate is responsible for accessing the table space and returning a list with all answers matching with the index arguments for the predicate at hand.

At the end, we test if the list of matching answers is empty or not. If the list is empty, we simply insert the *CallResult* answer returned by the call to the predicate at hand. Otherwise, if there are matching answers, we execute the *answer_subsumption_check_insert_update/5* predicate. This predicate is responsible for implementing the answer subsumption mechanism, putting into practice the rules described earlier for the example in Fig. 11.

A final and more interesting example of the power of using answer subsumption is the program shown in Figure 14, that takes advantage of the functionality of combining answers to produce a third answer.

```
:- table answer_subsumption(index,index,last).

answer_subsumption(Call,Op,Result) :-
   copy_term(Result,CallResult),
   copy_term(Result,CopyResult),
   call(Call,CallResult),
   answer_subsumption_matching_answers(Call,Op,CopyResult,AnswersList),
   (
      AnswersList \= []
   ->
      answer_subsumption_check_insert_update(Call,Op,CallResult,AnswersList,Result)
   ;
      Result=CallResult
   ).
```

Figure 13 Our implementation of Answer Subsumption with mode-directed tabling.

```
path(X,Z,N) :- answer_subsumption(pathNew(X,Z),sum,N).

pathNew(X,Z,N) :- path(X,Y,N), edge(Y,Z).
pathNew(X,Z,1) :- edge(X,Z).

sum(Old,New,Result) :- Result is Old+New.

edge(1,3).
edge(1,2).
edge(2,3).
```

Figure 14 Using answer subsumption to compute the number of paths between nodes in a graph.

The predicate *sum/3* adds the number of paths for the answer being found with the answer in the table, if any. For example, if we run the query goal *path(1,3,N)*, we will get the answer {*N=2*}, meaning that there are two different paths between nodes *1* and *3*, the first using the direct path between *1* and *3* and the other using the path through node *2*.

7 Conclusions

Mode-directed tabling is an extension to the tabling technique that supports the definition of selective criteria for specifying how answers are inserted into the table space. In this paper, we have presented a more general approach to the declaration and use of mode-directed tabling, implemented on top of the YapTab tabling system, and we have discussed the use of mode operators for implementing Justification, Preferences and Answer Subsumption in tabled logic programs. In particular, for Preferences and Answer Subsumption, we have presented alternative approaches that solve the limitations present in Guo's and XSB's original proposals, respectively. As further work, we plan to investigate how mode-directed tabling can be applied to other application areas.

Acknowledgements This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO/112158/2009).

References

1 W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

2 K. Govindarajan, B. Jayaraman, and S. Mantha. Preference Logic Programming. In *International Conference on Logic Programming*, pages 731–745. MIT Press, 1995.

3 Hai-Feng Guo and G. Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 181–196. Springer-Verlag, 2001.

4 Hai-Feng Guo and G. Gupta. Simplifying Dynamic Programming via Mode-directed Tabling. *Software Practice and Experience*, 38(1):75–94, 2008.

5 Hai-Feng Guo and B. Jayaraman. Mode-directed Preferences for Logic Programs. In *ACM Symposium on Applied Computing*, pages 1414–1418. ACM, 2005.

6 Hai-Feng Guo, B. Jayaraman, G. Gupta, and M. Liu. Optimization with Mode-Directed Preferences. In *7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 242–251. ACM, 2005.

7 G. Pemmasani, Hai-Feng Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online Justification for Tabled Logic Programs. In *International Symposium on Functional and Logic Programming*, number 2998 in LNCS, pages 24–38. Springer-Verlag, 2004.

8 F. Riguzzi and T. Swift. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *International Conference on Logic Programming, Technical Communications*, volume 7 of *LIPIcs*, pages 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

9 R. Rocha, F. Silva, and V. Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1 & 2):161–205, 2005.

10 A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying Proofs Using Memo Tables. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 178–189. ACM, 2000.

11 K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

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

13 T. Swift and D. S. Warren. Tabling with Answer Subsumption: Implementation, Applications and Performance. In *European Conference on Logics in Artificial Intelligence*, number 6341 in LNAI, pages 300–312. Springer-Verlag, 2010.

14 T. Swift and D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1 & 2):157–187, 2012.

15 Neng-Fa Zhou, Y. Kameya, and T. Sato. Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In *IEEE International Conference on Tools with Artificial Intelligence*, volume 2, pages 213–218. IEEE Computer Society, 2010.

16 Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a Linear Tabling Mechanism. In *Practical Aspects of Declarative Languages*, number 1753 in LNCS, pages 109–123. Springer-Verlag, 2000.

Generating flex Lexical Scanners for Perl Parse::Yapp*

Alberto Simões[1], Nuno Carvalho[2], and José João Almeida[3]

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

2 **Departmento de Informática, Universidade do Minho**
Campus de Gualtar, Braga, Portugal
narcarvalho@di.uminho.pt

3 **Departmento de Informática, Universidade do Minho**
Campus de Gualtar, Braga, Portugal
jj@di.uminho.pt

Abstract

Perl is known for its versatile regular expressions. Nevertheless, using Perl regular expressions for creating fast lexical analyzer is not easy. As an alternative, the authors defend the automated generation of the lexical analyzer in a well known fast application (flex) based on a simple Perl definition in the syntactic analyzer.

In this paper we extend the syntax used by `Parse::Yapp`, one of the most used parser generators for Perl, making the automatic generation of flex lexical scanners possible. We explain how this is performed and conclude with some benchmarks that show the relevance of the approach.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases flex, Perl, yapp, lexical analyzer

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.41

1 Introduction

There are diverse tools to write syntactic parsers in Perl. [1] describes some of these tools (like `Parse::Yapp`, `Parse::Eyapp` and `Parse::RecDescent`) and explain how they work. Also, [2] describes work in progress for an ANTLR generator for the Perl language. And recently a new module, named Marpa[1] also appeared. All these tools have a common denominator: they are syntactic parsers and little attention is given to the lexical analyzer.

While Perl is great with regular expressions, it is not that good when these expressions need to match a text file. Regular expressions were designed to match in a string, and to make them work with a file there are only two options:

- read the entire file to memory as a huge string, and apply regular expressions there: has the disadvantage of the memory used (as all text is loaded to memory), and if regular expressions are not written with efficiency in mind, they can take some time to match on huge strings.

* This work was partially supported by grant SFRH/BPD/73011/2010 funded by Science and Technology Foundation, Portugal.

[1] Available as a pure Perl implementation at `https://metacpan.org/release/Marpa` and as an hybrid Perl and C implementation at `https://metacpan.org/release/Marpa-XS`.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 41–50

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

- alternatively one can load chunks of text from the file, and try to use regular expressions in those chunks. If the string does not match one need to check why, as it might required another chunk to match.
  Of course most languages are line-oriented, making it easier to write this approach without exhausting memory. Nevertheless, this is not a generic solution.

This document proposes the use of the well known fast lexical analyzer flex [11] as the lexical analyzer together with one of the above mentioned Perl syntactic parser (namely `Parse::Yapp` and `Parse::Eyapp`), as described in [12][2]. For that, the `Parse::Yapp` language was enriched with extra information, and an automatic flex code and Perl–C glue code generators.

We start by discussing the state of the art on lexical scanners (section 2) and on syntax analyzers (section 3) in Perl.

Section 4 explains how C and Perl code can work together, and then specifies on how flex and `Parse::Yapp` can be glued together. The code generator (`Parse::Flapp`) is presented in section 5, followed by some tests and evaluations (section 6). Finally we draw some conclusions.

2 Perl Lexical Analyzers

Although most Perl written parsers use regular expressions as lexical analyzers, there are a couple of available modules that can be used as lexical analyzers, but most are too rudimentary, and that functionality is a side effect of their main goal. In this section we will compare on pure Perl approach and the well known flex. This comparison is mostly as a motivation for the usage of flex together with Perl, as we would not expect for a Perl implementation to be near as efficient as a C implementation.

- **Parse::Lex** is the only lexical analyzer available written entirely in Perl, and developed for that purpose;
- **flex**, while not a Perl approach, is the tool we are gluing in with Perl, and therefore a comparison is relevant.

We will first explain briefly how `Parse::Lex` works (we will skip flex on this explanation, as it is well known), and then present some comparison values in terms of memory usage and time efficiency.

2.1 Writing Lexical Analyzers in Perl

The common approach to write a lexical analyzer in Perl is to use regular expressions. The text to be analyzed is loaded into memory, and regular expressions are used to match and detect tokens. As tokens should be supplied to the syntactic analyzer in the same order they appear, these regular expressions can be anchored to the beginning of the string, making the matching faster. Also, given that strings are stored in memory, every time a token is found, the matching string can be replaced by the empty string, reducing the size of the string, and freeing some memory (although this can be a good solution for reducing memory usage, it will take some extra time).

[2] This article being cited, published in The Perl Review, is a tutorial of gluing a flex generated scanner with Perl code. It is a step by step guide of the needed code. The current work automatizes that process generating automatically the needed source code.

Unfortunately this approach has a big drawback, that is the memory used. Given this, we decided not to compare this vanilla technique. It is true that for some specific rules, and some specific input data, lexical analyzers like the ones generated with flex will need to read the entire document to memory. But that is not the usual situation, and therefore, loading before hand the entire document into memory is not appropriate.

Parse::Lex

The `Parse::Lex` module tries to abstract the lexical analyzer creation. Its lexer object is initialized with a flattened list of pairs (thus, a list with an even number of elements). Each pair include the token name and the regular expression that should be matched. An iterator is available, but unfortunately it does not detect when the end of input it reached. Therefore, a method to check for end of input needs to be used. Although the interface is very clean (see listing 1), this module creates very slow lexical analyzers. The next section will present details on its timings.

Listing 1 `Parse::Lex` lexical analyzer in Perl.

```
use Parse::Lex;

@tokens = ('PLUS'  => '\+',
           'MINUS' => '-'
           'NUM'   => '\d+', ...);
my $Lexer = Parse::Lex->new(@tokens);
$Lexer->configure(Skip => qr/[ \t\n]+/);

while (my $token = $Lexer->next) {
  last if $Lexer->eoi;

  ## do something with $token->name and $token->text
}
```

Unlike common lexical analyzers, where all rules are checked all the time and the rule that matches the longest data string is fired, in `Parse::Lex` the rules are analysed in order (much like as the vanilla technique pointed above would work). Therefore, if you have two rules that overlap (think of `a` and `a+b`) you need to specify the more generic first.

2.2 Lexers Efficiency Comparison

As stated before, generalized lexical analyzers written in Perl are prone to be based on regular expressions, which can produce inefficient analyzers. The use of *flex* based analyzers commonly produces faster and more efficient analyzers.

To sustain this claim several tests were made, one to benchmark CPU speed, and another to benchmark memory usage. Note that we are not stating that every flex based analyzer is faster than one written in pure Perl. But in general this is true.

Our tests compared the recognition of simple tokens: the four basic arithmetic characters, integer numbers and basic words (`/[a-zA-z]+/`). Spaces and newlines were recognized also, and ignored.

To perform the benchmarks three different lexical analyzers using different tools were implemented, just like detailed in previous section: `Parse::Lex` and flex.

The benchmark performed was to measure the lexical analyzers speed. Each analyzer was given the same input file to tokenize. The number of lines was increased and the time measured[3]. Table 1 shows the average execution time, in seconds, for each implementation for each given input file size.

Table 1 Average time, in seconds, to run lexical analyzers versus number of input lines (Intel Core 2 Duo, 2.4MHz).

Input lines	**Parse::Lex**	**Flex**
100	0.065454	0.015611
1 000	0.103696	0.017376
10 000	0.447719	0.024170
100 000	4.133743	0.118859

The performed benchmark help to sustain the claim that flex based lexical analyzers are prone to be faster than plain regular expressions based Perl analyzers.

This motivates the development of the approach described in this document, which plugs a flex based lexical analyzer to parser building frameworks.

3 Perl Syntactic Analyzers Overview

This section describes `Parse::Yapp` and `Parse::Eyapp`[4] because these tools will be used in the remainder of this work. Please refer to [1] for an overview on other syntactic analyzers in Perl. This section also includes a brief overview about Marpa because this tool was not included in the cited survey.

3.1 Perl Modules for Syntactic Analyzer construction

This section describes the tools that will be used for the construction of our parsers in Perl.

`Parse::Yapp` and `Parse::Eyapp`

`Parse::Yapp` (v1.05) is a pure Perl implementation of the well known `yacc/bison` algorithm [8]. This module reads a `yacc` specification with actions written in Perl, but is, otherwise, identical.

Unlike `yacc`, `Parse::Yapp` does not use a shared global structure (like `yylval`). Instead, the lexer function returns a pair that includes the identifier of the recognized token and the recognized string (or any other value you might want to pass to the syntactic analyzer). Note that inside the syntactic analyzer there is no need to define what types each production returns: semantic actions return Perl references to data structures, making it easy and clean.

The generated parser is reentrant, and it is possible to supply user data to the `yyparse` function.

`Parse::Eyapp` (v1.181) is an extended version of `Parse::Yapp`. It extends the `yapp` syntax with named attributes, extended BNF expressions, automatic abstract syntax tree

[3] Since time can be hard to measure accurately, each test was executed ten times. Then, the worst and best value were discarded, and the average time was calculated for the remaining values.

[4] In the remaining of the article we will refer to `Parse::Yapp` when a feature is both available on `Parse::Yapp` and `Parse::Eyapp`, and will explicitly refer to `Parse::Eyapp` when it is a specific feature of the extended module.

building, syntax directed data generation, tree regular expressions, tree transformations, directed acyclic graphs and it also includes a built-in lexical analyzer.

Marpa

Marpa is a parsing algorithm [9] for the recognition, parsing and evaluation of context-free grammars. The algorithm supports ambiguous grammars, and efficiently handles both left and right recursion. The algorithm is an evolution of Earleys' parsing algorithm [6], combined with improvements by *Joop Leo* described in [10] and by *Aycock et al* described in [3].

`Marpa::XS` (v1.006) is the latest stable implementation of this algorithm. This tool is able to create a parser for any grammar that can be described using the BNF notation [14]. But still the tokens recognizer must be written outside of the scope of the tool. Once the *tokenizer* is written it can be used to feed tokens, and corresponding values if required, to the parser, and the input can be parsed.

3.2 Efficiency Comparison

To compare `Yapp`, `Eyapp` and `Marpa` we implemented a simple calculator grammar as described on listing 2[5]. As semantic actions, instead of calculating the real value an abstract parsing tree was constructed. To test the three implementations we used a random generator.

Listing 2 Grammar used for syntactic analyzers benchmark.

```
gram: lines

lines: lines exp
     | exp

exp: exp '+' exp
   | exp '-' exp
   | exp '/' exp
   | exp '*' exp
   | '(' exp ')'
   | NUM
   | VAR
   | VAR '=' exp
```

As `Parse::Eyapp` is based on `Parse::Yapp` and we did not use any of the new features, we decided to use `Parse::Eyapp` built-in lexical analyzer and, for `Parse::Yapp` and `Marpa::XS`, use `Parse::Lex`, but both lexical analyzers are implemented in Perl. Table 2 presents running times for different input sizes, and table 3 shows memory usage. Regarding memory usage it is relevant to say that as we are building a parse tree it is natural the amount of used memory grows. Also, as at the end we are generating a dump for the tree structure, still more memory gets used.

Note that this comparison is not completely fair regarding `Marpa::XS`. Marpa computes all possible parse trees, making it able to parse ambiguous grammars, but also making it slower and exhausting more memory.

As a final note, our Marpa parser uses a ranking approach that is the Marpa feature that better mimic the precedence (and associativity) information given to `Parse::Yapp` and

[5] The precedence details are omitted in the grammar.

Table 2 Average time, in seconds, to parse input file versus number of input lines (Intel Core 2 Duo, 2.4MHz).

Input lines	**Parse::Yapp**	**Parse::Eyapp**	**Marpa**
100	0.108692	0.116950	0.375665
1 000	0.459240	0.469139	2.801585
10 000	4.436363	5.010674	70.969120

Table 3 Memory footprint for parsers versus number of input lines.

Input lines	**Parse::Yapp**	**Parse::Eyapp**	**Marpa**
100	3.86 MB	4.82 MB	8 202.98 MB
1 000	14.13 MB	15.07 MB	8 258.57 MB
10 000	117.04 MB	118.10 MB	9 008.07 MB

`Parse::Eyapp`. After some discussion with its author we found that a non ambiguous version of the grammar after some code refactoring gets some more interesting times and memory consumption. The time to process 10 000 lines drops to 9.218295 seconds, and the memory used to 279 MB.

4 flex and `Parse::Yapp`: How It Works

This section resumes the approach presented in [12] for gluing a flex syntactic analyzer with Perl.

As with any other language, the Perl community does not intend to implement everything in Perl. There are good libraries available (mainly written in C and C++) that should be used, and only an abstract interface written (also known as bindings). This means that the approach here described can be simulated using any other scripting language. It is just a matter of rewriting the glue approach.

The binding between a C library (or simply an object file) and Perl is written in a syntax known as XS [7]. It is a Domain Specific Language (DSL) [13] written on top of the C syntax, with some syntactical sugar that is recognized by the `ExtUtils::ParseXS` module, that generates a complete C (or C++) file that can be linked against Perl.

To glue flex with Perl we need to generate the standard flex input file. We decided to implement the flex file just like a standard implementation as if it was working with bison or yacc. For characters, the `yylex` function returns the character ASCII code (unfortunately flex is not supporting Unicode at the moment). For other tokens, the function returns an integer bigger than 256. This same flex files implements a function to access the `yytext` variable content. Listing 3 shows part of the flex generated lexical scanner.

Listing 3 Generated flex scanner.

```
%%
[0-9]+        { return 256; }
[A-Za-z]+     { return 257; }
[ \t\n\r]+    { /* ignore */ }
.             { return flapp_yytext[0]; }
%%
int flapp_yywrap(void) { return 1; }
char* flapp_yylextext(void) { return flapp_yytext; }
```

In the generated Perl lexer module, the method `yylex` is called to check for the next token type. If it is a character, it is returned both as token name and token content. If not, an array is accessed to check for the name of the token (note that this array is generated automatically), and the token name is returned together with the result of invoking the method to access the `yytext` variable content. Listing 4 presents the lexer interface for our calculator grammar[6].

Listing 4 Perl interface to a flex scanner.

```
sub flapp_lex {
   my $token = flapp_yylex();
   my @tokens = ('Num', 'Var');
   if ($token) {
      if ($token >= 256) {
         return ($tokens[$token - 256], flapp_yylextext())
      } else {
         return (chr($token), chr($token))
      }
   } else {
      return (undef, "")
   }
}
```

Other than these two files, all other generated files are related with the Perl module construction and compilation. These details are specific to Perl and we do not think they are relevant to be discussed in this article. Nevertheless, the next section will briefly introduce those files.

5 `Parse::Flapp`: The Code Generator

`Parse::Flapp` is a module that helps creating syntactic analyzers in Perl, glued with flex. At the time of writing the module only supports `Parse::Yapp`, but support for `Parse::Eyapp` and `Marpa::XS` is planned.

The module includes a command line tool, named `flapp`, that given a `Parse::Yapp` grammar with some extra minimal syntactic sugar, and a module name, creates a complete Perl module that implements the syntactic parser, for example:

```
$ flapp -module=My::Parser example.fyp
```

The syntactic sugar added to the `Parse::Yapp` (and `Eyapp`) grammar files is inspired in the way `Eyapp` defines terminal symbols. When listing the tokens that will be used in the grammar, the regular expressions that matches each token can also be defined. For example:

```
%token NUM = /[0-9]+/
%token VAR = /[A-Za-z]+/
```

In fact, `Eyapp` requires that expressions are between parenthesis, to instruct Perl to capture the matching string. For `Flapp` this is not required. But as the regular expressions are

[6] Instead of `yylex` the method is calling `flapp_yylex` as the Perl language parser is written using flex, and uses the default prefix. So, if we did not use a different name, the linkage process would not work.

used directly in the flex file, they need to be flex compatible, and not standard Perl regular expressions. This fact limits the token definition, namely if the user needs to interpolate string variables, or needs to use any construct not recognized by flex. In a future version `Flapp` might include a regular expression parser to validate if the expression is compatible with flex, or even rewrite portions of it. `Flapp` rewrites the grammar file removing the regular expressions. This makes the grammar file compatible with both `Parse::Yapp` and `Parse:Eyapp`.

The `flapp` script generates a Perl module that includes the following files (assuming the module name is `My::Parser`):

- `fl_Parser.l` — the flex lexical scanner, as described in the previous section (see listing 3);
- `lib/My/Grammar.yp` — the grammar received as input, with the `Parse::Flapp` specific syntax removed (this is a standard `Parse::Yapp` grammar);
- `lib/My/Parser.pm` — the Perl module which includes the lexical analyzer method, that interfaces with the flex lexical scanner (see listing 4);
- `Makefile.PL` — a standard Perl module makefile, that compiles and links the C code with Perl;
- `MANIFEST` — a manifest file, that lists all the files included in the module;
- `parse.pl` — a small Perl script to use the parser *right out of the box* (see listing 5);
- `Parser.xs` — the interface file that explains Perl how to deal with the C functions (see listing 6);

Listing 5 Small Perl script using the generated module (`parse.pl`).

```
use My::Grammar;
use My::Parser;

use Data::Dumper;
my $output = flapp();
print Dumper($output);

sub flapp {
  my $parser = My::Grammar->new;
  $parser->YYParse(yylex   => \&My::Parser::flapp_lex,
                   yyerror => \&My::Parser::flapp_error);
}
```

To make this module general [5] (at least in a near future), the module implementation was based on templates [4], one for each generated file. The use of templates makes the tool easier to extend and maintain.

6 Tests and Evaluation

To compare with the other approaches we ran the resulting `Flapp` parser for the same kind of data used before. The results were similar to `Parse::Yapp` both in time and memory (check table 4). Nevertheless, we can lower the memory consumption in approximately 25 MB, and the execution time in one second. Although these differences are not extremely high, they will be relevant for bigger input data.

Listing 6 C glue code, written in the XS syntax (`Parser.xs`).

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "lex.flapp_yy.c"

MODULE = My::Parser              PACKAGE = My::Parser

int
flapp_yylex()
    OUTPUT:
         RETVAL

char*
flapp_yylextext()
    OUTPUT:
        RETVAL
```

Table 4 Running times and memory footprint for `Parse::Yapp` and flex.

Input lines	Running time	Memory footprint
100	0.098 628 sec.	1 922.30 KB
1 000	0.375 966 sec.	9 127.16 KB
10 000	3.300 446 sec.	82 595.58 KB
100 000	34.743 845 sec.	857 262.58 KB

7 Conclusions

There is no doubt that flex is more efficient than any other lexical scanner implemented in Perl. Although there is not a pure Perl implementation of the flex algorithm, we do not think it would beat its C counterpart.

Nevertheless, Perl is great for dynamic data structures. We can argue that C data structure are faster, but they are harder to implement (a flex scanner is not that hard to use). Therefore, the composition of Perl with flex is relevant. But the process of using C from Perl is not that easy, and that can be an obstacle in the use of flex from within Perl.

The `Parse::Flapp` module solves this problem by making that task completely automatic, generating a Perl module ready to use and edit for further features.

The current `Parse::Flapp` implementation, still a prototype, generates new modules to parse with `Parse::Yapp` and `Parse::Eyapp` syntactic analyzers, but is easy to extend for other parser generators. In the future `Parse::Flapp` should also be able to update a generated module (and not only bootstrap). That would be indispensable for standard software development methodologies.

References

1 Hugo Areias, Alberto Simões, Pedro Henriques, and Daniela da Cruz. Parser generation in Perl: an overview and available tools. In Luis S. Barbosa and Miguel P. Correia, editors, *INForum'10 — Simpósio de Informática (CoRTA2010 track)*, pages 209–212, Braga, Portugal, Setembro 2010. Universidade do Minho.

2 Hugo Areias, Alberto Simões, Pedro Henriques, and Daniela da Cruz. Parser generation in Perl: Crafting an AnTLR back-end. In Raul Barbosa and Luis Caires, editors, *INForum'11 — Simpósio de Informática (CoRTA2011 track)*, pages 258–269, Coimbra, Portugal, Setembro 2011. Dep. de Eng. Informática da Universidade de Coimbra.

3 John Aycock and R. Nigel Horspool. Practical earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

4 Darren Chamberlain, David Cross, and Andy Wardley. *Perl Template Toolkit*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

5 Mark Jason Dominus. *Higher-Order Perl: Transforming Programs with Programs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

6 Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

7 Tim Jenness and Simon Cozens. *Extending and Embedding Perl*. Manning Publications, August 2002.

8 Stephen C. Johnson. Yacc: Yet Another Compiler-Compiler. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

9 Jeffrey Kegler. Marpa. Web site: http://www.jeffreykegler.com/marpa [Last accessed: 19-03-2012].

10 Joop M. I. M. Leo. A general context-free parsing algorithm running in linear time on every lr (k) grammar without using lookahead. *Theoretical computer science*, 82(1):165–176, 1991.

11 Vern Paxson, Will Estes, and John Millaway. *The flex Manual*. The Flex Project, version 2.5.35 edition, September, 10 2007. Available at `http://flex.sourceforge.net/manual/index.html`.

12 Alberto Simões. Cooking Perl with flex. *The Perl Review*, 0(3), May 2002.

13 Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 2000.

14 Wikipedia. Backus–naur form. Web site: http://en.wikipedia.org/wiki/Backus-Naur_Form [Last accessed: 19-03-2012].

A Purely Functional Combinator Language for Software Quality Assessment*

Pedro Martins[1], João P. Fernandes[1], and João Saraiva[1]

1 **HASLab / INESC TEC**
Universidade do Minho, Portugal
{prmartins, jpaulo, jas}@di.uminho.pt

Abstract

Quality assessment of open source software is becoming an important and active research area. One of the reasons for this recent interest is the consequence of Internet popularity. Nowadays, programming also involves looking for the large set of open source libraries and tools that may be reused when developing our software applications. In order to reuse such open source software artifacts, programmers not only need the guarantee that the reused artifact is certified, but also that independently developed artifacts can be easily combined into a coherent piece of software.

In this paper we describe a domain specific language that allows programmers to describe in an abstract level how software artifacts can be combined into powerful software certification processes. This domain specific language is the building block of a web-based, open-source software certification portal. This paper introduces the embedding of such domain specific language as combinator library written in the Haskell programming language. The semantics of this language is expressed via attribute grammars that are embedded in Haskell, which provide a modular and incremental setting to define the combination of software artifacts.

1998 ACM Subject Classification D.2.11 Software Architectures, D.4.1 Process Management

Keywords and phrases Process Management, Combinators, Attribute Grammars, Functional Programming

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.51

1 Introduction

Software quality assessment is a relevant research topic, and the implications of quality assessment are even more intricate and interesting when we consider open source software (OSS). With this is mind, the *Certification and Re-engineering of Open Source Software* (CROSS) project[1] was presented with the global goal of assessing the quality of general-purpose OSS software.

While we observe a growing integration of OSS in various public and industrial organizations, the fact is that there are no substantial standards or analysis tools that can provide an assertive quantification of the overall quality of such products. This means that their use still incorporates several risks.

In the context of CROSS and of the work presented in this paper, our general intention is to be able of certifying OSS. We understand a Certification as the process of analyzing a software solution while producing an information report about it. Certifications are expected

* This work is funded by the ERDF through the Programme COMPETE and by the Portuguese Government through FCT - Foundation for Science and Technology, project ref. PTDC/EIA-CCO/108995/2008.

[1] http://twiki.di.uminho.pt/twiki/bin/view/Research/CROSS/ [Accessed in 25 March, 2012]

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 51–69

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

to process an OSS solution and provide a technical analysis of it, decreasing the exposure to risk associated to the adoption of OSS.

To be more concrete, the challenges undertaken within CROSS are four-fold: i) to select and address several OSS-specific certification problems; ii) to develop techniques for the analysis of both code and its documentation; iii) to develop a certification infra-structure for OSS projects that is open to contributions and freely available; and iv) to embark in several collaborations with leading IT companies so that the overall results of the project are available for them.

The work described in this paper contributes to goals i) and ii) above. Indeed, we introduce a combinator language that allows users to easily construct tailor made certifications that can actually be the result of gluing together simpler certifications. The language that we propose is being used as a central piece of a more elaborated goal in the lines of iii): we want to develop an infra-structure, a Web Portal, that works both as a repository of software analysis tools and as a service that allows the analysis and certification of OSS. Such service has to maintain the open source spirit of heterogeneous and distributed collaboration: the portal has to store all the tools produced and, more important, allow any user to create new certifications by arranging the tools inputs/outputs in the appropriate order. Also, users should be able of combining already existing certifications into more complex analyses.

The language that we introduce in this paper aims at allowing an easy configuration of the flow of information among processes/tools that run either in parallel or in chain to create certifications, and at the automatic analysis and generation of low-level scripts that implement such configuration.

This paper is organized as follows: In section 2 we provide an overview of the motivation and potential challenges this work faces. In section 3 we introduce our combinator language together with small examples of its usage. In section 4 we present an Attribute Grammar-based type checker mechanism to control the flow of information inter-processes, which also is responsible for the script generation, as it is explained in section 5. In section 6 we provide an overview of related works, in section 7 we discuss opportunities for improving and extending our work and in section 8 we conclude the paper.

2 Motivation

We have already mentioned that the work presented in this paper is integrated within CROSS, a project whose aim is to develop program understanding techniques capable of measuring the degree of quality of open source software while being able to cope with its collaborative, distributed and heterogeneous character.

The techniques for analyzing source code to produce in the context of CROSS should, either individually or combined with others, result in the production of reports called Certifications. In the context of our work in this paper, we have developed an XML-based representation for these reports. Also, certifications are often composed by smaller units that are capable of communicating with each other in order to achieve a state where the overall mechanics of each unit and the flow of information among them is capable of producing quantifiable results. In the remaining of this paper, we will address ourselves to this smaller units that contribute to a general goal as Components.

In detail, a Component is therefore a bash tool, that is capable of accessing and producing meta-data via the standard channels (the standard input, STDIN, and the standard output, STDOUT). Also, a component must be able of receiving arguments that define the type of the information that is received via STDIN and the type of the information that is to

be channelled through STDOUT. Also, components are often developed independently by different, heterogeneous and distributed teams, and their development and their integration in more complex certifications closely follows the philosophy of open source software development itself.

In Figure 1 we sketch the flow of information that has been implemented in order to produce a sample Certification called Certification 2. This is a certification that expects Java programs and that analyzes them according to two distinct sub-processes that are independent with respect to each other and therefore can be executed in parallel. One of these processes chains a series of software units, namely Java Slicer, Interface Analysis and csv2Report. The other, which is itself a certification called Certification 1, implements a Cyclomatic Dependency analysis while producing an information report. Finally, and since all certifications must produce reports that conform to our format, the two distinct flows of information are aggregated by a Reports aggregator, a component whose single responsibility is precisely to aggregate outputs.

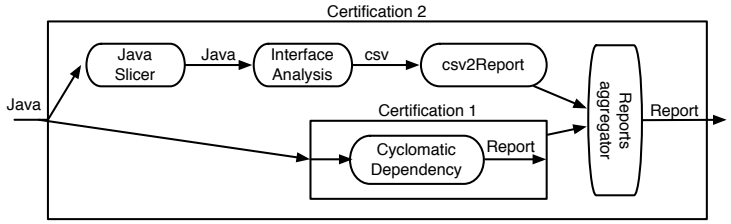


Figure 1 The flow of information implemented in Certification 2.

Several aspects of constructing certifications as illustrated in Figure 1 deserve further notice. For once, certification developers do not need to worry about concrete details of the sub-components or sub-certifications to use. Indeed, these sub-units must already exist in the certification framework, and the developer only needs to make sure that information flows, both semantically and syntactically, along the global certification process. This means that the output type of a component/certification must match the input type of the component/certification that immediately follows it. In these lines, for example, if a Slicer extracts the interface of a program, it can not feed a component that analyses concurrency requirements or super classes, even if we consider the same language being analyzed. Another thing to notice is that validations of this kind are automatically ensured by our system, as explained in section 4, that performs syntactic type checking.

In contrast to the approach that we propose in this paper, a traditional approach to implement a certification such as the one in Figure 1 would imply the cumbersome task of manually writing a script implementing the same features. Typically, such a script must be capable of launching processes in sequence and pipe their results, launching processes in parallel, timeout the processes and warn users if any of them is taking too long to run while minimizing this impact in processes that ran without problems, for example.

For demonstration purposes, we present in Appendix A an example of a script implementing Certification 2 in Perl. In fact, even being this a long and error-prone solution, it actually shows a simplified version of the script only. Furthermore, the implementation effort would significantly increase with more and more complex certifications (i.e., with certifications with larger numbers of sub-components). Furthermore, in this approach it is also very hard to systematically analyze the flow of information and to perform tests such as checking the existence of circular dependencies or ensuring that the types of inputs and outputs among components/certifications match.

In resume, we believe that having a large set of manually-written scripts is not a good practice: one can never be sure how well the script was written and how well it handles and processes errors. It is also very important to notice that, if the target system changes, this approach would require manually re-writing every script to support the new requirements and specifications.

In this paper we present a purely functional combinator language which allows an easy and intuitive combination of components/certifications and an easy creation of new certifications together with a script that implements them, and also with automatic inter-processes type checking and automatic code generation.

3 A Combinator Language for Certifications

The combinator language that we propose is written in Haskell, and starts by defining the data-types for certifications and components. These data-types are introduced in Listing 1 as Certification and Component, respectively.

Listing 1 Data types for Certification and Component.

```
data Certification = Certification Name ProcessingTree
data Component = Component Name InputList OutputList BashCall

data Language = Java     -- .java
              | C_Source -- .c
              | C_Header -- .h
              | Cpp      -- .cpp
              | Haskell  -- .hs
              | XML      -- .xml
              | Report   -- Report XML

type Arg      = String
type Name     = String
type BashCall = String
type InputList   = [(Arg, Language)]
type OutputList  = [(Arg, Language)]
```

A Certification has a name (e.g. Certification 1 as in Figure 1) and defines the particular information flow to achieve a desired global analyzis. This flow is represented by data-type ProcessingTree, that we introduced in Listing 3 and that we describe in detail later.

A Component is represented by a name, the list of arguments it receives and the list of results it produces. These lists, that are represented by type synonyms InputList and OutputList, respectively, have similar definitions and consist of varying numbers of arguments and results. The arguments(results) that are defined(expected) for a particular component are then passed to concrete bash calls. This is the purpose of type BashCall, which consists of the name of the process to execute on the system.

In the context of Figure 1, for example, the component Java Slicer may be executed by the bash call JSlicer with arguments -j and -i in order to slice the interface out of a piece of Java code. In a different scenario, executing the same process with arguments -j and -s would result in the slicing of the superclasses of the Java code (actually, if this execution occurred in the particular example of Figure 1, it would break the flow of information due to input/output mismatches).

For a generic Java Slicer component that could also be used in the context of Figure 1, we may define the instance of the Component data-type presented in Listing 2.

Listing 2 A sample instance of the Component data-type.

```
Component "Java Slicer" [("-j", Java)]
          [("-i", Java), ("-s", Java)] "./jSlicer"
```

Listing 3 Data-type for ProcessingTree.

```
data ProcessingTree = RootTree ProcessingTree
                    | SequenceNode ProcessingTree ProcessingTree
                    | ParallelNode ProcessingList ProcessingTree
                    | ProcessCert  Certification
                    | ProcessComp  Component Arg Arg
                    | Input

data ProcessingList = ProcessingList ProcessingTree ProcessingList
                    | ProcessingListNode ProcessingTree
```

In order to represent the flow of information defined for a certification, we have defined the data-type ProcessingTree, which is introduced in Listing 3.

The simplest processing tree that we can construct is the one to define a certification with a single component. This is expressed by constructor ProcessComp, which also expects a name to be associated to the component and the specification of the options to run the component with.

A certification can also be defined by a single sub-certification, here represented by ProcessCert.

In addition to these options, more complex certifications can be constructed by running processes in sequence, using SequenceNode, and in parallel, using ParallelNode. Constructor ParallelNode takes as arguments a processing list and a processing tree. The first argument represents a list of trees whose processes can run in parallel. The second argument is used to fulfill our requirement that all results of all parallel computations must be aggregated using a component. Therefore, this processing tree must always be a component (and this is ensured by our type checking mechanism in a way that we describe in detail in section 4) that is capable of aggregating information into one uniform, combined output.

The ProcessingTree data-type can be used, for example, to implement the global information flow of Certification 1 presented in Figure 1, which results in the instance presented in Listing 4.

Having in hand the data-types that we have defined so far, we could already create, in a manual way, certifications with all the capabilities that we propose to offer. As an example of this, Certification 2 of the previous section could be defined as presented in Listing 5.

Listing 4 A sample instance of the ProcessingTree data-type.

```
RootTree
   ProcessComp
        Component "Cyclomatic Dependency"
                  [("-j",Java)]
                  [("-r",Report)]
                  "./exec"
        "-j"
        "-r"
```

Nevertheless, expressing certifications in this manual approach would be of impractical use. This is precisely the main motivation to develop a language where simple components can be combined into more complex ones, which themselves can grow as large as needed in order to implement extensive certifications. So, instead of the certification implementation shown above, we suggest the equivalent representation that is introduced in Listing 6.

Listing 6 An example of a Certification using our Combinator Language.

```
javaSlicer =
    Component
        "Java Slicer"
        [("-j",Java)]
        [("-i", Java), ("-s", Java)]
        "./jSlicer"
interfaceAnalysis =
    Component
        "Interface Analysis"
        [("-j",Java)]
        [("-csv",CSV)]
        "./iAnalysis"
csv2Report =
    Component
        "csv2Report"
        [("-csv",CSV)]
        [("-r",Report)]
        "./csv2Report"
cyclomaticDependency =
    Component
        "Cyclomatic Dependency"
        [("-j",Java)]
        [("-r",Report)]
        "./cDepend"
reportsAggregator =
    Component
        "aggregator"
        [("-r",Report)]
        [("-r",Report)]
        "./csv2Report"

certification2 =
    Input >- (javaSlicer,"-j","-i")
          >- (interfaceAnalysis,"-j","-csv")
          >- (csv2Report,"-csv","-r")                    >|
    Input >- Input >- (cyclomaticDependency,"-j","-r")
                   +> "Certification 1"                  >|>
             (reportsAggregator,"-r","-r")
          +> "Certification 2"
```

With the addition of this code being smaller, elegant and easy to read and understand, the fact is that it will also be statically analyzed and type checked, and the script code that actually implements the certification it defines will be automatically generated. These features will be introduced in the remaining of this paper, together with the combinators that we use in our language, that we present in detail next.

Listing 5 A sample instance of the Certification data-type.

```
Certification
   "Certification 2"
   ParallelNode
      ProcessingList
           SequenceNode
                SequenceNode
                    ProcessComp
                        Component
                            "Java Slicer"
                            [("-j",Java)]
                            [("-i", Java), ("-s", Java)]
                            "./jSlicer"
                        "-j"
                        "-i"
                    ProcessComp
                        Component
                            "Interface Analysis"
                            [("-j",Java)]
                            [("-csv",CSV)]
                            "./iAnalysis"
                        "-j"
                        "-csv"
                ProcessComp
                     Component
                         "scv2Report"
                         [("-csv",CSV)]
                         [("-r",Report)]
                         "./csv2Report"
                    "-csv"
                    "-r"
           ProcessingListNode
               ProcessCert
                  Certification
                      "Certification 1"
                      ProcessComp
                          Component
                              "Cyclomatic Dependency"
                              [("-j",Java)]
                              [("-r",Report)]
                              "./cDepend"
                          "-j"
                          "-r"
     ProcessComp
         Component
             "aggregator"
             [("-r",Report)]
             [("-r",Report)]
             "./csv2Report"
         "-r"
         "-r"
```

The Sequence Processing Combinator

For sequencing operations, we define the combinator `>-`. This combinator defines processes that are to be executed in a chain, i.e, where the output of a process serves as input to the process that follows it. When sequencing processes, it is also the case that if one of process in the chain fails the entire chain will also fail.

The use of combinator `>-` must always be preceded by the use of constructor `Input`, which signals the beginning of an information flow. Then, as many components and certifications as needed can be used, as long as they again connected by `>-`. In Listing 7 we show an example of a chain of events defined using `>-`.

Listing 7 An example of Sequence of Processes using the Sequence Combinator.

```
Input >- (jSlicer,"-j","-i") >- (iAnalysis,"-i","-csv") >- certif
```

Combinator `>-` can be used to sequence certifications, components and other processing trees that are defined using the remaining combinators of our language. When it encounters a certification, the combinator connects the processes before it to the processing tree of the certification, and ensures that the result of this processing tree is then channeled to the processes that follow it. This is the case of the sub-certification called `certif` in Listing 7. When sequencing components, users need to supply to `>-` both the component and its input/output parameters. In the particular example of Listing 7, `jSlicer` is to be called with input parameter `-j` to state that the process accepts Java code as input and with output argument `-i` to state that it slices the interfaces out of that code.

It is worthwhile to notice that the arguments that are specified within components are very important in that they allow checking the flow of information inter-processes for correctness, as the input and output types must match when the information is channeled. When using `>-` to channel certifications, the user is constrained by the input and output types that were associated to it, and this is an information that must be carefully observed to ensure that the involved types do match.

Finally, the result of a sequence defined using `>-` is a processing tree that implements the combination of processes.

The Parallel Processing Combinator

Now, we introduce the combinator that enables the parallel composition of processes, This type of composition is actually supported by two combinators, `>|` and `>|>`. The first one is responsible for launching a varying number of processes in parallel, while the second is mandatory after a sequence of `>|` uses and chains all outputs of all processes to a component that is capable of aggregating them. In Listing 8 we show an example of how these combinators work together.

Listing 8 An example of Parallelization of Processes using the Parallel Combinators.

```
Input >- cert3                          >|
Input >- cert1 >- cert5                 >|
Input >- (jSlicer,"-j","-x") >- cert8 >|> (aggr,"-x","-r")
```

Combinator `>|` takes either a processing tree, a component, a certification or a set of processes constructed using the other combinators. The arguments of `>|` must always begin by constructor `Input`, to give a clear idea of the flow of information. In the case of this listing it is indeed easy to spot where the information enters a parallel distribution.

As for combinator `>|>`, it is mandatory for it to appear in the end of a parallelized set of processes. It is used to aggregate all the outputs of all the child processes into a single standard output, and it is able of combining varying numbers of parallel processes using an aggregation component.

It is worthwhile to explain further the relationship between the parallel combinators `>|` and `>|>`. In the trivial cases, `>|>` can actually replace the use of `>|`. This is the case of the process `Input >|> (aggr,"-x","-r")`, which is equivalent to `Input >- (aggr,"-x","-r")`, as both processes channel the input to the aggregation component `aggr`. Finally, combinator `>|` can never appear alone in a certification, due to the constraint that all parallel processes must be aggregated.

A Combinator to Create Certifications

Our combinator language includes also a combinator to create certifications and to associate names to them. This is precisely the purpose of combinator `+>`, which always combines a processing tree, given as its left argument, with a String, given to its right. It then creates a certification associating the name with the processing tree.

As an illustration of the use of `+>`, consider again the process implementations of Listings 7 and 8. In both cases, the result of running the implemented code is a processing tree (i.e., and element of type ProcessingTree), that needs to be given a name to become a certification (i.e., an element of type Certification). By simply appending, for example, `+> "certification"` to the end of both codes, we would precisely be creating certifications named certification with the respective trees of processes. In the case of Listing 7, this would result in the code show in Listing 9.

Listing 9 Using a Combinator to construct a Certification.

```
Input >- (jSlicer,"-j","-i")
      >- (iAnalysis,"-i","-csv")
      >- certif
      +> "certification"
```

An important remark about `+>` is that it analyzes the processing tree that it receives as argument and checks its correctness. This includes testing whether the implied types match, which means analyzing all the parallelized and sequenced processes for their input and output types, and see whether or not they respect the flow of information. Also, it is ensured that the processing tree produces a report which is a mandatory feature for a certification in our system.

Later in this paper, in section 4, we explain in more detail the features that are implemented by our type analysis and how they are actually implemented under our framework.

The 'Finalize' Combinator

The last combinator of our language is `#>>`, that combines a processing tree with a flag instructing it to either produce a script implementing that tree or to simply check its types for correctness. The examples presented in Listing 10 show the two possible uses for this combinator.

Listing 10 Using the finalize combinator.

```
Input >- (comp1,"-j","-x") >- (comp2,"-k","-o") #>> 't'
Input >- cert1 >- cert2 #>> 's'
```

In the first case, we are demanding a check on the types of running component comp1 after component comp2. This means that we are interested in knowing whether the return type of comp1 is the same as the input type of comp2.

With the second case, we are asking for the script that implements chaining certification cert2 after certification cert1 and also checks if the types match. In section 5 we explain in more detail how the script generation is achieved.

An Example Scenario

The examples that we presented so far were used to illustrate in simple ways the use of our combinator language. Still, we believe to have already demonstrated the simplicity that is involved in its use to create more and more complex certifications. In this section, we explore this argument further by introducing and describing the certification process of Listing 11.

Listing 11 Example of a parallel process in the middle of a sequence of processes.

```
Input >- (comp1,"-s","-ast") >- parallel >-
                                        (comp2,"-h","-r") >- cert

parallel  = Input >- (comp3,"-ast","-x") >|
            Input >- (comp4,"-ast","-x") >|
            Input >- (comp5,"-ast","-x") >|> (compAggr,"-x","-h")
```

In this example we have introduced a parallel computation in the middle of a sequence of processes. An example of where such a scenario could be useful is in the case of having a set of processes to analyze an Abstract Syntax Tree (AST), but having an input as source code. This code then needs to be converted to an AST using, in our illustration, comp1.

Following a manual approach to implementing a script for this scenario would lead to a complex development process. Indeed, one would have to manually edit it to make sure the process corresponding to comp1 feeds each process on the parallel computation (that in this case is composed by 3 sub-processes but that could easily grow further).

Furthermore, imagine that we do not want the results of the parallel computation, but rather we want them to be compared against a repository of results to analyze the characteristics of our AST. For this, a certification cert has been implemented, but it does not take as input the same format that is returned by our parallel processes. A possible solution using our combinator language is to channel the result of the parallel processes to an auxiliary component comp2 that converts the formats so the information can be fed to the certification. But this is something that is not easily implemented by hand.

The overall proposal of performing everything manually would be considerably difficult and error-prone. One would have to mess with legacy scripts, potentially built by different programmers, to understand them, and to create the correct chain of information. And further difficulties still need to be resolved if one wants to ensure script robustness, and that the processes are controlled in terms of processing times and failures, for example. And this significative effort of manually building scripts is furthermore severely compromised considering script evolution. Indeed, small changes in particular certification sub-processes may lead to severe overall changes being required.

We believe that our combinator approach has the advantage of not only making it easier to create flows of information among process, that can be easily edited, but also of being highly modular. Indeed, our combinators receive as arguments small fragments that can be edited, managed and transformed in simple ways. Also, it does not require a significant effort

for a programmer to change a particular certification, making it able of producing different results or improving it by introducing new safe processing mechanisms.

4 Type Checking on Combinators

In the previous section, we have introduced our combinator language for the development of software certifications. In this section, we introduce a set of validations that are automatically guaranteed to the users of our system.

For once, and since we have chosen to use Haskell in our implementation, we inherit the advanced features of both the language and its compilers. In particular, the powerful type system of ghc helps us providing some static guarantees on the certifications that are developed. Indeed, the order in which the combinators of our language are applied within a certification is not arbitrary, and the uses that do not respect it will automatically be flagged by the compiler. The simplest example of this situation is the attempt to construct a processing tree without explicitly using constructor Input, but of course more realistic examples are also detected, e.g., not wrapping up a set of parallel computations with the use of >|> as well as the application of an aggregator.

Apart from static analyzes that are ensured by the compiler, we have also implemented some dynamic ones. Indeed, we also want to analyze if the types match in the flow of information defined for a certification. This means that the input type of a process must match the output type of the process feeding it. Taking the example on Figure 1, the output type of Interface Analysis must be the same as the input type of csv2Report, which in this case is csv.

In our setting, we perform such tests on elements of type ProcessingTree, that we use our combinators to construct. These elements are then analyzed using validations that are expressed as attribute grammars (AGs) [8]. The reasons for choosing this approach are essentially of two different natures: i) for once, we are analyzing tree-based structures, for which the AG formalism is particularly suitable; ii) secondly, because AGs have a declarative nature which in our context contributes to intuitive implementations that are easy to reason about and to further extend. In fact, we believe that implementing in our framework advanced AG-based and well studied techniques such as the detection of circular dependencies [5] and the use of higher-order attributes [17].

The analyzes that we have implemented in an AG-style rely heavily in the concept of functional zippers [6]. Indeed, every element that we may want to analyze (i.e., upon which we define analyzing attributes) first needs to be wrapped up inside a Zipper. A more detailed description of Zippers and of how we use them can be found in section 6.

As we have already mentioned, our type checking is performed on trees of the type of ProcessingTree and, because we used an AG-based approach, this analysis is broken down into tree nodes which, in our case, are represented by the Haskell constructors of the ProcessingTree data type. The type checking is computed as the value of an attribute called typeCheck. This attribute, of type Boolean, indicates whether or not the analyzed types are correct. Apart from this attribute two other are involved: input and output, that support typeCheck. These attributes are of type Language (see Listing 1) and for each tree node give the input and the output types of that subtree.

Next, we will explain how these three attribute are calculated in each tree node.

Type Checking Component nodes

Components are the simplest units of our processing trees, and represent simple processes without any actual flow of information. This means that their type is always correct, and that the value produced for attribute typeCheck is always True.

As for the attributes input and output, they are computed analyzing the component and its associated element of the Component data type, against the invocation that is made for it in a ProcessingTree. Indeed, whenever a component is associated to a certification, an element such as ProcessComp comp inp out is defined. But comp itself is an element of type Component, i.e., has the form Component name inplist outlist call. So, our validation starts by detecting whether or not inp (respectively out) is an option of inplist (respectively outlist). In case it is, attribute input (respectively output) returns the Language option associated with inp (out). Otherwise an error is raised.

Type Checking Certification nodes

Certifications are, similarly to components, simple nodes of a processing tree.

In our setting, an interesting feature about certifications is that they must always be created using combinator +>. Therefore, everytime this combinator is employed, as in `tree +> name`, we automatically inspect the value of attribute typeCheck that is computed for tree. If this value is True, we create the certification Certification name tree; otherwise, no certification is constructed and an error is raised.

Actually, it is not only necessary that a processing tree type checks in order for us to be able of producing a certification out of it. Indeed, a requirement of our system is that certifications must always produce a report within our format. Therefore, we also check if the output attribute computed for tree is Report prior to the creation of an actual certification.

Finally, when we consider sub-certifications within a certification, again we use the fact that they need to be created using +>. Indeed, if the construction of a sub-certification succeeds, then it is always true that the tests so far described have also succeeded. Therefore, for such certifications, i.e., for certifications under ProcessCert nodes, we can always ensure that it type checks. Also, because we use an AG based analysis the attributes input and output are very simple to implement: we return the value of the same attribute that is synthesized at the sub-tree of ProcessCert nodes.

Type Checking Sequence nodes

The sequence node construction is useful whenever we have a processing tree that is followed by another. This node channels the information from the first processing tree into the second one and returns the result of this second processing tree.

The input and output attributes for this node are very simple to compute. In fact, input is the input type of the first processing tree, and output is the output type of the second processing tree.

Similarly, the typeCheck attribute is also very simple to determine: apart from checking if the output attribute of the first tree is equal to the input attribute of the second one, our AG-based implementation also demands the typeCheck attribute on each sub tree individually and checks whether both have value True.

Type Checking Parallel nodes

Parallel nodes are the hardest to type check. The reason for this is that, in a parallel processing definition, all its sub processes must have the same input type and the same output type, and this output type must the same as the input type of the component that aggregates all the results that are computed in parallel.

Parallel nodes always have two children: the second child, of type ProcessingTree, is a component that aggregates all the results of the processes that run in parallel, which are given as the first child of the node, of type ProcessingList.

The first validation we perform is to check whether the output value of the ProcessingList matches the input value of the ProcessingTree. If it does not, an error is raised; otherwise, the processes within the ProcessingList need to be type checked.

Now, type checking processing lists is more complex in that it needs to analyze all the inputs of all the sub processes, which can be components, certifications or processing trees and see if they match, and do the exact same thing for the outputs. In an AG setting this implementation can be achieved as follows: we use the equality of the input and output attributes for the current element of the list and for the subsequent elements. By type definition the processing list can never be empty and must always contain at least one element so the attribute will always return a value.

The input and output attributes for parallel nodes are simple to compute: the input is the input of one of the elements of the processing list, as they are all the same, and the output is the output of the aggregator component. One important note is that, due to the importance of all the types within a processing list being correct, we have followed a safe approach: the input and output attributes for a processing list are always given only after the type checking for the entire list is performed.

5 Script Generation

We have shown a how set of processes can elegantly be combined into a certification, either in a sequence, in parallel or in any combinations of these two. We have also shown how this combinators are easy to read, understand and modify, and how we implemented a supporting type checking mechanism that guarantees a correct match of types throughout the processing chain.

In this section we describe how we can generate Perl scripts that implement the certifications that are created using our combinators. These scripts can be seen as the low level representation of our certifications: they describe the processing chain, handle the individual processes for their completeness and manage the flow of information throughout all the processes the certification is made of.

The script generation follows the same AG-based strategy that we have applied to type check our certifications. The basic idea behind it is that each tree node, that represents a part of the processing tree, generates the corresponding sub-piece of the global script, and that the overall meaning of the AG is the entire, fully working, script.

In Listing 12 we present an example of a script that was generated by the following combination: `Input >- (comp1 ,"-j","-x") >- (comp2 ,"-k","-o")`.

In this script both components are executed via the system call command capture_exec, their existence is verified and their STDERR output is checked for problems. Afterwards, their results are channelled to the process that comes next, which in the case of the first component is the second component, and in the case of the second component is the STDOUT of the script since the computations ended.

Listing 12 Example of a script.

```
#!/usr/bin/perl
use strict;
use warnings;
use IO::CaptureOutput qw/capture_exec/;
use IO::File;
$| = 1;
my $stdout0 = $ARGV[0]; # This is actually stdin

my $cmd1 = "./comp1 -j -x";
my ($stdout1, $stderr1, $success1, $exit_code1)
        = capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "** The process $cmd1 does not exist! **"; }
if (not $success1)
        { die "** The process $cmd1 failed with msg: $stderr1 ! **";}

my $cmd2 = "./comp2 -k -o";
my ($stdout2, $stderr2, $success2, $exit_code2)
        = capture_exec( $cmd2 . "<<END\n" . $stdout1 . "\nEND" );
if ($?) { die "** The process $cmd2 does not exist! **"; }
if (not $success2)
        { die "** The process $cmd2 failed with msg: $stderr2 ! **";}

print $stdout2;
```

The scripts that we generate perform process control and scheduling of computations both on chained and in parallel flows of information while still being readable and understandable. Nevertheless, constructing such scripts manually is still an error-prone task even for small certifications with small number of processes. A larger certification (with, e.g, over a dozen sub-processes) would imply a significant amount of time to be implemented and debugged, just to name some phases of the development process.

In fact, this situation would further deteriorate if we were considering integrating in our framework more advanced scripting features. We could, for example, be interested in time-outing the processes independently to ensure they do not go past a certain time frame, in controlling better the input and output information from processes (checking, for example, if input information is able to be processed though STDIN, i.e, respects the specific implementations on different programming languages and environments[2], etc) or in ensuring that, anytime an error occurs, the script actually creates a small report that is integrated in the final certification instead of just showing information through the standard STDERR.

We believe, however, that following an AG-based approach similar to our own would facilitate the integration of these features in structured and simple ways as one-of tasks that once achieved become automatically available for any certification, old and new. Furthermore, because tree nodes are modular units of an AG implementation, it is even easier to upgrade small parts of the script as needed. For example, implementing timeout features on parallel processes would imply changing only the corresponding attribute on the desired tree nodes.

Generating scripts automatically presents several difficulties that are orthogonal to any generation mechanism, including to our own AG-based setting. Code translation is challenged

[2] `http://en.wikipedia.org/wiki/Here_document` [Accessed in 25 March, 2012]

by the usual concerns of assuring that the result is both syntactically and semantically perfect, and that all constructors/primitives/declarations of the target language are correctly declared and used. Implementing multiple processes, for example, implies a tight control on the variables that carry their results and their inputs. In a chain of processes from A to B, the variable that stores the information produced by A must be the one that feeds B, and all the variables must have different names (and if they do not, they must be used in different execution contexts). Also, this mechanism is even harder to implement within parallel processing, where all the outputs are aggregated into one single process (remember Listing 3, where a parallel tree node has always a processing list and a component that aggregates information).

To implement the generation of a script we have followed a simple rule for the variables scope: their name follows the order in which their corresponding process appears in the tree. So, for example, if the script implements two processes, where the process A receives the input, processes it and sends it to process B, whose output is the certification report, all the variables related to A (first process) end in 1, and all the variables related to B (second process) end in 2. The name of the variables is always the same (except for the last character that is the number) and in this way we guarantee that variables remain exclusive to the process they are related to.

If the certification is composed by a sequence from a processing tree to another processing tree, than the names of the variables on the second processing tree start with 1 plus the number of variables on the first processing tree. Such mechanism is very easy to implement in our AG setting: we have created a very simple attribute whose meaning is the number of sub-processes per tree node. So if the first processing tree of a sequence has four sub-processes, then the variables on that processing tree are named from 1 to 4, and on the second processing tree the variable names start in 5.

For parallel computations this mechanism is very simple, except for a few requisites. First, we must ensure that whatever comes before the parallel computation feeds all its sub-processes. To do so, we ensure that any information is first assigned to a variable from which all the sub processes read their input. Also, the results from all sub-processes are channelled to a variable that aggregates them. It is important to notice that the outputs are not combined: they are simply channelled to a variable that feeds the aggregator component of the parallel computation, and it is the responsibility of such aggregator to read various inputs via STDIN, instead of reading a single instance that is the aggregation of all the results. We do so to preserve the information instead of risking changing it by combining functions.

On the sub-processes of a parallel computation the exclusivity of the names of the variables names is not important, since these are different processes with different execution environments. Nevertheless, it is important to preserve exclusivity inside the processes themselves, which we easily do simply by recursively calling the attributes that were responsible for creating the script in the first place.

After defining the scope rules for variables, the script generation via the attribute toScript is easy to perform, once again thanks to our AG-based mechanism. The code generation follows the syntactic rules of the target language (in this case Perl) and ensures that the constructors/primitives/parenthesis are written and in the correct form.

The attribute toScript on the root of the processing tree creates the headings necessary to the script (such as declaring global variables or importing Perl libraries) and in each tree node we generate the corresponding Perl code, which implies defining system calls by composing processes.

6 Related Work

In [2] an implementation of the orchestration language Orc [7] is introduced as an embedded domain specific language in Haskell. In this work, Orc was realized as a combinator library using the lightweight threads and the communication and synchronization primitives of the Concurrent Haskell library [11]. Despite the similarities on the use of combinators written in Haskell, this paper differentiates from ours because we do not rely on any existing orchestration language. Rather, we generate low level Perl scripts from combinators whose inputs are direct references to system processes (components). Also, our processes management does not rely on Concurrent Haskell, but rather on the parallelization features of the target system via system calls on the script.

The use of attribute grammars as the natural setting to express the embeding of DSLs in Haskell is proposed in [12, 13, 14, 15]. These embeddings use powerful circular, lazy functional programs to execute DSLs. Such circular, lazy evaluators are a simple target implementation of AGs used by several systems [9, 16]. To make such implementations more efficient we have adapted well-know AG [5] and program calculation [3, 4] techniques to refactor circular programs into strict ones.

Zippers were originally conceived by [6] to represent a tree together with a subtree that is the focus of attention, where that focus may move left, right, up or down within the tree. By providing access to the parent and child elements of a structure, zippers are very convenient in our setting: attributes are often defined by accessing other attributes in parent/children nodes. In our work we have used the zipper library of [1]. This library is generic, in that it works for both homogeneous and heterogeneous datatypes, as any data-type for which an instance of the Data [10] type class is available can be immediately traversed using this library.

7 Future Work

Further improvements in the presented library would be specially important and have greater impact in the combinators analysis mechanisms.

A possible improvement is on the type checking mechanism. This mechanism is already on a solid state and perfectly checks for all kinds of types mismatches along the flow of information, but their output is still a simple error to the user, warning him/her about the fact that, somewhere along the chain of processes, something somewhere has a type error.

With huge certifications it becomes very difficult to localize a type error. One improvement we are looking into is on the warnings produced by the typeCheck attribute. Such warning could display valuable information such as showing until which part of the process the types are ok or actually indicating the specific line along the combinators where the type validity breaks.

Another important improvement would be in the number of AG-based analysis we perform. In this state, the library only checks for types errors and generates a script, but the world of AGs is old and heavily studied, and there are a huge amount of works with algorithms that when implemented would, almost for free, greatly improve our library. A perfect example of a well-known AG-based algorithm we are looking forward to implement are circularity checks along the processing tree, where an interdependency between the process A and the process B could actually lead to a deadlock on the final script.

The script itself could also be the subject of some improvements, such as performing individual timeouts on processes, do a better control on the inputs and outputs and improve the warnings to the user.

8 Conclusion

In this paper we have presented a combinator library that supports the scheduling of processes, both chained or as parallel computations. Together with the combinators, we have presented multiple examples of how computations can be elegantly rearranged into new process work flows, and how such combinators relate with each other to easily create complex certifications.

We have also introduced AG-based, purely functional mechanisms, that are not only capable of performing automatic type checking on processes work flows but also of automatically generating scripts.

Implemented in an AG environment, our type checking system automatically guarantees that the input and output types of each sub-processes are right to the definition of a process work flow and of a certification, and do not break such definitions, while also managing to automatically create low-level implementations of certifications in the form of Perl scripts.

We believe the advantages of our system are two fold: first, the combinators create an intuitive and simple yet powerful environment to create not only certifications but also processes work flows in general, while ensuring their validation.

Secondly, our AG approach can be easily transformed with any change that both the definitions of certifications or of processes work flows needs to support. This mechanism is modular, easily extensible and upgrades on code generation or type checking in the form of new features and functionalities are easy to design and implement in a modular and concise way.

The combinator library, together with built-in definitions of certifications and components, an example of a script, one of a component and a README, are available at `http://wiki.di.uminho.pt/twiki/bin/view/Personal/PedroMartins/CombinatorLibrary`.

A Perl script for Certification Management

```
#!/usr/bin/perl
use strict;
use warnings;
use IO::CaptureOutput qw/capture_exec/;
use IO::File;
$| = 1;
my $stdout0 = $ARGV[0]; # This is actually stdin

my $cmd1 = "./script+1.pl -c -r";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }

#Before starting a parallel computationg we default the stdout to 0
$stdout0 = $stdout1;

my $handle2 = new IO::File();
my $pid2 = open($handle2, "-|");
if ($pid2 == 0) {
my $cmd1 = "./script+1.pl -r -x";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
```

```
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }
print $stdout1;
exit;
}

my $handle3 = new IO::File();
my $pid3 = open($handle3, "-|");
if ($pid3 == 0) {
my $cmd1 = "./script+1.pl -r -x";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }
print $stdout1;
exit;
}

my $handle4 = new IO::File();
my $pid4 = open($handle4, "-|");
if ($pid4 == 0) {
my $cmd1 = "./script+1.pl -r -x";
my ($stdout1, $stderr1, $success1, $exit_code1) =
capture_exec( $cmd1 . "<<END\n" . $stdout0 . "\nEND" );
if ($?) { die "**** The process $cmd1 does not exist! ****"; }
if (not $success1) { die "**** The process $cmd1 failed with msg: $stderr1 ! ****"; }
print $stdout1;
exit;
}


# Lets grab all the results of the parallel processes
my $stdout4 = <$handle2>. " " . <$handle3>. " " . <$handle4>;
my $cmd5 = "./script+1.pl -x -x";
my ($stdout5, $stderr5, $success5, $exit_code5) =
capture_exec( $cmd5 . "<<END\n" . $stdout4 . "\nEND" );
if ($?) { die "**** The process $cmd5 does not exist! ****"; }
if (not $success5) { die "**** The process $cmd5 failed with msg: $stderr5 ! ****"; }

print $stdout5;
```

References

1 Michael D. Adams. Scrap your zippers: a generic zipper for heterogeneous types. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, pages 13–24, New York, NY, USA, 2010. ACM.

2 Marco Devesas Campos and L. S. Barbosa. Implementation of an orchestration language as a haskell domain specific language. *Electron. Notes Theor. Comput. Sci.*, 255:45–64, November 2009.

3 João Paulo Fernandes, João Saraiva, Daniel Seidel, and Janis Voigtländer. Strictification of circular programs. In *PEPM'11: Procs. of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 131–140. ACM, 2011.

4 João Paulo Fernandes, Alberto Pardo, and João Saraiva. A shortcut fusion rule for circular program calculation. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, pages 95–106, New York, NY, USA, 2007. ACM Press.

5 João Paulo Fernandes and João Saraiva. Tools and Libraries to Model and Manipulate Circular Programs. In *PEPM'07: Proceedings of the ACM SIGPLAN 2007 Symposium on Partial Evaluation and Program Manipulation*, pages 102–111. ACM Press, 2007.

6 Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

7 David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. The orc programming language. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems*, FMOODS '09/FORTE '09, pages 1–25, Berlin, Heidelberg, 2009. Springer-Verlag.

8 Donald Knuth. Semantics of Context-free Languages. *Mathematical Systems Theory*, 2(2), June 1968. *Correction: Mathematical Systems Theory* 5 (1), March 1971.

9 Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998.

10 Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 26–37, New York, NY, USA, 2003. ACM.

11 Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.

12 João Saraiva. *Purely Functional Implementation of Attribute Grammars*. PhD thesis, Department of Computer Science, Utrecht University, The Netherlands, December 1999.

13 João Saraiva and Doaitse Swierstra. Data Structure Free Compilation. In Stefan Jähnichen, editor, *8th International Conference on Compiler Construction, CC/ETAPS'99*, volume 1575 of *LNCS*, pages 1–16. Springer-Verlag, March 1999.

14 João Saraiva and Doaitse Swierstra. Generic Attribute Grammars. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 185–204, Amsterdam, The Netherlands, March 1999. INRIA Rocquencourt.

15 Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and Implementing Combinator Languages. In Doaitse Swierstra, Pedro Henriques, and José Oliveira, editors, *Third Summer School on Advanced Functional Programming*, volume 1608 of *LNCS Tutorial*, pages 150–206. Springer-Verlag, September 1999.

16 Doaitse Swierstra, Arthur Baars, and Andres Löh. The UU-AG attribute grammar system, 2004. `http://www.cs.uu.nl/groups/ST`.

17 Doaitse Swierstra and Harald Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *International Summer School on Attribute Grammars, Applications and Systems*, volume 545 of *LNCS*, pages 48–113. Springer-Verlag, 1991.

PH-Helper – a Syntax-Directed Editor for Hoshimi Programming Language, HL

Mariano Luzza[1], Mario Marcelo Beron[1], and Pedro Rangel Henriques[2]

1 **National University of San Luis**
San Luis, Argentina
{mluzza,mberon}@unsl.edu.ar
2 **Universidade do Minho**
Braga, Portugal
prh@di.uminho.pt

Abstract

It is well known that students face many difficulties when they have to learn programming. Generally, these difficulties arise from two main reasons: i) the kind of exercises proposed by the teacher, and ii) the programming language used for solving those problems. The first problem is overcome by selecting an interesting application domain for the students. The second problem is tackled by using programming languages specialized for teaching.

Nowadays, there are many programming languages aimed at simplifying the learning process. However, many of them still have the same drawbacks of traditional programming languages: the language used to write the statements is different from the programmers' native language; and the syntactic rules impose many tricky restrictions not easy to follow.

This paper presents an approach for solving the problems previously mentioned. The approach consists of using: an *application domain* motivating for the student, the Project Hoshimi (PH); and a *programming environment*, PH-Helper that is a simple and user-friendly syntax-directed editor and compiler for Hoshimi Language (HL), the actual PH programming language.

1998 ACM Subject Classification D.2.6 Programming Environments

Keywords and phrases Syntax-directed Editors, Visual Programming Enviroments, DSL

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.71

1 Introduction

Since the early days of programming, people realized how difficult is to teach programming principles and imperative programming languages.

Many researchers in computer science and didactics have been working over this problem. On one hand, searching for its causes, relating difficulties with students background and courses curricula, and looking after the definition of the ideal profile for a successful computing student. On the other hand, designing languages and sketching programming environments that can overcome student barriers. Prolog [26, 4, 27, 21, 7, 8] (and other declarative programming languages), Logo [23, 31, 32][1], and Scratch [25, 17][2] are some of

[1] A detailed description can be found at http://en.wikipedia.org/wiki/Logo:(programming_language). See also the project homepage at http://stager.org/logo.html or http://el.media.mit.edu/logo-foundation/. Complementary information is available at http://mckoss.com/logo/.
[2] A detailed description can be found at http://en.wikipedia.org/wiki/Scratch-(programming_language). See also the project homepage at http://scratch.mit.edu/.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 71–89

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

the more relevant contributions coming out from this search effort. As fully discussed in [29], Alice [13, 14, 5, 20][3] can be compared to Scratch as a teaching tool for introductory computing. It uses 3D graphics and a drag-and-drop interface to facilitate a more engaging, less frustrating first programming experience. The underlying idea is to create a 3D programming environment that makes it easy to create an animation for telling a story, playing an interactive game, or a video to share on the web.

However the problem is far away from being overridden. Even nowadays, all over the world students face tremendous difficulties when they are introduced to programming. Obviously the most difficult step is the ability to understand problem statement and to write the algorithm; this is precisely the focus of the teaching activity. But unfortunately other minor issues emerge that are strong obstacles that obstruct students progress. One is the use of keywords in English language; another one is the (sometimes) complex details concerning language syntax, like punctuation symbols of composed statements structure. Visual programming languages and environments attempt to surmount these barriers that many times lead beginners to loss motivation to accomplish programming tasks [22, 30, 24, 1]. However visual programming does not scale properly and this approach had an impact far below the one expected 20 years ago.

Structured editing[4], also called *Syntax-directed editing* [18] or even *Language-based editing*, is another relevant approach that can actually help programmers to cope agilely with programming languages and overcome the referred syntactic idiosyncrasies [12, 16, 15]. The basic idea is to develop text editors that are aware of a specific language structure or syntax. In [11, 6] Henriques et.al. review the set of *Language-based tools*, like the editors under discussion, that can be derived and automatically generated from the language's context free grammar. Also in [19] the topic is explored to develop a meta-language based editor to create and analyze grammars. Some *Structured Editors* are *reactive* and others are *proactive*.

In the first class, the editor knows the language syntax but it does not guide the programmer; the programmer is free to write what he wants, and during or after writing the editor uses *highlighting*, *indentation* or other visual techniques to enhance the language keywords and the text structure. WinEdt[5] is just one example of this first class.

> WinEdt is a powerful and versatile text editor for Windows with a strong predisposition towards the creation of [La]TeX documents. It is used as a front-end for compilers and typesetting systems, such as TeX, HTML, etc. WinEdt's highlighting schemes can be customized for different modes and its spell checking functionality supports multi-lingual setups, with dictionaries (word-lists) for many languages.

Autocompletion—the ability to complete the words that are being typed, according to the context that restrict the choices[6]—is another feature offered by modern reactive language-sensitive editors that improve significantly the typesetting process reducing simultaneously the error-proneness. Emacs[7] and Vim[8], as well as TexMaker[9] are examples of text editors that

---

[3] See also the project homepage at `http://www.alice.org/` or `http://www.cs.duke.edu/csed/alice/aliceInSchools/`.

[4] Look at `http://en.wikipedia.org/wiki/Structure_editor` for a large discussion on that topic.

[5] See the homepage at `http://www.winedt.com/` for more details, or visit `http://www.winedt.org/`.

[6] See `http://en.wikipedia.org/wiki/Autocomplete` for more details and examples.

[7] See the homepage at `http://www.gnu.org/software/emacs/` or a guided tour at `http://www.gnu.org/software/emacs/tour/`.

[8] More information at `http://www.vim.org/` or `http://www.yolinux.com/TUTORIALS/LinuxTutorialAdvanced_vi.html`.

[9] For details look at `http://www.xm1math.net/texmaker/`.

offer autocompletion. *Autoreplace* is a related feature that involves automatic replacement of a particular string with another.

Also the code completion feature provided by Microsoft, known as IntelliSense, is similar to autocompletion but at a semantic level. IntelliSense editors are context sensitive and are aware of all program identifiers so far declared; in this way, they are able to suggest those identifiers that can be used in a certain place inside the program,

From the perspective of the work described in the paper, proactive structured editors are more interesting. In the second class, the editor is aware of the language syntax and it actively guides the programmer along the typesetting. At each moment, the editor knows what can be written so it shows the possibilities and after the user choice it goes on in the same way until it reaches an identifier or constant whose specific value just the programmer knows; this is the only thing he actually has to type. Obviously this kind of editors offer an effective help to the programmer that does not need to have a complete knowledge of the programming language syntax. Moreover this family of tool can be derived directly from the grammar and the editors can be generated automatically. Tim Teitelbaum was a pioneer [28] in this area. Relevant work in this field—automatic generation of syntax-directed editors—but concerned with visual editors, like the one (PH-Helper) discussed in this paper, has been done by Arefi et.al. [2, 3].

Another concern that is highly relevant when teaching computer programming is the problem statement by itself (its context, application area, and goals): many times, the problems proposed by teachers are not challenging enough, failing to motivate students. Without a strong motivation and practical validation, is even more difficult to attract students for this complex task. So, this topic (the choice of interesting and effective problems) is another issue that shall be taken into account towards a successful learning process. Regarding this point, we believe that start programming courses teaching Domain Specific Languages (DSL) instead of choosing immediately a General Purpose Language (GPL)can be a wise approach to overcome this last difficulty. A DSL is a formal language just a GPL is, so it is possible to reach the same learning objectives—like problem understanding, data structures choice, algorithm development, code implementation following elegant ways and good practices—but supported in a smaller, higher-level language (more abstract and concise), designed specially for a more restricted domain that can be more natural and appealing for students. Opposed to the proposals described above—as Prolog, Logo, Scratch, Alice—and many other that although innovative are sill GPL s, our proposal is a DSL aiming at taking profit from the benefits just set above.

Project Hoshimi [10][10], that will be introduced in section 2, is another trial to surpass part of the programming struggles above identified. Created by Richard Clark for the 2005 Microsoft's Imagine Cup contest, Project Hoshimi is a game based on .Net technology. The basic idea is to create a scenario (in the context of human bloody system) and ask students to program small robots that can navigate through the body and protect him from some diseases. The scenario is attractive and based on common sense and the task is challenging enough. The robots programming is done using a small set of primitive and intuitive commands.

[10] A detailed description can be found at `http://fr.wikipedia.org/wiki/Project_Hoshimi`. See also the project homepage at `http://www.projethoshimi.fr`. A Guide to The Imagine Cup Project Hoshimi is available at `http://blogs.msdn.com/b/edunhill/archive/2007/10/22/guide-to-the-imagine-cup-project-hoshimi-ai-competition.aspx`.

This paper is about an editing environment, called PH-Helper, that we are developing as a front-end for Project Hoshimi that aims at overwhelming the remaining difficulties. Namely, PH-Helper adds some more power to Hoshimi language (see section 3) and provides a syntax-directed editor—section 4 fully discusses it—that helps beginners with the language syntax avoiding boring errors. From a pedagogical perspective, a syntax-directed editor can also be adequately instrumented to force, or just to give advices, on the use of good programming practices like proper name conventions for the different type of identifiers, indentation, code commenting, etc. This topic is something that we plan to explore in a future version. We also discuss two other functionalities provided by PH-Helper: the compiler that generates C# to allow running the developed programs; and an XML exporter (see section 5). In order to demonstrate how PH-Helper works a case study is presented (see section 6).

Section 2 is a bit long description of the main characteristics of Project Hoshimi. It was included with two purposes: to make the paper more self-contained; and to make more natural to describe the extensions introduced, the code generation strategy and mainly to justify the decisions concerned with PH-Helper editor and enhance the syntactic aids it offers. Of course the reader can skip it and go directly to the other sections, coming back just on demand.

2 Project Hoshimi, an Overview

As said above and can be read at `http://fr.wikipedia.org/wiki/Project_Hoshimi`, Project Hoshimi (PH for short) is a computer game aimed at promoting creative use of programming languages and tools. PH is useful for:

- Conceptualizing *programming* as a creative activity through strategic simulation.
- Teaching *object oriented programming* and .Net technology.

In general terms, PH is a game whose goal is to cure human diseases using a set of NanoBots. Nanobots are small robots that can be injected in blood flow system and are able to solve health problems found in the human body. Each NanoBot has "artificial" intelligence for healing partially certain diseases. So we can said that the game consists in the *strategic simulation* of NanoBots behavior. Artificial Intelligence (AI) behavior is provided by the student programming those strategies in .Net technology.

2.1 Game Environment

The game is carried out inside of human body, where:

Game map is a tissue. Each map has 200 x 200 positions and the NanoBots have two possible movements: Horizontal and Vertical.

Game area is composed of blood (red), bones (gray), nerves (blue) and impassable sectors (black).

AZN area contains molecules employed for curing diseases. These molecules must be collected by a special kind of NanoBots known as NanoCollector. This area is never empty.

Hoshimi Points (HPo) are the disease zones; in this places the NanoNeedles are elaborated. Each HPo can receive up to 100 AZN molecules.

Injection Points (IPo) are the places where NanoBots are inserted. These points should not be changed during the game.

2.2 NanoBots

NanoBots are the central, or main, entities in the game. For this reason they are described in next subsections.

2.2.1 Kind of NanoBots

There are several kinds of NanoBots, namely:

NanoIntelligence (NanoI): this kind of NanoBot has two functionalities: i) Create all other NanoBots in the community, and ii) Give instructions to the other NanoBots. Each community (a team in the context of the game) has only one NanoI that is the first inserted inside the human body. It can move but it can not shoot.

NanoNeedle: this kind of NanoBot is created at Hoshimi Points in order to provide AZN molecules to the system. It belongs to the static defense; so it can not move but it can shoot.

NanoCollector: this kind of NanoBot collects AZN molecules and transfers them to the NanoNeedles. It is the only class that can move and shoot.

NanoExplorer: this kind of NanoBot is aimed at doing recognition task. This class has the widest vision range and the fastest movements, however it can not shoot.

NanoBlocker: this kind of NanoBot is used for diminishing enemy movements by changing blood density. It can not move and shoot.

NanoContainer: this kind of NanoBot is similar to NanoCollector class but it has bigger capacity. It can move but it can not shoot.

NanoWall: this kind of NanoBot generates a force field. This field can not pass by the enemies. It can not move and shoot.

NanoIPCreator: this kind of NanoBot generates a second injection point. It can move but it can not shoot.

2.2.2 NanoBots Characteristic

The NanoBots have the following characteristics:

Constitution: quantity of health also known as hit points.

Scan: vision distance.

Defense Distance: NanoBots' attack range.

Maximum Damage: maximum damage that a NanoBot can cause to the enemy during one time period.

Container Capacity: quantity of AZN that a NanoBot can store and transport.

Collect Transfer Speed: quantity of AZN that NanoBot can collect and transfer by time period.

All the characteristics aforementioned can be changed by the user using the following rules:

1. The characteristic values can not exceed a maximum value.
2. The addition of characteristic values can not exceed a "Total" established.

2.2.3 NanoBots Commands

The following commands are recognized by NanoBots:

ForceAutoDestruction – This operation is used by the NanoBot for self-destroying. Generally, this command is used when: i) There are many NanoBots or ii) More NanoBots are needed. All NanoBots have this command except the NanoI.

StopMoving – This command is employed for stopping the NanoBot movement. All others actions, such as attack or transport AZN, can be used. Afterward of executing this order other command can be used. This method is used for all NanoBots with movement.

MoveTo(Point) – This command is employed for moving NanoBots until the place indicated by *Point* (a point has two coordinates, `x` and `y`). All NanoBots with movement can run this command.

MoveTo(PointLst) – This command works as *MoveTo(Point)*. However, it receives a sequence of Points as its parameter. This sequence indicates different places where the NanoBots must be moved. All NanoBots with movement can run this command.

DefendTo(Point, int) – This command is used for attacking the point received as parameter. The parameter *int* is used for indicating the number of times that the NanoBot will attack its goal. All NanoBots with attack capability have this command.

CollectFrom(Point, int) – This order is used for collecting molecules AZN. This molecules are gathered from *Point*. The parameter *int* is used for indicating the number of turn that NanoBot will attack this point. All NanoBots with gathering capabilities execute this command.

TransferTo(Point, int) – This instruction allows NanoBots to download molecules AZN in the place indicated by *Point*. The parameter *int* is used for indicating the number of times that the NanoBot will download molecules in this point. All NanoBots with download capabilities execute this command.

Build(Type) – This command is executed by NanoI to create new NanoBots.

The reader interested in knowing more details about these commands shall look at `http://www.projethoshimi.fr/lab/richardc/index.php`—a compilation of tutorials for the competitions.

2.3 Defense Strategies

Depending on the game configuration, it is possible to play with other human player or other enemy controlled by the computer. The defense is scheduled in two levels. The first one is concerned with AI general strategy. The second one allows that a NanoBot attacks a particular position. Generally, the vision range is greater than the attack range, for this reason before attack the NanoBot must put the enemy inside its attack range. Obviously, the NanoBot must not be inside of enemy attack range. In order to create an effective defense strategy, the user must:

1. Surround vulnerable unit with protection, generally using NanoI.
2. Create static sentinels by employing NanoNeedles.
3. Guard the positions and behavior of the enemies.
4. Stay away from enemies.
5. Avoid to be closer of injection points.
6. Use formations highly cohesive.

3 Extending Hoshimi language

The current game platform available has a visual programming interface for developing strategies. This interface is poor, because it only implements the NanoBots basic operations described in section 2.2.3.

However the Project Hoshimi programming language, HL, has some severe restrictions: it does not allow to work with variables, neither supports data structures such as list, hash tables, etc., nor assignments.

PH-Helper gives a practical solution to the problems mentioned above and it provides more flexible statements for implementing strategies.

On one hand, PH-Helper allows to declare and handle variables of primitive (int, Point, etc.) or composed (list, dictionaries, hash tables, etc) types. Variables are defined when the user creates a NanoBot. PH-Helper also extends the original language with the assignment statement; it also allows the programmer to pass variables as parameter to other instructions.

On the other hand, the conditional statements are also extended supporting now any kind of logical expression. Moreover, PH-Helper adds new loop statements, such as *foreach*, for traversing complex data structures like dictionaries and lists.

All the programming activity above described is carried out using an intuitive and simple graphical interface. This interface clearly shows: i) The type of variables that the user can define, and ii) For each visual statement, its parameters and available operations.

To finish this section, it is important to notice that the extensions described are already implemented in the current version of PH-Helper. But if the user needs more functions and properties that are present in the Project Hoshimi but not in the editor, he can extend it[11]. The steps needed for extending functions are:

1. Define the new function in C#.
2. Modify the Function Configuration File (FCF).

Listing 1 DTD for FCF descriptions.

```
<!ELEMENT FUNCTIONS (FUNCTION*)>
<!ELEMENT FUNCTION (PARAMETERS, CODE)>
<!ELEMENT PARAMETERS (PARAMETER*)>
<!ELEMENT PARAMETER EMPTY>
<!ELEMENT CODE (#PCDATA)>
<!ATTLIST FUNCTION name CDATA type CDATA>
<!ATTLIST PARAMETER name CDATA
                    type CDATA>
```

This configuration file (FCF) contains a description of all functions supported by PH-Helper. This description is written in a dialect of XML (also referred to as FCF); the respective DTD is shown in listing 1. In order to add a function, the user only needs to specify a `FUNCTION` element by providing the following elements: `parameters`, `function C# code`, `name` and `type`.

New properties can also be added to the kernel; the following steps describe what is necessary to do that:

1. Define the property or variable in C#.
2. Modify the Properties Configuration File (PVCF).

Similar to the previous case, this configuration file (PVCF) contains a description of all properties and variables supported by PH-Helper. The description is again written in a dialect of XML (named PVCF) with the DTD depicted in listing 2. In order to add a property, the user only needs to specify a `VARIABLE` element by providing the following elements: `property C# code`, `name` , `scope`, `readOnly` and `type`.

[11] The student can define new extensions, but it is advisable that the teacher carries out this task because it is too complex.

Listing 2 DTD for PVCF descriptions.

```
<!ELEMENT VARIABLES (VARIABLE*)>
<!ELEMENT VARIABLE (CODE)>
<!ELEMENT CODE (#PCDATA)>
<!ATTLIST VARIABLE name CDATA
                   scope (global | local)
                   readOnly (yes | no)
                   type CDATA>
```

Listing 3 Examples of Function and Variable extensions to HL.

```
<FUNCTION name="Length" type="Int32">
 <PARAMETERS>
 <PARAMETER name="str" type="String"/>
 </PARAMETERS>
 <CODE>
  return str.Length;
 </CODE>
</FUNCTION>
```

(a)

```
<VARIABLE name="IsAlive"
 scope="local" type="Bool"
 readOnly="yes">
 <CODE>
 {get{return HitPoint > 0;}}
 </CODE>
</VARIABLE>
```

(b)

Listing 3.a shows an example of a function extension. This extension consist in adding the *Length* function. This function has a string parameter called *str* and it returns an integer that represents the length of *str*.

Listing 3.b shows an example of a variable extension. This extension adds a new property called *IsAlive*. This property has a local scope, it is a readonly property and its type is bool.

4 Editing Hoshimi Programs with PH-Helper

To define a strategy using the PH-Helper Main Window, three steps must be carried out (see figure 1):

- *Create a New Program*
- *Create NanoBots Types*
- *Define Strategies*

Create a New Program, is used for creating a new *Working Space*. A *Work Space* is composed by several sub-spaces. Each sub-space represents a NanoBot and it is composed by:

- Toolbar: It is used for doing editing operations, such as: cut, copy and paste, change node's position, etc.
- Language Command Bar: This bar contains all instructions available in the language. These instructions are separated by three logical sections.
  The first one holds all flow control instructions. For example: *state* (an equivalent instruction to case statement), *decision* (an equivalent instruction to if-then-else statement), *repetition* (an equivalent instruction to while statement), etc.
  The second one provides all Project Hoshimi primitive Actions, like: *move*, *stop*, *collect*, etc.
  The last one contains all new commands added by PH-Helper. The most important is the one that allow to save states. It is the variable assignment.
- Program Window: This window visualize the current program. The Hoshimi Programs are naturally represented as a *r-tree*. The tree nodes are classified in:

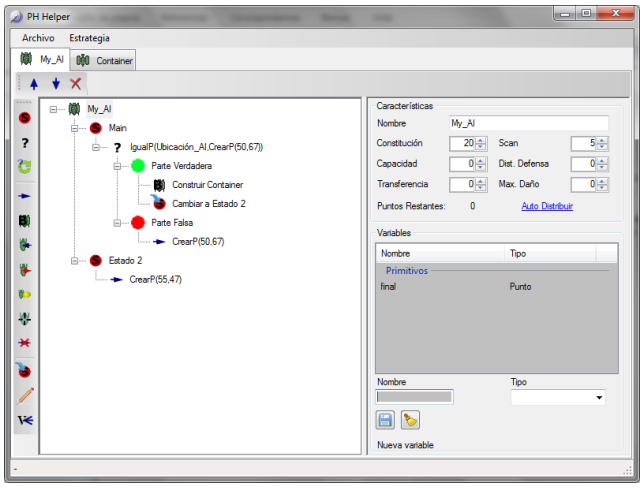


Figure 1 PH-Helper Main Window.

- Leaf Node: This kind of node represents an atomic instruction. For example: *write* (for writing information on the log console), etc.
- Parent Node: This kind of node represents a group of actions. For example: the instruction *if-then-else* has two child node, they are *true* and *false* branches. They in turn are composed by actions. It is important to remark that these actions can be in turn composed by actions or group of actions.

- Parametrization Window: This window is aimed at providing the parameters needed by the selected instruction in the program window. Generally, these instructions are functions with n parameters. In this context, PH-Helper restricts the construction of expressions taking into account the parameter type and the value type returned by the expression. An example is shown in figure 2. This figure shows all data needed for specifying an assignment operation.

  In this case, the Parametrization Window is divided in two parts:

  - Variable: It is the *lvalue* of assignment instruction, and it has two lists:
    1. *Type* is used for filtering the variables according to the selected type.
    2. *Names* is employed for selecting the variable.
  - Value: It is the *rvalue* of assignment instruction, and it has three tabs:
    1. Variable: It shows a lists of variables. The type of these variables is the same that the one selected in the Variable section.
    2. Functions: This tab shows a set of functions whose return type is the same that the one selected in the Variable section.
    3. Constants: This tab allows to specify a literal.

    It is important to notice that if a function is selected in the tab *Function*, the parameter selection process (previously explained) is repeated for each parameter. Otherwise, the process is ended.

Create NanoBots Types is used for creating several roles. A role describes the NanoBot behavior, the number of roles is user-dependent. When a role is created two main activities must be specified:

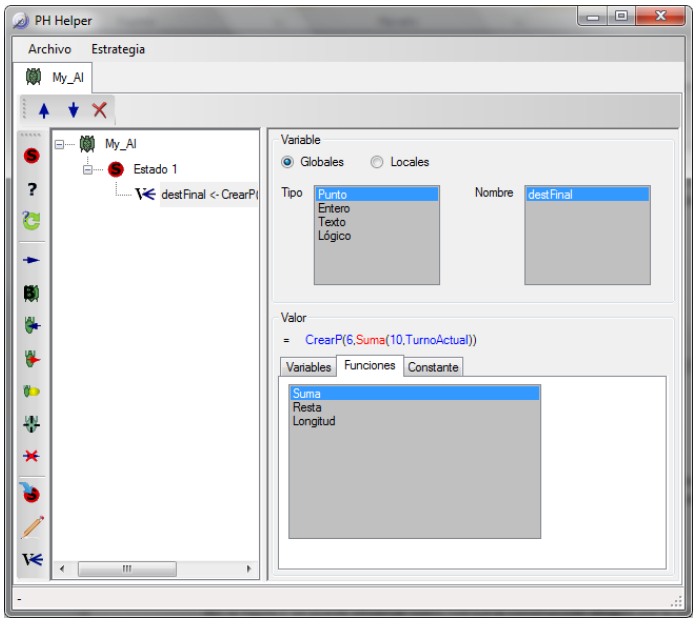


Figure 2 PH-Helper Parametrization Window – Variable Assignment Operation.

- The specification of NanoBot Type: In this activity, the user selects the NanoBot type[12] which can not be changed during the game. The NanoBot type is used for two important goals. The first one sets the available commands, and the second one limits the characteristic's maximum values.
- The specification of NanoBot Characteristics: In this activity, the user gives a value to each NanoBot characteristics[13], which can not be changed during the game.

The activities previously mentioned are easily carried out with PH-Helper. When a NanoBot is created its type is directly established.

The user only needs to select the option *Strategy* and then the option *Add*.

In this moment, a NanoBot type list is shown.

The last task consists in selecting one NanoBot type from the list.

These operations can be observed on the top of figure 3.

After doing these tasks, a new tab is created for the NanoBot (it is a sub-space as described at the beginning of this section). In the Program Window a root node is inserted. This root represents the action for specifying the NanoBot characteristics. The user can establish the characteristic values by following two steps. The first one consists in selecting the root action. The last one consists in changing the characteristic values displayed in the Parametrization Window. The process is illustrated in figure 3, at bottom.

Define Strategies is used to build the action tree. This task is simple to do, the user just needs to add the wished actions by clicking on the *Command Language Bar*. At any moment the user can save the project. This task is achieved by applying the following steps:

1. For each root node r do

[12] Remeber that the available types were described in section 2.
[13] Remember that the characteristics were described in section 2.

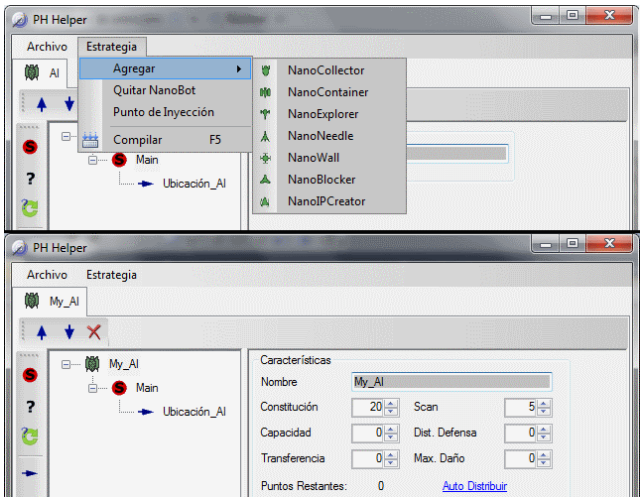


Figure 3 Types and Characteristics available in PH-Helper.

a. For each action A in r do

i. Create a XML element for A. This element contains the A parametrization.

ii. Append the XML code generated in previous step in the XML document.

2. Save the XML document into a file.

The flow chart of this procedure is shown in figure 4.

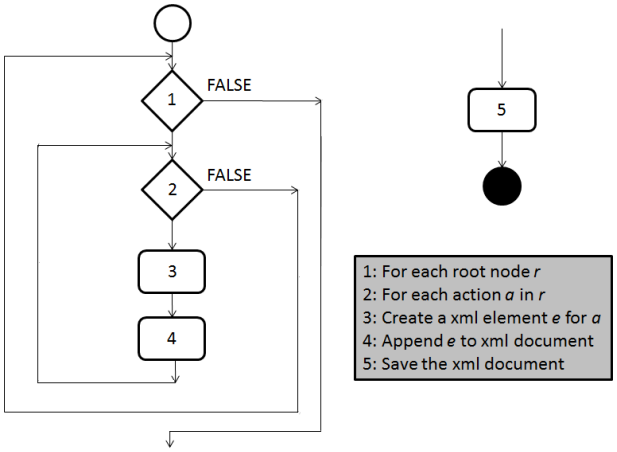


Figure 4 Save Procedure provided by PH-Helper.

The DTD that defines the XML dialect used by PH-Helper to save the edited code is shown in listing 4.

Basically, this DTD allows to define a strategy ia an action set. The actions are specified taking into consideration their parameters. Each action can have sub-actions. In this way a tree structure is built.

Listing 4 XML dialect used to save edited code.

```
<!ELEMENT STRATEGY (ACTION*)>
<!ELEMENT ACTION (VARIABLES, ACTION*)>
<!ELEMENT VARIABLES (VARIABLE*)>
<!ELEMENT VARIABLE EMPTY>
<!ATTLIST ACTION name CDATA
                 tag CDATA>
<!ATTLIST VARIABLE name CDATA
                   type CDATA
                   keyType CDATA
                   valueType CDATA
                   isGlobal CDATA
                   isSystem CDATA
                   isReadOnly CDATA>
```

5 Compiling Hoshimi Programs with PH-Helper

The process for compiling Hoshimi programs is based on the following tasks (a complete view is shown in figure 5):

1. Error Verifying: PH-Helper provides syntax-directed assistance for avoiding errors. However, some semantic errors can even occur. For example, the variable type used in an assignment can be changed. If this happens, a type error arise. Other example is presented when a state referenced in change state operation is deleted. The error sub-system works as follows:
   a. For each action A do
      i. Apply the method *CheckErrors*. In this method, A verifies its parametrization. If it is not correct then an error collection is returned. In other case, an empty collection is returned.
      ii. For each error E recovered in previous step do
         A. Print a message error on *Error Console*.
      iii. If an error is detected, then finish the process.
2. Source Code Storage: This phase is carried out applying the algorithm described below.
   a. For the root action R do
      i. Apply the method *GetCode*. This method is recursively invoked in R sub-actions with the goal of gathering the source code. Each action is responsible of building its proper source code.
      ii. When the strategy source code is collected, it is stored into a file.

The steps described above are applied for each root node in the program. When all the source code components are recovered, they are compiled and the corresponding IL (.Net Intermediate Language) is generated. It is important to remark that some library files are always included in the compilation process, as is the case of:

- MyUtils: this file contains common functions like *DistanceCalculation*, *GetNearestPoint*, *Barycenter*, etc.
- AISystem: this file contains the scheduler and other administration structures.

Furthermore, a project file is also generated. This task is achieved to visualize all files in Visual Studio .Net.

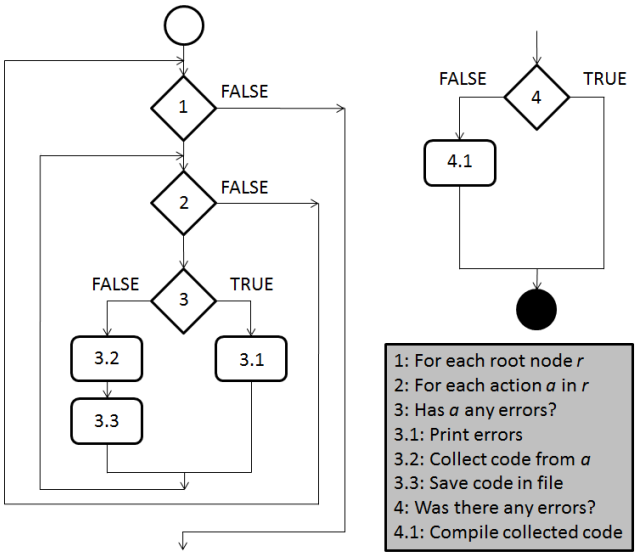


Figure 5 PH-Helper Compilation Workflow.

6 Case Study: The AZN Way

As it was explained in section 2, one of the game objectives is to cure diseases. In order to do this, a NanoNeedle (hereinafter called *needle*) must be created in the diseased zone (Hoshimi Points – HPo). Once created, the needle must be filled with the only resource available in the game: the AZN. Each needle can contain up to 100 units of AZN. It is important to mention that it is convenient to fill the needles until the top, because they provide a score proportional to the AZN stock. Needles can not move or gather AZN by themselves. Because of this, the collaboration between a NanoAI (hereinafter denoted as AI) and a nanobot with gathering capabilities is needed. NanoCollectors (hereinafter called *collectors*) and NanoContainers (also known as *containers*) are two kinds of nanobots that meet this property.

The AIs job is to go to the desired destination (HPo) and build there a needle [14] [15]. Furthermore, the AI must build collectors or containers too. These tasks can be carried out in any order, however it is encouraged to build the gatherers and then the needles. In this way, the firsts can go and gather AZN, while the AI moves to the HPo. Any number of collectors can be built. Nevertheless, it is best to choose a multiple of the amount needed to fill a needle, considering its capacity. If collectors are chosen, is wise to build 5 or 10 of them, because they have a capacity of 20 units of AZN. If containers are chosen instead, is advisable to build a pair number of them (generally between 4 and 10) because they can transport 50 units of AZN.

[14] On the one hand, nanobots with movement capabilities appear in the team injection point when built. On the other hand, nanobots without movement capabilities emerge in the current location of the NanoAI.

[15] Several nanobots with movement capabilities can be at the same point, but just one of them can be active. Because of that, only one needle can be built in a HPo.

6.1 The Problem Statement

In this section, a simple problem is given as exercise with the goal of illustrating the operation of the tool described in this article.

The problem statement is: *Elaborate a strategy for creating and filling 3 needles in the following points: <147, 22>, <159, 26> and <170, 38>*. Tip: It should be noticed that the nearest AZN is at point <154, 54>.

6.2 The Solution

To solve the problem, two nanobots must be added to the AI. One of them must be of NanoNeedle type, and the other one must possess gathering capabilities. Containers will be used in this solution, but the user can choose another type like collectors. The default values of the characteristics are recommended for each nanobot, nevertheless the student can experiment with other values. If these steps were carried out successfully, three tabs will be displayed on the screen. Each one of them is labeled with the name of the represented nanobot. Figure 6 shows the screen previously described.

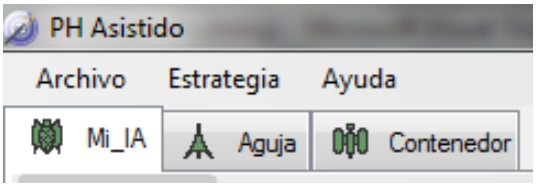


Figure 6 PH-Helper Tabs corresponding to the 3 necessary nanobots created.

In this case, the tabs are labeled Mi_IA, Aguja and Contenedor which are the Spanish words for "My AI", "Needle" and "Container".

Two steps must be specified to define an AI strategy for building a needle at the desired HPo.

First, to save time, it is convenient to build the containers. Generally, the best approach for solving problems is to separate each task into states. In this case, the first state consists in creating the containers. This task is carried out using a conditional statement and a counter. If the counter is less than the number of desired nanobots then a new nanobot will be created; otherwise, the process will finish its execution. It is important to remark that: i) Before adding the actions, the counter must be declared, and ii) The conditional action is executed several times by the Hoshimi Project engine. The complete state and the parametrization of the main action are shown in figure 7.

The next step is to specify the task of moving the AI to the desired location (HPo) and build there a needle. The procedure consists in asking if the current AI location equals the HPo location. If that is the case, the needle is created, and then the AI changes to the next state; otherwise, the AI keeps moving towards the HPo location. The state for building a needle and the predicate parametrization is shown in figure 8.

Two similar states must be specified for the remaining points, with the corresponding HPo locations. The AI responsibilities are performed by all the tasks previously specified.

The needles strategies are very simple because they can not move. The needle can defend by itself but the corresponding strategy will not be implemented because it was not included in the problem statement.

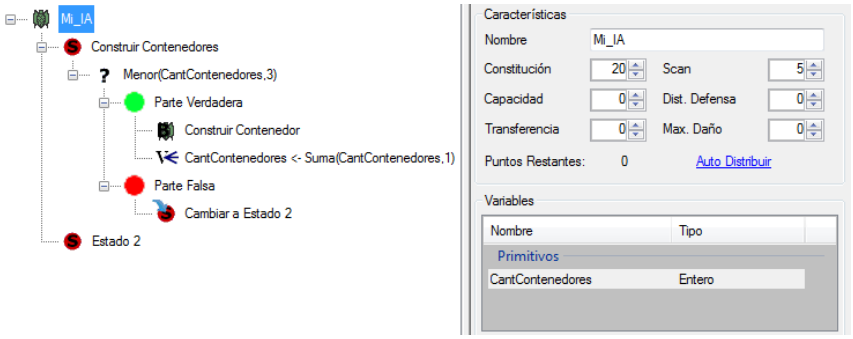


Figure 7 PH-Helper Building a Container.

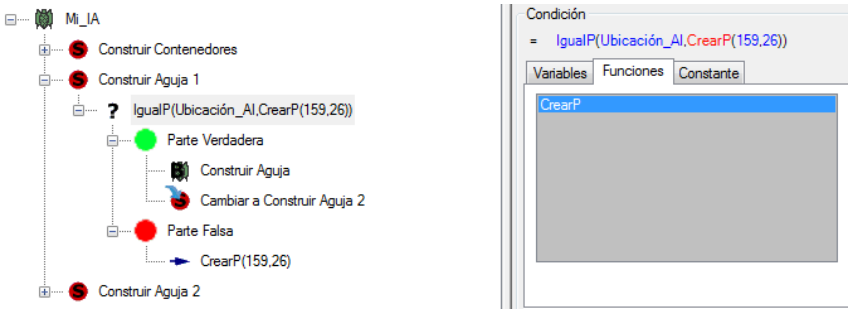


Figure 8 PH-Helper Predicate Parametrization.

The containers strategy has the following tasks:

1. Go to the AZN point.
2. Gather AZN.
3. Go to a HPo with needle.
4. Transfer the AZN.

Clearly, this is translated into four states. The first one simply compares the nanobot location with AZN location. If they are the same, then it changes to the next state; otherwise, the containers keep moving towards the AZN location. The second state commands the container to gather AZN for 10 turns at a rate of 5 units loaded by turn, this action will fill the stock of the container. Then it changes to the next state. The third state is similar to the first, but it uses the HPo location instead of the AZN location. The final state is similar to the second, but it transfers the AZN to the needle.

With all strategies ready, only remains to compile. This process generates an assembly (dll file) that will be used by the Project Hoshimi. An equivalent C# source code is also generated. Part of the source code corresponding to the AI strategy is shown in figure 9, and in figure 10 a snapshot of the strategy execution is displayed.

7 Conclusion

In this paper, a tool aimed at helping to teach programming was presented. This tool is intended to keep students away from those harmful issues that distract them from learning programming concepts (algorithms/strategies and data structures), that is the actual focus

```
switch (___ESTADO)
{
    case "Construir Contenedores":
        if (Menor(CantContenedores, 3))
        {
            AI.Build(typeof(Contenedor));
            CantContenedores = Suma(CantContenedores, 1);
        }
        else
        {
            ___ESTADO = "Construir Aguja 1";
        }
        break;
    case "Construir Aguja 1":
        if (IgualP(Ubicación_AI, CrearP(159, 26)))
        {
            AI.Build(typeof(Aguja));
            ___ESTADO = "Construir Aguja 2";
        }
        else
        {
            MoveTo(CrearP(159, 26));
        }
        break;
```

Figure 9 PH-Helper generated C# Source Code.

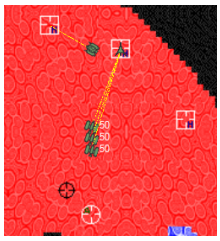


Figure 10 PH-Helper Execution Snapshot.

when solving exercises using the computer. The main difficulties that the tool should keep clear arise from the following facts: i) Programming Languages use idioms different from Programmers' native language, and ii) the syntax of Programming Language is full of tricky details that disturb the quiet writing of programs.

In order to overcome these problems, PH-Helper allows the use of a simple visual language and then generates C# source code. On one hand, the visual language is composed of intuitive icons that can be easily related among them. By intuitive we mean icons that any student, aware of the problem domain, can immediately understand. On the other hand, PH-Helper offers an editing environment that is directed (or guided) by HL syntax, avoiding in this way annoying syntactic errors.

PH-Helper features described above allow the student to be concentrated in solving problems, understanding programming essentials, instead of being bother with technical details.

In order to facilitate programming, HL, the Hoshimi Language, was extended: i) Some statements were enhanced, such as, *loop* and *decision* statements; ii) *Variable definitions* and *Assignment* statement were added; iii) New *functions* and *properties* can also be defined by the programmer.

For each strategy PH-Helper generates C# code that can then be executed in the appropriate .Net environment to test the ideas implemented. Moreover, notice that the generated code is saved into a file and so it can be used to teach object-oriented programming. In this sense, our project also contributes for teaching this important topic, not as easy as we

could forecast in the beginning [9]. Old code generators, mainly those depending directly on the text being freely edit by the programmer (like HTML generators, etc,) produced awful, unreadable code. However we know that recent code generators for formal languages are delivering nice code, plenty of comments, that can actually be used for students to learn with its inspection; we did many experiments in that direction when using compiler generators.

To sum up, we can say that the work here reported was concerned with three challenging and complementary research directions: programming languages design and programming languages implementation towards the improvement of programming courses.

As future work, the research team has scheduled the following tasks: i) To deploy as soon as possible a first PH-Helper stable release to start the practical experiments with real users in real class environments to assess the user-friendliness of the tool and its effective pedagogical aid; ii) To apply the strategies used in PH-Helper development to build similar tools for other domain specific languages. For instance, at UNSL (National University of San Luis), in the first programming course an algorithmic language, similar to natural language, is used for teaching. With this approach, the student can not execute programs. Therefore, the student can not verify his solutions. For the reasons previously mentioned, we plan to develop a visual language with an editor similar to the one described in this paper.

Another interesting topic, for future research, is to plan a set of experiments that can let us to measure and understand how much that approach, of starting programming with DSL s, can effectively help in learning to program properly with GPL s.

References

1 Francisco P. Andrés, Juan de Lara, and Esther Guerra. Domain Specific Languages with Graphical and Textual Views. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *AGTIVE*, volume 5088 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2007.

2 F. Arefi, C.E. Hughes, and D.A. Workman. The object-oriented design of a visual syntax-directed editor generator. In *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 389 –396, sep 1989.

3 Farah Arefi, Charles E. Hughes, and David A. Workman. Automatically generating visual syntax-directed editors. *Commun. ACM*, 33:349–360, March 1990.

4 W. F. Clocksin and Chris Mellish. *Programming in Prolog*. Springer, 1981.

5 M.J. Conway. *Alice: easy-to-learn 3D scripting for novices*. University of Virginia, 1998.

6 Daniela da Cruz, Ruben Fonseca, Maria Joao Varanda Pereira, Mario Beron, and Pedro Rangel Henriques. Comparing generators for language-based tools. In *CoRTA-07 - Compiler Related Techonlogies and Applications, Covilhã, Portugual*, July 2007.

7 Saumya K. Debray. *The SB-Prolog System, version 3.1: A User Manual*. Dep. of Computer Science / Univ. of Arizona, 1.st edition, Dec. 1989.

8 P. Deransart and G. Ferrand. Initiation a prolog: Concepts de base. Support de Cours 86-2, Université d'Orleans, Dep. de Mathématiques et Informatique, Jun. 1986.

9 Stavroula Georgantaki and Symeon Retalis. Using educational tools for teaching object oriented design and programming. *Journal of Information Technology Impact (Jiti)*, 7(2):111–130, 2007.

10 Javier Gonzalez Sanchez, Ramiro A. Berrelleza Perez, and Maria Elena Chavez Echeagaray. Introducing computer science with project hoshimi. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 908–914, New York, NY, USA, 2007. ACM.

11 Pedro Henriques, Maria Joao Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using lisa system. *IEE Software Journal*, 152(2):54–70, April 2005.

12 Christopher D. Hundhausen, Sean F. Farley, and Jonathan L. Brown. Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Trans. Comput.-Hum. Interact.*, 16(3):13:1–13:40, Sep. 2009.

13 A. Hutchinson, B. Moskal, W. Dann, and S. Cooper. The alice curriculum and its impact on women in programming courses. In *Annual Meeting of the American Society for Engineering Education (ASEE06)*, 2006.

14 A. Hutchinson, B. Moskal, W. Dann, S. Cooper, and W. Navidi. The alice curricular approach: A community college intervention in introductory programming courses. In *Innovations 2008, International Network for Engineering Education Research*, pages 157–176, 2008.

15 Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers text editing. In *CHI '05: HUMAN FACTORS IN COMPUTING*, pages 1557–1560. Press, 2005.

16 Andrew Jensen Ko. Designing a flexible and supportive direct-manipulation programming environment. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 277–278, Washington, DC, USA, 2004. IEEE Computer Society.

17 J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education*, 10(4):1–15, 2010.

18 Raul Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1982.

19 MI-students, Daniela da Cruz, and Pedro Rangel Henriques. Agile - a structured-editor, analyzer, metric-evaluator and transformer for attribute grammars. In Luis S. Barbosa and Miguel P. Correia, editors, *INForum'10 — Simposio de Informatica (CoRTA'10 track)*, pages 197–200, Braga, Portugal, September 2010. Universidade do Minho.

20 Barbara Moskal, Deborah Lurie, and Stephen Cooper. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 75–79, New York, NY, USA, 2004. ACM.

21 Ulf Nilsson and Jan Maluszynski. *Logic, Programming and Prolog*. John Wiley & Sons, 1st edition, 1990.

22 K.A. Olsen, P. Harnes, B. Pedersen, and O.-J. Tosse. The dsp system-a visual system to support teaching of programming. In *Visual Languages, 1988., IEEE Workshop on*, pages 199 –206, oct 1988.

23 Seymour Papert. *Mindstorms: children, computers, and powerful ideas*. Basic Books, Inc., New York, NY, USA, 1980.

24 John Peterson. A Language for Mathematical Visualization. In *Proceedings of FPDE'02: Functional and Declarative Languages in Education*, 2002.

25 M. Resnick, Y. Kafai, and J. Maeda. A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities. MIT Media Laboratory, Proposal to National Science Foundation (Information Technology Research), 2003.

26 P. Roussel. *Prolog: Manual de reference et d'utilisation*. Groupe d'Intelligence Artificielle, Marseille–Luminy, 1.st edition, Sep. 1975.

27 Leon Sterling and Ehud Shapiro. *The Art of Prolog*, chapter 16. Series in logic programming. MIT Press, 1986.

28 Tim Teitelbaum. The cornell program synthesizer: a syntax-directed programming environment. *SIGPLAN Not.*, 14(10):75–75, Oct. 1979.

29 Ian Utting, Stephen Cooper, Michael Kölling, John Maloney, and Mitchel Resnick. Alice, greenfoot, and scratch – a discussion. *ACM Transactions on Compututer Education*, 10(4):17:1–17:11, Nov. 2010.

30 Maria Joao Varanda and Pedro Rangel Henriques. Visualization / animation of programs based on abstract representations and formal mappings. In *HCC'01 - 2001 IEEE Symposia on Human-Centric Computing Languages and Environments*. IEEE, September 2001.

31 Daniel Watt. *Learning With Logo*. McGraw Hill, 1983.

32 Molly Watt and Daniel Watt. *Teaching With Logo: Building Blocks For Learning*. Addison-Wesley Pub, 1986.

Problem Domain Oriented Approach for Program Comprehension

Maria João Varanda Pereira[1], Mario Berón[2], Daniela da Cruz[3], Nuno Oliveira[3], and Pedro Rangel Henriques[3]

1 **Polytechnic Institute of Bragança**
Bragança, Portugal
mjp@ipb.pt
2 **National University of San Luis**
San Luis, Argentina
mberon@unsl.edu.ar
3 **Universidade do Minho**
Braga, Portugal
{danieladacruz,nunooliveira,prh}@di.uminho.pt

Abstract

This paper is concerned with an ontology driven approach for Program Comprehension that starts picking up concepts from the problem domain ontology, analyzing source code and, after locating problem concepts in the code, goes up and links them to the programming language ontology.
Different location techniques are used to search for concepts embedded in comments, in the code (identifier names and execution traces), and in string-literals associated with I/O statements. The expected result is a mapping between problem domain concepts and code slices. This mapping can be visualized using graph-based approaches like, for instance, navigation facilities through a System Dependency Graph.
The paper also describes a PCTool suite, Quixote, that implements the approach proposed.

1998 ACM Subject Classification I.2.2 Automatic Programming

Keywords and phrases Program Comprehension, Ontology-based SW development, Problem and Program domain mapping, Code Analysis. Software Visualization

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.91

1 Introduction

Software maintenance is known to be the most time-consuming and expensive phase on the software life cycle [4, 15]. It is incepted by the emergence of new requirements and entails a first phase to comprehend the program and a second to evolve it according to the requirements. Software evolution (once it is comprehended) is a fast task because there is not much expertise involved besides the programming basics. Program comprehension, however, requires more advanced skills where source code analysis techniques play an important role.

Program comprehension theories sprang from cognitive and psychological sciences [36, 34]. Such theories state that cognition is achieved by the construction of a mental model as a structured way of gathering knowledge about the program under analysis. Mental models are constructed either (*i*) top-down [5], i.e., from the knowledge on the problem domain to the knowledge embodied in the program domain, (*ii*) bottom-up [47], i.e., from the knowledge on the program domain to an abstraction capable of being mapped into the problem domain

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 91–105

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

or (*iii*) hybrid [28, 31, 51], i.e., combining the other two, interchangeably. Nevertheless, the majority of these theories agree that a complete understanding of a program is reached when the analyst can relate the program domain—how statements are executed (operational semantics)—with the problem domain—what are the effects caused by the execution of those statements (logical semantics).

Program and problem domains are two sets of knowledge. The former aggregates concepts related with the program, the action of programming and the programming language. The latter gathers discourse-level concepts which are close to the humans' perception because of being fruit of empirical knowledge or expertise in the area where these concepts appear. The knowledge in both domains may be represented in ontologies providing a systematic way of mapping concepts of both domains, and therefore bring them closer.

At the beginning of a program comprehension activity, the problem domain is usually known, i.e., it is possible to identify the main concept in which the program is centered. Related concepts may come from the analyst's empirical knowledge or from the description of a requirement or a task for software maintenance. These descriptions are (usually) short natural language statements performed as a discourse at the problem level. This way, the identification of problem domain concepts is easy and enriches the analyst's knowledge on the problem domain. In a task-oriented approach for software maintenance, such descriptions and the terms involved may be used to focus the analyst's work. The number of concepts involved per task is within what the human brain can handle. Therefore, it is possible to search for them in the source code in desirable time. However these approaches are not always easy to follow and may require more complex solutions. Moreover, tool support for program comprehension is a requirement for automatize, systematize and make effortless the cognitive process.

In this paper we propose an approach and a PCTool suite implementing it, that would allow for a full comprehension of the program slice to maintain and its dependencies. It takes advantage of both ontologies to knowledge representation and task-oriented approach for concept location. We rely on the assumption that the problem domain is known and the comprehension is achieved when the concepts of both program and problem domains are mapped.

The main novelty of our approach is, therefore, the use of ontologies to formally describe the problem and program domains and drive the comprehension by creating a systematic way of mapping the concepts on both domains.

In abstract, we start by collecting the concepts from task or requirement descriptions (henceforth referred to as maintenance statements) using the problem domain ontology. Meanwhile the source code is analyzed to extract information considered important to understand the meaning of each part of the program. Information retrieval techniques (applied upon such information) are used to collect program blocks and program identifiers and associate them with the concepts involved in the maintenance statement. Later, the program ontology concepts are mapped into the previous associations allowing for interpreting the program elements involved in the maintenance task. Finally, the approach is endowed with traditional software visualization techniques for exploration of the provided results. The system is developed as it will be described along the paper.

Section 2 surveys related work, to support the appropriateness of our proposal and some of the choices and decisions taken along the subjacent project (that will be discussed along the paper). A detailed description of the approach is presented in section 3 in a very concise style. In section 4, three developed tools are described. The first two tools perform the static and dynamic source code analysis needed to implement some parts of the proposed

approach. Moreover a third tool for comment analysis is also described in this section. The second part of our program comprehension system is concerned with software visualization and it is described in section 5, right before the conclusion of the paper in section 6.

2 Related Work

The use of ontologies is spread into literature and it appears strongly related with the web semantic area. In our case, we study the use of ontologies in program comprehension.

An ontology is informally described by several authors but the most known is Gruber [21] with the following definition: *An ontology is an explicit formal specification of a shared conceptualization of a domain of interest.* In order to use ontologies in Computer Science it is required to be machine readable, accepted by a community and restricted to a given domain. So, the formal definition of an ontology is based on a set of classes, a set of instances (objects), a set of relations, a set of instances of relations and attributes (object properties) [13]. Several representation languages can be used to describe an ontology: RDF, RDF-schema, OWL.

Ontology learning (automatic, or semi-automatic support in ontology development) and ontology population (the process of defining and instantiating a knowledge base) are the main activities concerned about the use of ontologies [7, 6]. Following Jinsoo Park [25], data resources may be textual data, dictionaries, knowledge bases, semi-structured schemata or relational schemata. As information extraction techniques, the same authors refer natural language processing, statistics and information retrieval. There are also some authors that infer ontologies based on the comparison of other ontologies from the same domain [9]. So, there are lots of research work on learning ontologies from texts, where text mining (e.g. text categorization, text clustering, concept/entity extraction, production of granular taxonomies, sentiment analysis, document summarization, entity relation modeling) is the basic technique [25]. All these activities consist on collecting, selecting, grouping, classifying the words extracted from the data resources, mapping them to concepts and analyzing a set of possible relations. There are several tools for ontology extraction [25] like: OntoLT, Text2Onto, OntoBuilder, Doddle-owl, asium, soat, vetlan, Mok Workbench. Most of these tools require the hands of a domain expert, operating, then, as semi-automatic.

In our case and at a first phase we consider that the problem domain ontology is already constructed and we want to locate these concepts on source code trying to map them to the program domain ontology. So, we need to explore deeply concept location techniques. The most important work in this area is presented by Vaclav [46, 32]: *The important task is then to understand where and how the relevant concepts are implemented in the code.* The techniques can be based on a top-down process—it consists on analyzing the domain to discover its concepts and then trying to match parts of the code with these concepts—or in a bottom-up process—it consists on analyzing the code and trying to cluster the parts that are most closely related according to a certain criteria. However, the most common is a combination of both [37]. The techniques presented by Vaclac are based on string pattern matching (grep), searching through the static code following call graphs and software reconnaissance using code instrumentation in order to discover the parts of the program that are related with each concept. Other technique is concerned with use of an information retrieval system: for instance, the LSI (Latent Semantic Indexing) [33, 41, 12] method that is based on queries to map external documentation to code. The work of Freitas described in [17] uses this kind of techniques to identify the comments associated to each problem domain concept.

Other authors systematically transform program identifiers into fragments of natural language sentences and then check whether the sentences are meaningful for humans using Google web search engine [10].

Fry [19] concludes that there are many natural language clues in program literals identifiers and comments. Natural language analysis of source code complements traditional program analysis. So, the idea is to use algorithms to automatically extract verb information from program source code comments and methods signature using concept location techniques.

The works described along this section were considered and influenced the design and decisions behind our development proposal. Full program comprehension tools—such as Alma [8], Alma2 [40], CodeCrawler [27], DA4Java [44], JIRiSS [45], Cerberus [14], Rigi [35], SHriMP [49, 48], SHriMP with Creole [30]—described in our previous works [3] or [39]) were also taken again in consideration, but the proposed ontology-based philosophy turns the present approach into a novelty.

3 An approach for Program Comprehension

The Program Comprehension (PC) approach we advocate in this paper is:

- **Problem Domain oriented**, because the PC process always starts with the terms involved in the description of the maintenance task to accomplish (a natural language statement, written as a discourse at the problem level); the idea is to locate these terms in the program, actually in its source code.
- **NL supported**, because maintenance task terms, before being located in the code, are first located in the *natural language sentences* (`NL-strings`) embedded in the source code, this is, in the *comments* and in the *text-messages* that are included in the input/output (`I/O`) statements in the form of string literal.
- **Ontology driven**, because ontologies are used to represent the knowledge that characterize both the Problem domain and the Program domain. This means that the maintenance task terms, we want to look for, should be formalized as *concepts* or *relations* belonging to the Problem Ontology (PrbO); when located in the program source code, these terms should be interpreted according to the *concepts* and *relations* in the Program Ontology (PrgO), an extension of the Programming Language Ontology (PLO).

After this short characterization, we can describe our proposal more precisely decomposing the approach into the following steps:

- read the maintenance task statement (a natural language sentence) and identify in the PrbO the terms (concepts and relations) that rigorously describe that task (sketch in Figure 1);
- pick up these terms from the PrbO and use an Information Retrieval (`IR`) engine to search for all `NL-strings` (*comments* or *`I/O` string literals* where the terms, or similar terms[1], appear more frequently (sketch in Figure 2);
- go through the retrieved `NL-strings` read them and choose the code chunks associated with the `NL-strings` that best fit the maintenance needs; use static or dynamic graphs (like the System Dependency Graph, or Execution Trace Graphs) to help on that choice (sketch in Figure 3);
- select the identifiers contained in the retrieved code chunks that are more similar to the terms used in the initial search (sketch in Figure 4);

[1] Dictionaries or Thesaurus shall be used to look for synonymous or translations.

- use the PrgO to interpret these identifiers, recognizing in this way the program elements involved in the maintenance task (sketch in Figure 5);
- use traditional PC techniques—like code visualizations and animation tools—to help in a deeper analysis of those code chunks to get a better understanding of them (sketch in Figure 6).

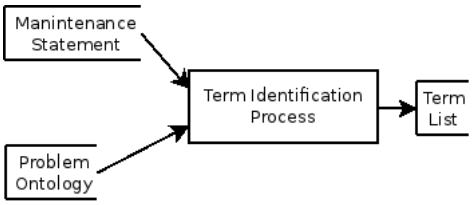


Figure 1 Term Process Identification Sub-system.

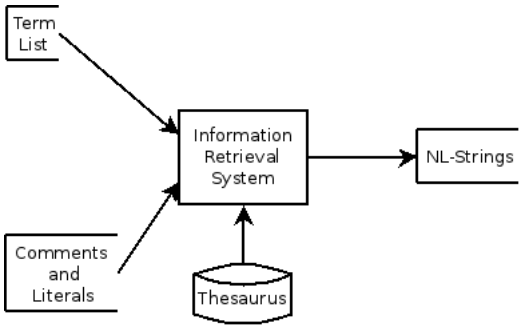


Figure 2 Information Retrieval Sub-system.

Figure 7 depicts the PC approach proposed summarizing the detailed list of steps above described.

To start the implementation of Quixote, we developed some tools combining several approaches and techniques for source code analysis and for concept location using ontologies. These tools will be described in the next sections, identifying the step where each one should be integrated.

4 Source code analysis tools

Obviously the *source code* is the first information resource to be considered for understanding a program.

In this section, a set of tools for analyzing `Java` source code is described. Each tool extracts static or dynamic information from the source code and applies different strategies to enable both the inspection and the comprehension of the software system under study.

Usually, the techniques employed in this context are centered in analyzing declarations and statements to identify data structures and control-flow, building representations such as: data/control-flow graphs, function-call graph, module dependency graph, etc. Strategies for analyzing natural language information sources, for instance comments and literals associated with I/O statements, are not commonly considered. However these program elements

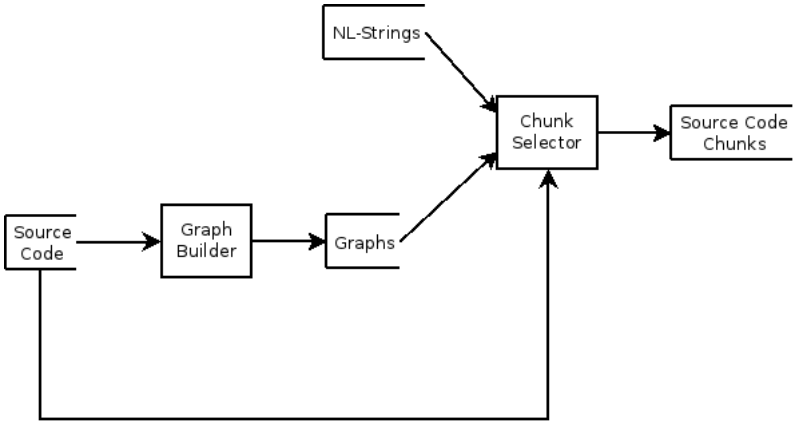


Figure 3 Chunk Selector Sub-system.

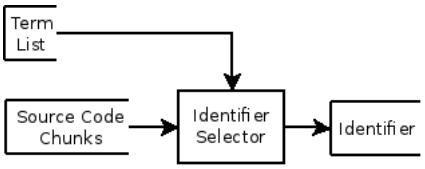


Figure 4 Identifier Selector Sub-system.

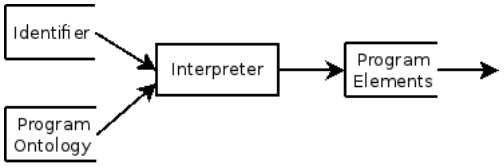


Figure 5 Interpreter Sub-system.

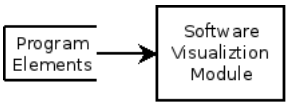


Figure 6 Visualization Sub-system.

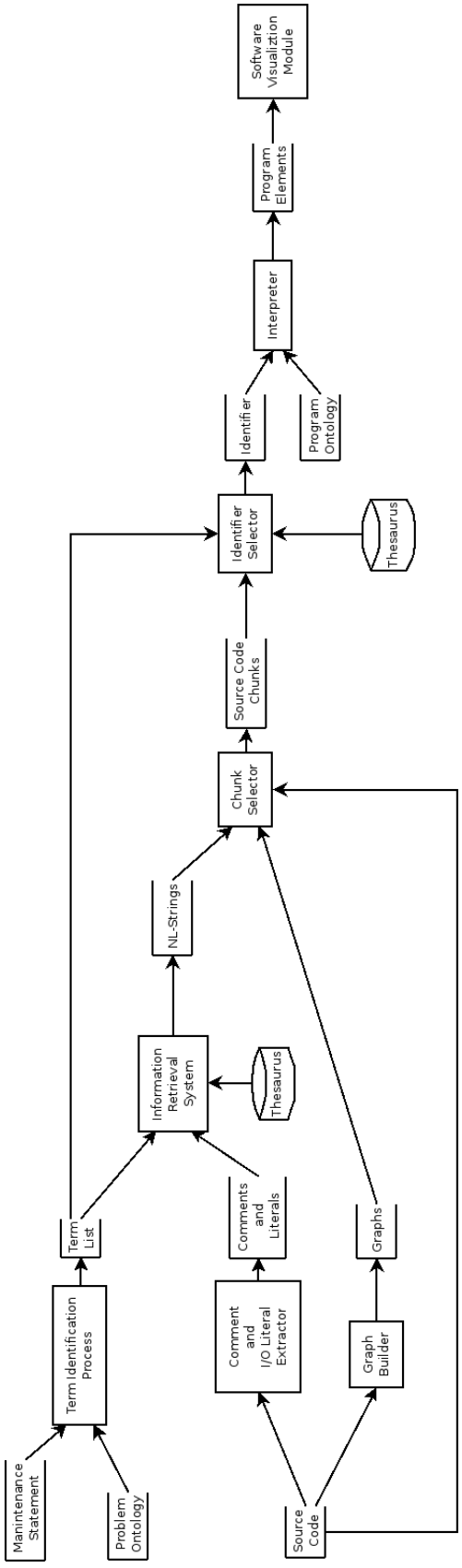


Figure 7 Program Comprehension Approach.

are important because they can provide information related with the problem domain, and it is well known that discover the concepts and relations of that domain is a hard task.

The tools that will be described in the following subsections were developed considering the arguments above mentioned.

4.1 Static Source Code Analysis

The first tool implements an innovative approach to work with static system information.

This approach builds as usual a parser tree and a rich symbol table where all the identifiers extracted from the code are associated with their context (scope and code block) and with their role or class according to the concepts defined in the Programming Language Ontology (PLO); in this way a Program Ontology (PrgO) is built. The PrgO is a useful internal representation of the knowledge inferable from the source code at a programming level (program domain). It allows the user to navigate through the programming language concepts down to their instances (identifiers) in the code, or from the instances up to the programming concepts.

In parallel we are working over the set of detected identifiers exploring them lexically and morphologically in order to enable the discovery of similarities with the concepts of the problem domain. For that purpose, several techniques are used: splitting identifiers into terms; expanding terms; and looking for translations, synonyms or Word_net meanings.

In this case, `AnTLR` [42], a well known compiler construction tool, was used to parse the code and to extract program identifiers (names of classes, interfaces, methods, variables, etc), but other tools like `LISA` [23], `Eli` [20], `JavaCC` [26], `CoCo/R` [24], `lex/yacc` [29] could have been used for the same purpose.

This tool will be used in steps, *Identifier Selector Sub-system* and *Interpreter Sub-system*, shown in Figures 4 and 5, in order to aid in the interpretation of the identifiers according to the PrgO. Till the moment this part of the system copes with C and Java programs.

4.2 Dynamic Source Code Analysis

The second tool is based on dynamic information [22]. Code Instrumentation (`CI`) is a strategy for gathering this kind of information. `CI` consists of inserting *specific sequences of statements* (*inspectors*) in strategic locations of the source code with the goal of capturing the system behavior. This task is performed automatically by parsing the source program. In [3] a tool called PICS was developed to inspect C programs using exactly the same methodology. Now we are interested to cope with object-oriented code.

JDIE (Java Dynamic Information Extractor) is a tool aimed at extracting dynamic information from `Java` programs using `CI`. In order to carry out this task, two factors must be taken into consideration:

1. The information to extract;
2. The source code places where this information can be recovered.

Considering the first factor, JDIE extracts information related with *methods*. This information is useful because it allows to detect which were the methods actually employed for a specific scenario. Concerning the second factor, the places selected are: the *beginning* and the *end* of each system method. With the observations previously mentioned, the instrumentation was carried out by inserting the *inspectors* at the beginning and at the end of each system method. These *inspectors* record the name of the method when its execution starts and finishes. This simple approach, that is effective for imperative languages like *C*, rise some limitations when applied to an object-oriented language like `Java`. Methods can

finish with *return* statements that contain invocations to other methods inside them. In this case, the strategy does not work because the execution traces recorded are not correct. The solution to this problem is carried out by applying the following transformation to each *return* statement:

1. Create a local variable, *rvalue*, with the following characteristics:
   a. The type of *rvalue* is the method type (i.e., the type of the value returned by that method);
   b. The value of *rvalue* is determined by assigning to it the expression found inside the *return* statement.
2. Modify the statement *return* replacing the expression inside its body by the variable *rvalue* defined in previous step.

This approach works well. However, several problems appear when the methods have loops. Observe, the source code shown below.

```
for (i=0; i<30.000; i++) {
  g(x);
  h(x);
}
```

The instrumentation strategy will inform 30.000 times that function g begins and ends its execution. Similar information will be recorded 30.000 times for function h. In other words, a huge amount of information will be produced. One possible way for solving the problem presented by loops consists in the use of a stack. This stack contains in its top a value that indicates the number of times that the functions used inside the loop shall be registered. This stack is necessary because the iterations can be nested. The value is decremented at the end of each iteration. When it is zero, no more information concerned with this method will be registered. When the loop finishes its execution, the top of the stack is deleted.

To sum up, the instrumentation strategy can be fully described by the following algorithm: For each method M found in the source code, do:

1. Insert an extraction statement at the beginning of M.
2. For each statement *return* found in M, do:
   a. Create a local variable, *rvalue*, with the following characteristics:
      i. The type of *rvalue* is the method type (i.e., the type of the value returned by that method);
      ii. The value of *rvalue* is determined by assigning to it the expression found inside the *return* statement.
   b. Modify the statement *return* replacing the expression inside its body by the variable *rvalue* defined in previous step.
3. Insert an extraction statement at the end of M.

The reader can see [1] for more details about the instrumentation scheme described above.

The result of this instrumentation work is a sequence of function calls. Usually trace summarization techniques must be used in order to reduce the amount of extracted information. Over that summarized information several analysis are performed—graph based analysis (for instance, to discover sequence of calls) and identifier analysis—aimed at understanding the behavior of the code under study.

Besides the classic use of this instrumentation tool, there is the possibility of relating the traces extracted (graphs) with the identifiers in the PrgO, upgrading both. Moreover, the dynamic traces can be used in conjunction with the module for concept location in strings

(comments and literals), aiding the user choosing the sequence of strings to read and relate to code.

This tool will be used in the step of our proposal that is described in Figure 3, Chunk Selector Sub-system, in order to construct the graphs. Additionally static and dynamic analysis can help to infer the actual types of input/output data in generic programs that will also be useful in program comprehension tasks.

4.3 Comment Analysis

Comments are interspersed by the Programmer among code lines, at software development phase. They are scattered all over the source code, sometimes wrapping a block of code (placed at the beginning or at its end), other times complementing a single statement. It is important to remember that comments are inserted by a programmer with two main purposes: to help himself during the development phase (and in this case they are not too much useful); to help other programmers, at the maintenance phase, in understanding his ideas. In that case, comments will contain, for sure, concepts associated with the problem domain, and they will be very relevant for PC tools.

The third tool we want to introduce, Darius, is responsible for locating automatically concepts in comments extracted from source code. The approach adopted is aimed at finding a relevant code chunk[2] using information retrieval techniques to locate problem domain concepts within comments. Darius extracts all the comments and classifies them per type (inline, block or JavaDoc comment), keeping their context, i.e., the code lines before/after the comments.

Picking up concepts from the ontology that describes the problem, it is possible to find all the comments that contain that concept (similar words) and rate them. Reading comments from the retrieved list, the programmer can select those that seem to him meaningful and dive directly into the associated chunk. Our idea, building this tool, is not to analyze comments to understand the associated statements. In the other way around, we propose to locate problem domain concepts on comments, and then identify the relevant code chunks associated with them. This approach is based on the ontology that describes the problem domain.

Darius [17] is precisely a tool developed to corroborate the previous statement.

In order to address the *concept assignment problem*, Darius follows standard IR strategies many times referred in the literature. To accomplish its task, we built it up from the following components:

- the document database builder that constructs the logical view of the documents (that in this case are the source program comments) and stores all of the relevant information associated with the comments;
- the IR engine (actually two models are available) that explore the information of the document database, and retrieve documents according to the queries the users provide;
- the graphical interface that provides all the interaction with the system and displays the results of finding Problem Domain concepts using comment information.

All the technical details regarding the development of each one of these components can be seen in [17][3].

---

[2] The code block where the programmer should focus the attention for some software maintenance purpose.

[3] This thesis is available at the URL `http://www3.di.uminho.pt/~gepl/QUIXOTE/FreitasJL2011thesis.pdf`.

Moreover, Darius provides some features to study the frequency of comment occurrences in the source files of a given project (for more details, see [18]). The data extracted and the measurements performed allow the user to compute some statistical information that is useful to verify if the source code contains enough comments that make this PC approach worthwhile[4]. Darius identifies and extracts *inline*, *block* and *javadoc comments* and provides some metrics useful to appraise the commentary policy followed by the program authors.

Clearly, this tool will be incorporated in step, *Information Retrieval Sub-system*, shown in Figure 2, in order to aid in locating problem concepts in the comments.

5 Visualization

To build *S*oftware Visualization components (for the sake of simplicity represented in Figure 6 as just one module in the last step), visualization techniques—concerned with scaling, drawings, coloring and processing speed—for graphs (flowcharts, function graph, module graph, system dependency graph, etc.), for trees (tree function, treemaps, and so on), for textual representations (source code structure or emphasized blocks, program metrics), for graphics, and animations will be fully exploited. Easy to use and advanced navigation features enabling the interconnection among all the above described visual artifacts and the interaction with the user, will be included to aid in the location of the code chunks and in the mapping between domains.

To produce these different visualizations, and address all the described features (actually not only concerned with the final results and restricted to the last process phase, but spread out all over the system), we rely on our strong background in this field. Theoretical works like the thesis [43, 11, 3, 38, 8] gave the support to the applications developed under the program comprehension project PCVIA [50]. Tools like Alma [8], Alma2 [40], PICS [2] and WebAppViewer [16] provide graphical features to expose information of both program and problem domains.

In Quixote PCTool suite, we plan to create visualizations for the three ontologies — PrbO, PLO, PrgO— as well as for the program source code, and the system dependency graph. Adequate navigation techniques among them will allow the visualization of the envisaged mapping.

6 Conclusion

It is well known that a programmer understands programs when he can relate the problem and program domains. Nevertheless, it is recognized that such relation is difficult to reach and build. Having present this inconvenient, and with the purpose of providing a new solution to this challenge, several approaches in the literature were analyzed. From that study it came out that basically all of the approaches rely on the use of static and dynamic information to build program representations and to infer problem concepts.

Strategies, based on knowledge representation techniques enabling semantic directed manipulation for precise definition of the concepts used in both the problem and program domains, are rare in the literature.

In this paper a novel ontology-based approach for Program Comprehension was presented. Here, ontologies allow for a precise description of a domain in terms of its concepts and relations.

[4] Notice that this approach has no meaning if source code is not interleaved with comments.

The approach mentioned above uses four inputs for mapping problem domain concepts into system source code. The first one is the Problem Domain Ontology. It describes the problem concepts and the relations between them. The second one, Maintenance Statement, is used for identifying the problem domain concepts involved in a maintenance task. The third one, Source Code, is an important information resource employed by several tools concerned with the extraction of identifiers and concepts location. The fourth one, Program Ontology, is used to take advantage of the semantic provided by the relation between the program elements linked with programming language ontology.

The program domain ontology (PrgO) is built automatically and is based on the programming language ontology (PLO) and on the source code. The PLO is previously developed and by default incorporated in our PCTools suite.

The information provided by the inputs previously mentioned are processed for: (*i*) linking the problem concepts with chunks of source code that implement them and (*ii*) providing several views for helping the programmer to understand there behavior. The processing referenced before implies: (*i*) extraction of static and dynamic information and (*ii*) interpretation of the information gathered, in order to understand how the system works.

On the one hand, the information extraction is carried out by applying several techniques of static and dynamic analysis and strategies of natural language processing. On the other hand, the interpretation phase is accomplished using the program ontology and identifier list appropriately filtered. At the end of the processing phase, it is possible to visualize, from several perspectives, the chunks of code that implement the problem domain concepts involved in the maintenance task. As a final remark, the approach described makes a deeper exploration of information of both the problem and program domains for reaching clear and robust relations between them.

References

1 Hernan Bernardis, Daniel Riesco, Carlos Salgado, Mario Beron, and Pedro Rangel Henriques. Analisis dinamico para la creacion de estrategias de comprension de programas. In *WICC 2012 - XIV Workshop de Investigadores en Ciencias de la Computacion, Misiones, Argentina*, April 2012.

2 M. Berón, P. Henriques, M. Varanda, and R. Uzal. PICS un sistema de comprensión e inspección de programas. *Congreso Argentino de Ciencias de la Computación CACIC 2007*, 13:462–473, 2007.

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

4 Barry W. Boehm. *Software Engineering Economics.* Prentice Hall PTR, 1981.

5 Ruven E. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554, Nov. 1983.

6 P. Cimiano, A. Hotho, and S. Staab. Learning concept hierarchies from text corpora using formal concept analysis. *Journal of Artificial Intelligence Research*, (24):305–339, 2005.

7 Philipp Cimiano. *Ontology Learning and Population from Text.* Springer-Verlag New York Inc., 2010.

8 Daniela da Cruz, Pedro Rangel Henriques, and Maria João Varanda Pereira. Constructing program animations using a pattern-based approach. *ComSIS – Computer Science an Information Systems Journal, Special Issue on Advances in Programming Languages*, 4(2):97–114, 2007.

9 Jan Jurjens Daniel Ratiu, Martin Feilkas. Extracting domain ontologies from domain specific. 2008.

10 Lars Heinemann Daniel Ratiu. Utilizing web search engines for program analysis. *18th IEEE International Conference on Program Comprehension*, 2010.

11 Eva Ferreira de Oliveira. Caracteristicas de um sistema de visualiza cão para compreensão de programas web. Master's thesis, University of Minho, Sep. 2006.

12 S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

13 Lucas Drumond and Rosario Girardi. Extracting ontology concept hierarchies from text using markov logic. In *SAC*, pages 1354–1358, 2010.

14 Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society.

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

16 Ruben Fonseca, Daniela da Cruz, Pedro Henriques, and Maria Jo ao Varanda Pereira. How to interconnect operational and behavioral views of web applications. In IEEE, editor, *ICPC'08 - 16th International Conference on Program Comprehension*. Amsterdam, Holanda, June 2008.

17 José Luís Freitas. Comments Analysis for Program Comprehension. Master's thesis, Dec 2011.

18 José Luís Freitas, Daniela da Cruz, and Pedro Rangel Henriques. The role of Comments on Program Comprehension. In Luis Caires and Raul Barbosa, editors, *INForum'11 — Simpósio de Informática (CoRTA'11 track)*, Coimbra, Portugal, September 2011. Universidade de Coimbra.

19 Z. P. Fry, D. Shepherd, E. Hill, L. Pollock, and K. Vijay-Shanker. Analysing source code: looking for useful verb-direct object pairs in all the right places. *Natural Language in Software Engineering, IET Software*, 2(1):27–36, 2008.

20 R. Gray, V. Heuring, S. Kram, A. Sloam, and W. Waite. Eli: A complete, flexible compiler construction system. Research report, Univ. of Colorado at Boulder, Oct. 1990.

21 T. Gruber. Towards principles for the design of ontologies used for knowledge sharing. *L. of Human and Computer Studies*, 43:907–928, 1994.

22 Bernardis H. Instrumentacion de programas escritos en java para interconectar los dominios del problema y del programa. In Universidad Tecnologica Nacional, editor, *40 Jornadas Argentinas de Informatica e Investigacion Operativa. 40 JAIIO. Concurso Estudiantil. EST 2011.*, volume 40, 2011.

23 Pedro Henriques, Maria Joao Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using lisa system. *IEE Software Journal*, 152(2):54–70, April 2005.

24 Pedro R. Henriques and Jose Joao Almeida. O Gerador de Compiladores COCO. Relatório de instalação, G.D. Ciências da Computação, D.I./ Univ. Minho, Mar. 1990.

25 Sangkyu Rho Jinsoo Park, Worchin Cho. Evaluating ontology extraction tools using a comprehensive evaluation framework. *Data&Knowledge Engineering*, 69:1034–1061, 2010.

26 Viswanathan Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. *IEEE Software*, 21:70–77, 2004.

27 Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

28 S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *Software, IEEE*, 3(3):41–49, 1986.

29 J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. Ed. Dale Dougherty. O'Reilly & Associates Inc., 1992.

30 Rob Lintern, Jeff Michaud, Margaret-Anne Storey, and Xiaomin Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2003. ACM.

31 David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *J. Syst. Softw.*, 7(4):341–355, 1987.

32 Radu Vanciu Maksym Petrenko, Vaclav Rajlich. Partial domain comprehension in software evolution and maintenance. *16th IEEE International Conference on Program Comprehension*, 2008.

33 Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. *WCRE 2004 - 11th IEEE Working Conference on Reverse Engineering*, 2004.

34 Thomas P. Moran and Stuart K. Card. Applying cognitive psychology to computer systems: A graduate seminar in psychology. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 295–298, New York, NY, USA, 1982. ACM.

35 H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

36 Ulric Neisser. *Cognitive Psychology*. Appleton-Century-Crofts, New York, 1967.

37 Timothy Lethbridge Nicolas Anquetil. Extracting concepts from filenames. 1998.

38 Nuno Oliveira. Improving program comprehension tools for domain specific languages. Master's thesis, University of Minho, Braga, Portugal, October 2009.

39 Nuno Oliveira, Pedro Rangel Henriques, Daniela da Cruz, Maria João Varanda Pereira, Marjan Mernik, Tomaž Kosar, and Matej Črepinšek. Applying program comprehension techniques to karel robot programs. In *Proceedings of the International Multiconference on Computer Science and Information Technology - 2nd Workshop on Advances in Programming Languages (WAPL'2009)*, pages 697–704, Mragowo, Poland, October 2009. IEEE Computer Society Press.

40 Nuno Oliveira, Maria João Varanda Pereira, Pedro Rangel Henriques, and Daniela da Cruz. Visualization of domain-specific program's behavior. In *Proceedings of VISSOFT 2009, 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 37–40, Edmonton, Alberta, Canada, September 2009. IEEE Computer Society.

41 Pocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. On the equivalence of information retrieval methods for automated traceability link recovery. *18th IEEE International Conference on Program Comprehension*, 2010.

42 Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

43 Maria João Varanda Pereira. *Sistematização da Animação de Programas*. PhD thesis, University of Minho, Nov. 2003.

44 M. Pinzger, K. Grafenhain, P. Knab, and H. C. Gall. A tool for visual understanding of source code dependencies. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 254–259, 2008.

45 Denys Poshyvanyk, Andrian Marcus, and Yubo Dong. Jiriss - an eclipse plug-in for source code exploration. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 252–255, Washington, DC, USA, 2006. IEEE Computer Society.

46 Vaclav Rajlich and Norman Wilde. The role of concepts in program comprehension. *IWPC-10th IEEE International Workshop on Program Comprehension*, 2002.

47 Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Parallel Programming*, 8(3):219–238, Jun. 1979.

48 Margaret-Anne Storey. Designing a software exploration tool using a cognitive framework of design elements. In Kang Zhang, editor, *Software Visualization: From Theory to Practice*, pages 113–148. Springer, 2003.

49 Margaret-Anne Storey, Casey Best, Jeff Michaud, Derek Rayside, Marin Litoiu, and Mark Musen. SHriMP views: an interactive environment for information visualization and navigation. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 520–521, New York, NY, USA, 2002. ACM.

50 Maria João Varanda and Pedro Henriques. Program comprehension by visual inspection and animation.

51 A. von Mayrhauser and A. M. Vans. From program comprehension to tool requirements for an industrial environment. In *In Proceedings of IEEE Workshop on Program Comprehension*, pages 78–86, 1993.

The Impact of Programming Languages in Code Cloning

Jaime Filipe Jorge[1] and António Menezes Leitão[2]

1 **Instituto Superior Técnico**
Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
jaime.f.jorge@ist.utl.pt
2 **Instituto Superior Técnico**
Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
antonio.menezes.leitao@ist.utl.pt

Abstract

Code cloning is a duplication of source code fragments that frequently occurs in large software systems. Although different studies exist that evidence cloning benefits, several others expose its harmfulness, specifically upon inconsistent clone management.

One important cause for the creation of software clones is the inherent abstraction capabilities and terseness of the programming language being used.

This paper focuses on the features of two different programming languages, namely Java and Scala, and studies how different language constructs can induce or reduce code cloning. This study was further developed using our tool Kamino which provided clone detection and concrete values.

1998 ACM Subject Classification K.6.3 Software Management

Keywords and phrases Clone Detection, Software Engineering, Programming Languages, Software Management

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.107

1 Introduction

There are a number of bad smells in the process of writing software systems[1]. *Code cloning* is one of them. A code clone is a duplication of source code where programs are copied from one location and subsequently pasted onto other locations.

Studies found that code cloning is ubiquitous in software systems[2][3][4]. Some authors consider cloning to be harmful to software development [2][3][5][6]. A study[7] reports that in some systems half of the changes to code clone groups are inconsistent and that corrective changes following inconsistent changes are rare.

Its existence may be due to a number of reasons. Clones may be caused by programmer inexperience or programming methodology. In this paper we will focus in programming language constructs as causes for code cloning.

Subject to our study are two programming languages: Java and Scala. The Java programming language is one of the *de facto* standards for writing object-oriented systems. Its language constructs are result of years of best practices and it is still passing the test of time by having a large of companies using it. We will provide insights as to how these language constructs influence code cloning.

Scala is a more recent statically typed JVM language that provides different abstraction mechanisms, which are a result of learning from Java's wrong and right doings.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 107–122

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

We provide clone analysis using these two programming languages and we will conclude how abstraction capabilities and verbosity/terseness influence code cloning. This study originated from the development of our tool, Kamino, which is a software clone evolution analyzer that provides developer with means to acknowledge clone duplication, observe and mitigate its consequences.

The rest of the document is structured as follows: section 2 describes software clones and programming language impact, section 3 describes a case study where we instantiate some of the Java problems, section 4 describes in detail some of the problematic situations which fuel software clones and how they can be mitigated using Scala, section 5 will map the scenarios described earlier to real values obtained using our tool Kamino and section 6 presents our conclusions.

2 Software Clones

A code clone is a duplication of source code in which fragments are copied from one location and subsequently pasted onto other locations, which can then be modified or not. Upon duplication, it is often difficult to differentiate the original fragments from duplicated ones, hence the two fragments are uniquely called clones.

Studies show that cloning is ubiquitous in software systems. For instance, Baker et al.[2] found that, on large systems, 13% to 20% of source code can be cloned code. Baxter et al.[3] reported 12.7% source code clones whereas Mayrand et al.[4] experienced 5% to 20%.

Well known software systems have been subject to clone study. In terms of their percentage of code cloning, GCC averages 8,7%[8], X Windows has 19%[2] whereas Linux and JDK have , respectively, 22,7% and 29%[9].

From the developers perspective, there are a number of factors which propel code cloning. On the one hand, we can think of limitations inherent to the programmer himself. For example, when he needs to work with a new system. Whether because it is a large system (which induces difficulty in mastering its domain) or the programmer is dealing with code which he does not own, both aspects tend to promote a kind of example-driven programming approach, in which developers copy fragments of existing functionality and modify them to solve the problem at hands, i.e., each developer's task. Logically, this methodology enables clone propagation.

In what regards programming methodology, there are also reasons which motivate cloning. Systems that rely heavily on generated code (generative programming), tend to have higher clone numbers, since they actually enforce a template which can be seen as a framework of a clone.

Programming languages can ultimately induce code redundancy. Less expressive languages promote code cloning by requiring greater verbosity and need for boilerplate in programming. This code verbosity may hinder the original developers intent. Other problem related to programming languages are lack of reusing mechanisms. Examples of such mechanisms are inheritance composition, generic functions, higher order functions, macros, type classes, etc. Upon nonexistence of such mechanisms, programmers are motivated to copy and paste code.

Software duplication has a function of cost, i.e. , implementing a refactored alternative to a cloned code fragment is not always clearly beneficial [10]. This may be caused by *difficulty in creating or understanding the refactored version* or by *unwanted size increase that it would imply in the system*. Both of these causes can be correlated to the original premise that programming languages may induce cloning namely because refactoring may be complex to

create/understand and may create larger versions of the original cloned fragments.

Hence, we can also create the parameters on which we can grade programming languages, based on their capabilities towards mitigating code cloning:

Abstraction Capabilities This criteria judges the idiomatic and simple nature of abstraction possibilities to given cloned fragments in the programming language itself. It may be possible to create such abstraction constructs but they may not be valuable in the cost function described above (namely in complexity in construction and understanding). These abstraction capabilities may include method extractions, inheritance, mixins, high-order functions or anonymous inner classes, duck typing, etc.

Language Verbosity This criteria evaluates the size of a refactored alternative to cloning as well as the overall boilerplate syntax required to implement a given feature. These map to the cost function described earlier: if a developer prefers cloning to a refactored version that is larger in size, then one can infer that the language is proliferating code cloning by being verbose.

To instantiate these concerns, we now present a case study of code cloning in which we observe problematic situations of lack of abstraction and a surplus of verbosity.

3 Case Study

To motivate the study of programming languages as fueling agents for software clones we now provide a case study of a mature Java project. Containing approximately 7000 Java classes, with a grand total of 775921 LoC. The project was subject to a clone detection experiment, using our tool Kamino, which analyzed several hundred revisions and provided feedback on possible inconsistent cloning situations.

We used a clone detector named Simian[1] to provide information about clone detection. Simian is a text based clone detector that provides fast execution and some level of token insensitivity (i.e. mark two code fragments as equal even though they present different text tokens).

More specifically, Simian found 57672 duplicate lines in 5230 blocks in 2031 files and processed a total of 385622 significant (775921 raw) lines in 7136 files. Dividing the duplicated lines by the total lines we 7.4%. Furthermore, if we divide the number of classes that contain duplicated code by the total number of Java classes, we obtain 28.5%. Hence, our study using Simian found that, 7.4% of all lines codes are duplicated and that 28.5%

Program 1 shows two classes where code duplication is evident. We can observe that there are two major types of similarities: (1) Classes have the same amount of duplicate fragments with the only difference being the type of the object in consideration (`ChangeCanProc` on the left, `TransferCanProc` on the right); (2) if we compare the methods (of either right and left programs) `getValidCanProc()` and `getValidExtCanProc()`, we can observe that the only difference resides in the `if` condition: one has the negation of the other's boolean expression. We can also observe that there is a pattern involving a `for` cycle, an `if` condition and a method being called inside the body of the `if`.

[1] `http://www.harukizaemon.com/simian/`

Listing 1 Case Study: Two fragments of code (left and right) existing in two different Java classes.

```
    if (degree == null) {
       return Collections.emptyList();
    }

    List<ChangeCanProc> result=
new ArrayList<ChangeCanProc>();
    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
    final ChangeCanProc child =
        (ChangeCanProc) process;
        if (child.isCandidacyValid() &&
        child.hasCandidacyForSelectedDegree(
              degree)) {
            result.add(child);
        }
    }
    return result;
}

public Map<Degree, SortedSet<ChangeCanProc>>
     getValidCanProc() {
    Map<Degree, SortedSet<ChangeCanProc>> result =
new TreeMap<Degree, SortedSet<ChangeCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
        final ChangeCanProc child =
        (ChangeCanProc) process;
        if(child.isCandidacyValid() &&
     !child.getCandidacyPrecedentDegreeInformation()
                .isExternal()) {
            addCandidacy(result, child);
         }
     }

    return result;
}

public Map<Degree, SortedSet<ChangeCanProc>>
     getValidExtCanProc() {
    final Map<Degree, SortedSet<ChangeCanProc>> result
= new TreeMap<Degree, SortedSet<ChangeCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final CandidacyProcess process
          : getChildProcessesSet()) {
        final ChangeCanProc child =
              (ChangeCanProc) process;
        if (child.isCandidacyValid() &&
     child.getCandidacyPrecedentDegreeInformation()
            .isExternal()) {
           addCandidacy(result, child);
        }
    }

    return result;
}

private void addCandidacy(final Map<Degree,
          SortedSet<ChangeCanProc>> result,
      final ChangeCanProc process) {

    SortedSet<ChangeCanProc> values =
    result.get(process.getCandidacySelectedDegree());
    if (values == null) {
      result.put(process.getCandidacySelectedDegree(),
               values = new
             TreeSet<ChangeCanProc>(
              ChangeCanProc
              .COMPARATOR_BY_CANDIDACY_PERSON));
    }
    values.add(process);
```

```
    if (degree == null) {
       return Collections.emptyList();
    }

    List<TransferCanProc> result=
new ArrayList<TransferCanProc>();
    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
    final TransferCanProc child =
        (TransferCanProc) process;
        if (child.isCandidacyValid() &&
        child.hasCandidacyForSelectedDegree(
              degree)) {
            result.add(child);
        }
    }
    return result;
}

public Map<Degree, SortedSet<TransferCanProc>>
     getValidCanProc() {
    Map<Degree, SortedSet<TransferCanProc>> result =
new TreeMap<Degree, SortedSet<TransferCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
        final TransferCanProc child =
        (TransferCanProc) process;
        if(child.isCandidacyValid() &&
     !child.getCandidacyPrecedentDegreeInformation()
                .isExternal()) {
            addCandidacy(result, child);
         }
     }

    return result;
}

public Map<Degree, SortedSet<TransferCanProc>>
     getValidExtCanProc() {
  final Map<Degree, SortedSet<TransferCanProc>> result
= new TreeMap<Degree, SortedSet<TransferCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final CandidacyProcess process
          : getChildProcessesSet()) {
        final TransferCanProc child =
              (TransferCanProc) process;
        if (child.isCandidacyValid() &&
     child.getCandidacyPrecedentDegreeInformation()
            .isExternal()) {
           addCandidacy(result, child);
        }
    }

    return result;
}

private void addCandidacy(final Map<Degree,
          SortedSet<TransferCanProc>> result,
      final TransferCanProc process) {

    SortedSet<TransferCanProc> values =
    result.get(process.getCandidacySelectedDegree());
    if (values == null) {
     result.put(process.getCandidacySelectedDegree(),
                values = new
              TreeSet<TransferCanProc>(
               TransferCanProc
                .COMPARATOR_BY_CANDIDACY_PERSON));
    }
    values.add(process);
```

Observing the examples, one might ask: is this set of clones a result from the inexperience of developers or is it influenced by programming language deficiencies? We will enlighten this question in the next section where we will describe limitations inherent to the Java programming language and how Scala tackles them.

4 Abstraction Capability Comparison Between Java and Scala

Java is one of the *de facto* programming languages for object-oriented development. Java combines the syntax of C++ with the semantics of classic object-oriented languages such as Smalltalk. The language had a significant investment in the development of virtual machine and compiler research that allowed it to reach mainstream and widespread audience and effectively penetrate business development.

Scala[11] is a statically typed programming language that runs on JVM and blends pure object orientation with functional programming. It is fully interoperable with Java but provides functional aspects such as function literals (lambdas), closures, functions as first class values, currying and partial function application. Scala provides an union of different features from experimental programming languages (such as Pizza[12] and Java's generics) and industry strength programming languages (such as Java's and C# syntax, Smalltalk's uniform object model, ML's functional approach, Haskell's implicits, Lisp's flexible syntax and Ruby and Python's functional approach with object-oriented design).

The next sections will cover the earlier mentioned topics of language abstraction and verbosity. For each, we will try to observe both language's constructs.

4.1 Abstraction Capabilities

We define abstraction capability as the ability of a developer to lift patterns out of his code and place them in a single location where they can be reused. We will provide several abstraction examples that avoid code duplication.

4.1.1 Method Extraction

As motivation, we will provide a programming scenario inspired in the real example presented in listing 1. In our example, a student (described in listing 2) is characterized by an id, a name, an age and a list of courses he's attending. Bean style selectors and modifiers are left out of the Student class for example sake, but they should be assumed as already defined.

Listing 2 Student class.

```
public class Student {
    private int id;
    private String name;
    private int age;
    private ArrayList<String> courses = new ArrayList<String>();
}
```

Assuming we were given a collection of students one could ask for the students with id lower than a given number. We could write method that implemented this functionality (described in the left side of listing 3) This is a common pattern in Java: given a list of objects, each object that satisfies a given criteria is inserted in a container that is returned in the end. The corresponding code is on the left side of listing 3.

We can observe (in the right side of listing 3) an equal rewrite, using non idiomatic use of the language, of the same program in Scala. Notable differences include method declaration (using `def`), type declaration information (after argument declaration), type inference in variable declaration, `for` comprehension of *foreach* (using `<-`) and non obligation of `return` statement.

Now, if in another point in our program we intended to extract the students that are younger than a given age, we would have two approaches: duplicate the code and change the comparison predicate or; refactor the code to provide a more general method that filters students based on their id, age or any other property. As we saw in section 3, this was one of the causes for code duplication since the developer chose the former, i.e., duplicating the method and executing minor changes. For argument sake we will choose the latter.

Listing 3 Naive filter in Java (left side) and in Scala (right side).

```
public ArrayList<Student> filter(
              List<Student> students,
              int id){
    List<Student> resultingStudents
       = new ArrayList<Student>();
    for(Student student : students)
        if (student.getId() <= id)
            resultingStudents.add(student);
    return resultingStudents;
}
```

```
def filter(students:List[Student],
           id:Int):ListBuffer[Student]={

    var resultingStudents = ListBuffer[Student]()

    for(student <- students)
        if (student.getId() <= id)
            resultingStudents.append(student)
    resultingStudents
}
```

Listing 4 General method extraction approach.

```
public static <T> Collection<T> filter(
    Collection<T> target,
    ComparePredicate<T> predicate) {
            Collection<T> result =
                   new ArrayList<T>();
            for (T element: target) {
                if (predicate.receive(element)) {
                    result.add(element);
                }
            }
            return result;
        }

public interface ComparePredicate<T> {
    boolean receive(T type);
}

// Client code
final String name = "Jack";
Util.filter(students,
            new ComparePredicate<Student>(){
    public boolean receive(Student student){
         return (student
           .getName().compareTo(name) == 0);
    }
});
```

```
// With function passing
def getStudents(students:List[Student],
        studentSelector:Student => Boolean)={
    for (student <- students;
         returningStudent = student
          if studentSelector(student)
        ) yield student
}
// Client Code:
getStudents(students, student:Student =>
                   student.name == "Jack")
```

```
// Alternative
// Get students using library functions:
// Client Code:
students.filter(_.name == "Jack")
```

A general and abstracted approach in Java requires interfaces, type parametrization and anonymous inner classes (as depicted in the left side of listing 4). This approach works for all Student's properties (provided that the `ComparePredicate` is implemented) but also for all types of Classes.

This alternative is complex and programmers, unless writing libraries or frameworks, do not embark in such generalization (proved by the case study). This example of filtering a list can be extended to various other scenarios, such as: finding the max/min of a list, grouping or splitting elements of a list; reducing the list into a value; etc, each of which should be implemented with each corresponding interface and generalized static method.

This constitutes a burden and a price which most developers are not willing to pay, thus ending up in the original duplication scenario.

In terms of code cloning this means that the developer is left with a deficient function passing mechanism, evidenced by the lack of full function passing mechanisms which does not correctly mitigate the practice of code duplication. There are a number of functional libraries (such as guava-libraries[2], functionaljava[3] and lambdaj[4]) to compensate this limitation in Java.

The most general Java approach to filter a collection of objects can be achieved in Scala as shown in the top right side of listing 4). As we can observe, there is no return type in the method declaration (it is inferred) and the `for` expression is capable of iterating, filtering and `yield`ing. The biggest difference, however, is the function type declaration, which describes a function that receives a `Student` and returns a Boolean. Client code needs only to pass a function value. Notice that the variable `id` inside the function value needs not be final (it can be calculated elsewhere), thus allowing closures to be created.

However, it is possible to achieve terser code. We can observe the last code fragment (on the lower right side of listing 4), which depicts a call to a method inside the List collection: `filter`. Scala was developed to ensure that use cases such as filtering a collection was included inside collection objects. For this reason, a developer is able to filter, find an element, find a max/min element, iterate, map, reduce or group over collections, thus enabling code reduction and mitigating duplication.

4.1.2 Interfaces and Traits

Java's interfaces were originally introduced to enhance Java's single inheritance model[13], much like *protocol*s in Objective C.

Interfaces constitute a contract to a client, so that one can call a given set of functionality, but also constitute a form of polymorphism on a object that implements several interfaces (i.e. can be treated as the type that it implements).

Interfaces implement no functionality nor do they include any variables: they only declare constant values and method signatures that need to be implemented on the classes that implement the interface.

This means that interfaces are considered a poor man's reuse mechanism. They do not factor code nor do they generalize implementation. This, in turn, means that abstracting functionality through inheritance in Java always implies lifting that functionality to a super (regular or abstract) class or delegating that functionality to third party objects which will allow future composition.

From a code duplication perspective, a developer that found a code fragment to be a candidate for a refactoring would lift such code fragment to a super class, making it available to all its sub classes. But what if sub classes do not have a common super class? Take, for instance, an example situation: there is a `Cat` and `Dog` class that have a duplicated method `move`. To reduce duplication, a developer refactors method `move` to their common super class: `Animal`. However, if a `Car` class exists which requires the same method `move`, it does not make sense to include it in the class hierarchy that extends from `Animal`. The developer is forced to allow code duplication in `Car` to to avoid an inconsistent class hierarchy. Hence

[2] http://code.google.com/p/guava-libraries/
[3] http://functionaljava.org/
[4] http://code.google.com/p/lambdaj/

the Java inheritance system forces the programmer to choose between a latent trade off: reusability or coherence. The preferred property is generally the latter, which implies a greater number of code duplication in Java projects.

Scala does not have interfaces, at least not by that terminology. Instead, Scala took inspiration from Ruby's mixins and created Traits, which are similar to Java's interfaces with the difference of Traits allowing method implementation. This way, a class can coherently extend a super class and then mixin a number of traits to insert additional functionality. This is a form of multiple inheritance that avoids the some of problems found in C++, namely diamond problem. Hence, observing the previous example, a Trait containing the method `move` could be created, allowing `Cat`, `Dog` and `Car` to mixin the functionality without disturbing class hierarchies. Thus, coherence and reusability can be achieved using Traits which allow for concern separation, modular design and code reuse.

4.2 Language Verbosity

Programming language verbosity can hinder code comprehension and semantics since the programmer needs to observe a number of keywords and sentences to grasp underlying functionality/behavior. From another point of view, however, certain language formalities can also serve a documentation purpose. Best practices and formalizations can serve the role of documentation as they normalize code to a common standard.

We shall analyze Java's formalizations/best practices from the perspective of describing the intent of the programmer and compare it with a Scala alternative.

4.2.1 Getters and Setters

Listing 5 Getters and setters in Java (left side) and Scala (right side).

```
/**
 * Student Class in Java defining
 * an id, a name and the getters/setters
 * for both atributtes
 */
public class Student{
    private int id;
    private String name;

    public int getId(){
        return id;
    }

    public void setId(int id){
        this.id = id;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

```
/**
 * Student class in Scala also defining
 * an id, a name and implicit getters/
 * setters for each attribute
 */
class Student(var id:Int,var name:String)
```

A first example of code that might add unintentional complexity to a given program is Java's *getters* and *setters*. In the Java convention, a class should have its properties declared as `private` and expose a pair of methods that are public getters and setters for those properties. This is so common in Java that Java editors (such as eclipse or netbeans) automatically generate these methods for a given set of properties. They can be harmful since they can hide potential methods that describe functionality. Furthermore, for any

given programming task, these methods could in fact be skipped of code review as best practices state that they do not contain any application functionality or business logic. One can argue that their existence might pollute the code base since the added documentation value is residual.

Scala provides implicit generation of getters and setters for properties of a class. One is still able to create accessors but the language reduces boilerplate by generating what Java defines as best practice.

To create getters, one should mark the attribute with the `val` keyword which creates a method to access the property, e.g. `object.attribute`

To create setters, one should mark the attribute with the `var` keyword. This not only creates the public access method but also creates a public method to set the value with a new value: `example.attribute = newAttribute`.

There is not a direct symmetry to the Java's getter and setter since in Scala the names for both are equal to the attributes they represent, i.e., the getter and setter for a given attribute are the attributes name instead of post-fixing the verb *get* like in Java.

listing 5 depicts the differences of creating a class with two attributes and creating the getters and setters for those same properties. The code reduction achieved in the Scala version loses the Java's documentation purposes but gains more terse and readable code. We left out the constructor of the *Student* in the Java's version since we were concentrated in studying the accessors. However, Scala also includes a constructor with the same argument list as the one provided in the definition of the class.

4.2.2 Anonymous Inner Classes

Listing 6 Sorting using anonymous inner function (left side) and Scala function literals (right side).

```
sort(students,
     new Comparator<Student>(){
  public int compare(Student student1,
                     Student student2){
       // logic of comparing students
    }
});
```

```
// Function literal
(studentA:Student,studentB:Student)
 => /*logic of comparing students*/

// Function literal with type inference
(studentA,studentB) => /*logic*/

// Function literal with placeholders
_ <= _
```

As was seen in the previous section, anonymous inner classes provide an emulation of first class functions, limited by the fact that Java does not currently support full closure behavior. First class functions enable for code reuse by allowing the programmer to inject behavior into another function which can freely choose when and where to evaluate that code. In Java, to enable the creation of functions, one must create an anonymous inner class which has to implement a given contract known by client code. The creation of a anonymous inner class involves the implementation of a method inside a class which, using this paper's terminology, constitutes two code regions.

If there is no contract (i.e. interface or abstract class) for a anonymous inner class to implement, these must be defined. For instance, if we want to define a predicate interface (such as we did in listing 4), we must define a name for that class and for the method that it has to implement. These constitute at least two code regions that can subject to code duplication and also difficult the reading and comprehension of code.

Fortunately there are cases where the Java language already provides such interfaces. *Comparator* is an example of such interface.

As depicted in listing 6 (on the left side), client code is creating a `Comparator` anonymous inner class to pass it to a static sorting function in `Collections`. As we can see, the code that matters is represented by the comment, but it has to be surrounded by a method declaration and class instantiation.

Scala has first class functions, which is a good replacement of anonymous inner classes. In listing 6 (on the right side) there are three types of lambdas.

The first one is a function that receives two arguments of type `Student` and applies a comparison logic. This can be passed as a function value to other functions. However, if we know the types of the arguments to apply to that lambda, one could use type inference to add them, as depicted in the second lambda form. This can be even more concise by removing the intermediate naming by using placeholder syntax ('_'), which is a substitute for variable naming. This is effective upon function calling such as `List(1,2,3,4,5).reduceRight(_ + _)` which sums all the elements of a list and returns 15.

Scala's function literals and values are more concise than Java's anonymous inner classes, which translates into less verbose code.

4.2.3 Constructor Madness

Java allows for the creation of overloaded constructors and methods which provide a way to create optional parameters. The drawback is that this can create a multitude of different combinations (which must be codified in each constructor). Furthermore, upon extension of a given class, which provides overloaded constructors, the sub class must itself define the constructors to enable the propagation of optional parameters. This is called constructor madness [14] (as depicted in the left side of listing 7), since the code is polluted with code duplication of combinations of constructors, which makes inheritance more difficult and reduces comprehension/readability.

Scala also suffered from this problem. However, since version 2.8, Scala introduced named and default parameters[15] which can be applied to class constructors as well as methods. Given a class constructor, one can define optional parameters by defining a default value for that parameter (as evidenced in class `A` of program 7). This allows for omission of parameters that have a default value upon invocation of constructors/methods. However, this may cause ambiguity upon value assignment to a given property, i.e., in a list of parameters that have default values, if one wishes to invoke a method/constructor that assigns a value to the last parameter, one must include values for all previous parameters. Thus, to remove this restriction, named values were created to selectively indicate the name of the property to be initialized. This allows for increased readability, code conciseness and is less error prone upon method/constructor invocation.

Listing 7 Constructor madness in Java and default and named parameters in Scala.

```
class A {
    public A(int x,int y,int z){
        /*...*/
    }
    public A(){
        this(1,2,3)
    }
    public A(int x){
        this(x,2,3)
    }
    public A(int x,int y){
        this(x,y,3)
    }
}
```

```
class A(x:Int=1, y:Int=2, z:Int=3)
```

5 Code Duplication Implications

In the previous section we observed a number of situations where two programming languages provide different abstraction capabilities. In this section we will map each previous situation to a code cloning scenario using our own clone detector: Kamino.

For each abstraction mechanisms, we will analyze the source code regarding clone detection. This way we will obtain data for judging how one language and, more important, how a language's constructs influence and induce code cloning. This section will also provide values to materialize the discussion provided in the previous sections. For the required clone analysis we will use Kamino, which will be described in the following section.

5.1 The Clone Detector: Kamino

Kamino is a tool for clone evolution analyzer. A clone, as described earlier, is a copied and pasted fragment of code that originates naturally in every code base. By introducing time in the process of clone detection, we allow the cloned code fragments to evolve. Evolving cloned fragments are guided by evolutionary patterns[16] which describe how a cloned fragment changed from one point in time to another. Using these patterns, one can create a genealogy of a cloned fragment. One of the possible evolutionary traits, and also the most harmful, is the inconsistent change. This pattern happens when two previously cloned fragments are no longer similar due to some modification in one of them. The modification is inconsistent because it should have been applied to both instead of just one. If the modification is a bug fix, then there is a code fragment in the code base that will have to be fixed. Naturally, if a system has more cloned code then it can be subject to more inconsistent changes.

Kamino also provides a framework for dealing with different clone detectors and using them to extract clone evolution information.

Figure 1 described Kamino's architecture. As we can observe, four main processes exist: (1) the main Kamino process, which mediates the others and incorporates the mechanisms and algorithms described in our approach; (2) the clone detector, which contains the interface for dealing with the primary clone detectors; (3) the meta clone detector, which describes the Kamino unique approach to clone evolution extraction, and; (4) the repository process, which enables a common interface to different repository types. The Database connection is managed solely by the main Kamino process.

Kamino effectively creates clone genealogies for a given system. The main objective of our tool is to build evolutionary lines of cloned fragments with the purpose of providing insights to code but also warning the programmer when clones are modified inconsistently. For this purpose, Kamino includes two main modules: the clone detector module and the clone evolution inference module. For our analysis we will use the clone detector module of our tool.

Given the scope of this paper, Kamino will incorporate its own clone detector. This clone detector parses the whole code into blocks, where a block is defined as a region of code beginning with '*{*' and ending with '*}*'. There may be nested code regions inside code regions and multiple code regions inside a given source code file. After this pre-processing, each code region is hashed into a value, resulting into a flat list data structure of all the hashed code regions existing in the code. Finally, Kamino will group each region by hash equality, where two regions with the same hash are said to be clones of each other. The clone detection approach in the following scenarios will be the same: equal code regions (i.e. code between curly brackets) are considered as clones. Furthermore, Kamino employs a filtering process: if a clone region is a subset of another clone region (i.e. the line range

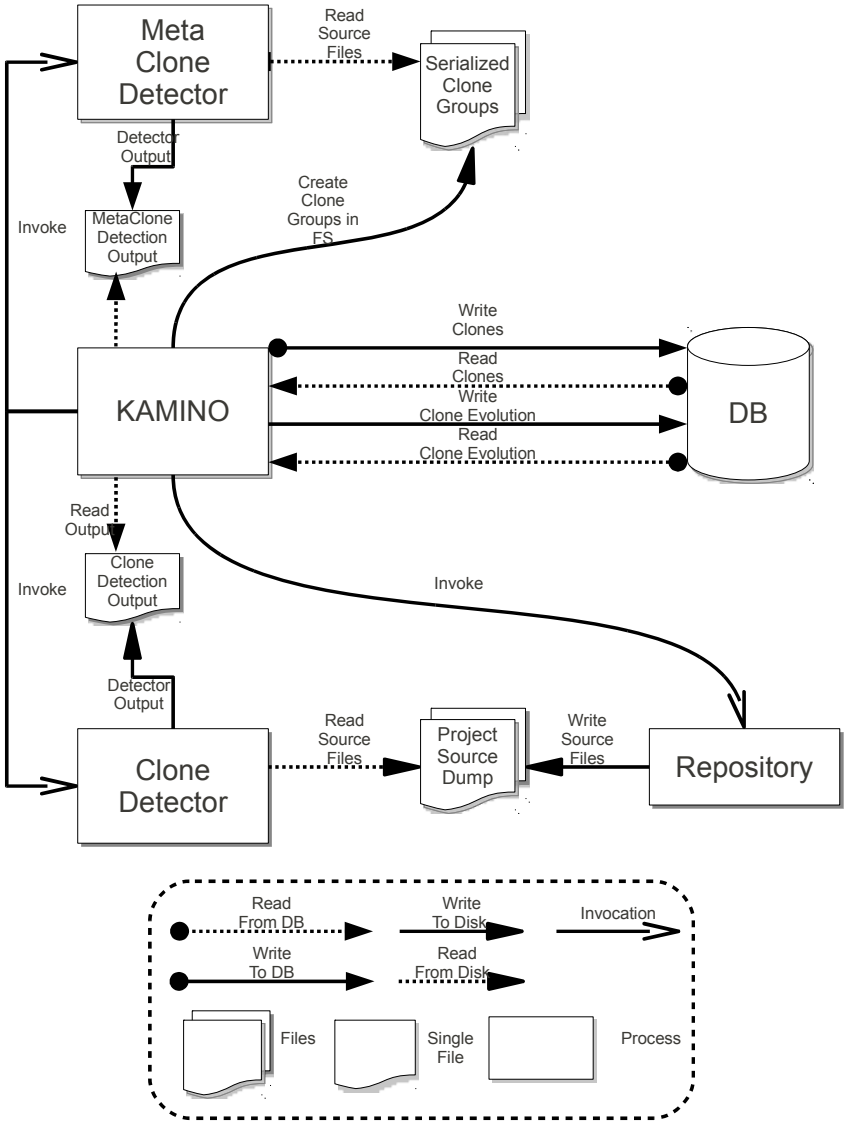


Figure 1 Kamino architecture. Four main processes exist: the main Kamino process, which mediates the others and incorporates the mechanisms and algorithms described in our approach; the clone detector, which contains the interface for dealing with the primary clone detectors; the meta clone detector, which describes the Kamino unique approach to clone evolution extraction, and; the repository process, which enables a common interface to different repository types. The Database connection is managed solely by the main Kamino process.

is within a line range of another clone) then the larger clone is considered and the smaller clone is filtered.

Experiments have showed that there are clones associated with the languages constructs described in the previous section. Throughout this section we will detail these experiments and provide concrete clone data.

5.2 Method Extraction

As we have seen in the final part of section 4.1.1, by having first class functions we can greatly reduce and refactor code upon method extraction. But how does this translate into code cloning?

As we have stated, our code clone is based on the equality between code regions. A general approach would be to detect clones in the worst case scenario: whole file duplication. In a situation where one duplicates all the code in the listing 4, i.e., copy the whole Java program into a file and copy the whole Scala program into another file, the clone detector returns **3** duplicated code regions in the Java program where it returns **1** in the Scala program. Inner block regions are not considered since Kamino filters clones that are subset of other clones, i.e., the larger clone is always selected.

Because Scala has first class functions, there is no need to create an interface nor the client code needs to create an anonymous inner class. This translates into a reduction in the number of code regions. Furthermore, if we only consider the third code fragment in the Scala program, there would be no code clones detected. As we have stated this is a typical programming pattern in Scala and Java which means that there is a high probability of finding code very similar to this in every code base. Thus, in this case, our clone detector finds that there is a **3:1** clone ration in the introduction of first class function in method extraction.

5.3 Interfaces and Traits

As we have seen in section 4.1.2, Java's interfaces help in the design of a system (by defining implementation contracts) but do not guarantee code reuse (because class that implement a given interface must implement the methods defined by that interface, notwithstanding of having the exact same implementation). One could create a scenario where a single interface is defined and two classes implement its methods equally, i.e. have the same implementation for the methods required by that interface. Instantiating for an interface with one method, there would be (at least) **1** clone. In the Scala counterpart, i.e. using traits, there would be no clones since all the code is refactored into one place which can then be mixed in, hence no code duplication. Thus, traits successfully remove code duplication caused by Java's interface.

5.4 Getters and Setters

As referenced in section 4.2.1, Java's best practices require the programmer to provide two methods for each private attribute: a getter and a setter. In terms of code cloning, this would mean two more code regions for each attribute that can be detected as equal in another class. This may constitute two problems.

The first problem is the confusion created to the clone detector. If there are regions of code in which the programmer has no interest in knowing whether they are duplicated or not, then the clone detector should remove them. They can cloud other clones that are

much more meaningful in the eyes of the programmer. This makes the clone detector much more complex for it has to incorporate these filtering rules.

Furthermore, a second problem exists. If a programmer adds more logic to a getter/setter than the best practices advise, then, if we instruct the clone detector to ignore such regions, there may be false negatives (i.e. clones that the clone detector fails to detect) in the results of the tool. Kamino has the premise: *it's better to have false positives (which can further be filtered when showing results to the user) then false negatives.*

If a programmer introduces an attribute with the same name in a different class and provides a getter and a setter with equal implementation (which there is a high probability of since they are guided by best practices), there would be **2** code regions marked as clones.

In the Scala counterpart, the getters and setters are effectively generated, which leaves the source file without the extra two code regions to be marked as clones. Thus, in respect to getters and setters, Java has **2** possible clones where Scala nullifies this problem.

5.5 Anonymous Inner Classes

We already saw how anonymous inner classes versus first class function translate into code cloning in a previous example. However, in the previous scenario, one considered the whole program. Hence the anonymous inner classes clones were removed since they represented subset clones (i.e. clone regions that are inside other greater clone regions).

If we consider the examples showed in section 4.2.2, we could verify that for each anonymous inner class (implementing one method) a function could be written in Scala without any code region whatsoever. One could argue that, because Scala also has anonymous inner classes, the number of code regions should be same. However, in the example we considered, the use of anonymous inner classes is a way of achieving lambdas, i.e. functions defined in place that can be returned or received as arguments. As such, function literals in Scala are considered.

In conclusion: if a programmer duplicates code using the creation of an anonymous inner class (which, because this is often client code, there is a high probability of), there will be at least **1** clone region that can be marked as clone of another code region in another location of a software project. In Scala there is no such code region to be marked as clone, hence there are effectively no code duplication.

5.6 Constructor Madness

Our final example of language limitation regarding verbosity is in section 4.2.3 and is related to the constructor madness problem in Java. When trying to achieve default parameters (i.e. parameters that have a default value if not provided) in the constructor of a given class in Java, one must provide concrete implementations for all of the combinations of default parameters. Observing the code regions inside each overloaded constructor call in the Java listing 7, a programmer could classify all the similar calls to *this* as clones. Since our clone detector is based on equal code regions, it would not detect these code regions as clones.

However, in the worst case scenario (if one duplicates the whole source file, changes the class name and adds an attribute), there would effectively be **4** equal code regions marked as clones. We are being specific to constructor calls but methods have the same problem: the only way to achieve default parameters is through overloading. This causes more code that can be copied and pasted into subsequent locations.

Scala has default parameters which reduce the number of code regions and nullify the constructor madness problem. If we simplify this scenario to one default parameter (requir-

ing one constructor overloading of the constructor), one could obtain **1** clone (if we do not count the primary constructor) whereas in Scala there would be **0** clones.

6 Conclusion

We studied programming languages in the perspective of code cloning, specifically in the Java and Scala programming languages.

We exemplified a real case study where we could observe several code duplication problems which have a high probability of being caused by language verbosity and abstraction mechanisms. We observed abstraction capabilities, namely: anonymous inner classes versus first class functions; interfaces versus traits and; language verbosity (getters/setters, anonymous inner classes verbosity and constructor madness).

We concluded that cloned code is more prone to be created when a language is more verbose (since there is a higher probability of identical code) and also when the effort of refactorization is higher than the simple duplication of source code (in which abstraction mechanisms are of great influence).

We applied our tool, Kamino, to provide clone detection analysis in the comparison of both languages, where we observed that more terse and compact language constructs leave less room for clones. Evidence showed that the more expressive a language is lesser confusion is created in the clone detector and the number of clones drops significantly.

We postulate that a tool like Kamino could be useful in less expressive/more verbose languages where a high volume of cloned fragments can found (and possibly managed) and where language limitations induce the programmer to copy and paste source code to ease the task of software development.

As future work, we would like to study other abstraction mechanisms, such as type classes in haskell, and how software architectures influence the origin of cloned code.

References

1 K. Beck and M. Fowler, "Bad smells in code," *Refactoring: Improving the design of existing code*, 1999.

2 B. Baker, "On finding duplication and near-duplication in large software systems," in *wcre*, p. 86, Published by the IEEE Computer Society, 1995.

3 I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *icsm*, p. 368, Published by the IEEE Computer Society, 1998.

4 J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of the International Conference on Software Maintenance*, vol. 96, pp. 244–253, 1996.

5 T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, pp. 654–670, 2002.

6 K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1, pp. 77–108, 1996.

7 J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pp. 170–178, IEEE, 2007.

8 S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *icsm*, p. 109, Published by the IEEE Computer Society, 1999.

9 Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, pp. 176–192, 2006.

10 C. Roy and J. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, p. 115, 2007.

11 M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," tech. rep., Citeseer, 2004.

12 M. Odersky and P. Wadler, "Pizza into java: Translating theory into practice," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 146–159, ACM, 1997.

13 J. Gosling and H. McGilton, "The java language environment," *Sun Microsystems Computer Company*, 1995.

14 A. Leitao, "The next 700 programming libraries," in *Proceedings of the 2007 International Lisp Conference*, p. 21, ACM, 2007.

15 L. Rytz and M. Odersky, "Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the scala language," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2090–2095, ACM, 2010.

16 M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 187–196, ACM, 2005.

HandSpy – a system to manage experiments on cognitive processes in writing

Carlos Monteiro[1] and José Paulo Leal[2]

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

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

Abstract

Experiments on cognitive processes require a detailed analysis of the contribution of many participants. In the case of cognitive processes in writing these experiments require special software tools to collect gestures performed with a pen or a stylus, and recorded with special hardware. These tools produce different kinds of data files in binary and proprietary formats that need to be managed on a workstation file system for further processing with generic tools, such as spreadsheets and statistical analysis software. The lack of common formats and open repositories hinders the possibility of distributing the work load among researchers within the research group, of re-processing the collected data with software developed by other research groups, and of sharing results with the rest of the cognitive process research community.

This paper presents HandSpy, a collaborative environment for managing experiments in cognitive processes in writing. This environment was designed to cover all the stages of the experiment, from the definition of tasks to be performed by participants, to the synthesis of results. Collaboration in HandSpy is enabled by a rich web interface developed with the Google Web Toolkit. To decouple the environment from existing hardware devices for collecting written production, namely digitizing tablets and smart pens, HandSpy is based on the InkML standard, an XML data format for representing digital ink. This design choice shaped many of the features in HandSpy, such as the use of an XML database for managing application data and the use of XML transformations. XML transformations convert between persistent data representations used for storage and transient data representations required by the widgets on the user interface. This paper presents also an ongoing use case of HandSpy where this environment is being used to manage an experiment involving hundreds of primary schools participants that performed different tasks.

1998 ACM Subject Classification J.4 Social and Behavioral Sciences: Psychology

Keywords and phrases InkML, collaborative environment, XML data processing

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.123

1 Introduction

Writing is a basic tool for a successful personal and academic growth. Given the importance of this subject social scientist are actively researching the cognitive processes involved in writing. As the goal of this research is in general to determine the factors that influence a development of writing skills, the participants are normally school children. The object of these research studies are writing productions on different tasks such as narratives, copies, dictations and alphabet transcriptions.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 123–132

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

The tools used for researching cognitive processes in writing are usually composed of two components: an hardware component capable of recording writing gestures and a piece of software to analyze the data. These tools use proprietary data formats for the collected data that hinders the possibility of sharing collected data to be analyzed in other systems or for other purposes. Moreover, the data collected with these type of hardware typically generates several files for each participant, on each task. Due to the large quantity of participants the amount of generated data files is considerable, hence managing data files is a major problem in this type of experiments.

This paper describes HandSpy, a new web based system offering typical features of cognitive processes research tools, but innovating on the experiment management. By providing a repository , researchers can set up an experiment for storing all the entities involved, such as tasks definitions, trait information on participants involved in the experiment and the generated data. The system follows a paradigm of collaborative experiments where various researchers can work on the same experiment simultaneously. HandSpy uses a standard XML[5] format for data files which enables users to collect data from various hardware devices as long as the generated files are kept in this format.

The analysis process differs also from the other existing systems. Instead of inputting the analysis results of each writing production on an external statistical analysis suite to generate results, HandSpy uses a selection engine to define a pool of productions bounded by a set of characteristics, the selected productions are subjected to an analysis process and the merged results of the selection can be exported.

The paper is organized as follows. Section 2 describes technologies used to develop HandSpy and outlines the main characteristics of two tools designed to study handwriting processes. Section 3 is a main description of HandSpy design and implementation methods. Section 4 is a description of a case study conducted with HandSpy and describes a data collection platform used to record data. Section 5 concludes this paper and sets the next steps for HandSpy improvement.

2 Related Work

This section covers background topics related to the development of a collaborative environment for managing experiments on cognitive processes in writing. The proposed environment is based on the use of the InkML [6] data format thus the first subsection is devoted to this standard. To the best of the authors knowledge, no collaborative environment with goals similar to those of HandSpy was ever described in the literature. However, there are some tools for collecting and processing gesture data for experiments in writing. The second subsection describes two tools to which HandSpy owes credit. The final subsection provides an overview on two types of components used in the development of HandSpy, namely the XML database and web interface toolkit.

2.1 InkML

The recent trend of sketching or writing on digital devices capable of recording hand gestures created the need for a standard to describe this kind of data. InkML is a W3C recommendation for storing and exchanging what is commonly called *digital ink*. It is an XML data format to describe a set of strokes digitally representing handwriting and other ink input gestures.

The ink in InkML is defined by characteristics associated with the act of creating a trace such as the width and color of the trace, the pen orientation while writing, the pen distance

Listing 1 InkML "hello" example.

```
<ink xmlns="http://www.w3.org/2003/InkML">
 <trace>
    10   0,  9  14,  8  28,  7  42,  6  56,  6  70,  8  84,  8  98, 8 112,
     9 126, 10 140, 13 154, 14 168, 17 182, 18 188, 23 174, 30 160, 38 147,
    49 135, 58 124, 72 121, 77 135, 80 149, 82 163, 84 177, 87 191, 93 205
 </trace>
 <trace>
   130 155, 144 159, 158 160, 170 154, 179 143, 179 129, 166 125, 152 128,
   140 136, 131 149, 126 163, 124 177, 128 190, 137 200, 150 208, 163 210,
   178 208, 192 201, 205 192, 214 180
  </trace>
  <trace>
   227  50, 226 64, 225 78, 227 92, 228 106, 228 120, 229 134, 230 148,
   234 162, 235 176, 238 190, 241 204
  </trace>
  <trace>
   282  45, 281 59, 284 73, 285 87, 287 101, 288 115, 290 129, 291 143,
   294 157, 294 171, 294 185, 296 199
  </trace>
   <trace>
   366 130, 359 143, 354 157, 349 171, 352 185,  359 197, 371 204,
   385 205, 398 202, 408 191, 413 177, 413 163, 405 150, 392 143, 378 141
   </trace>
</ink>
```

to the surface (whether the trace was made with the pen down or hovering the writing surface), among others.

A set of traces can be grouped in a context defining optional features such as starting time, writing surface dimensions and characteristics of the trace. The Listing 1 is an example of the "Hello" word described in a basic InkML file. The word has five letters represented with five trace elements, the values are separated by commas and represent the coordinates (X,Y) of every point to draw each letter.

2.2 Hand Gesture Collecting Tools

The two main tools currently available to research on cognitive processes in writing are Eye And Pen[3] and Ductus[8]. The use of these tools to conduct experiments involves two main steps: data recording and data processing. Despite some differences on the data recording step, both software solutions provide an interface to predefine settings to direct calculations. The information generated by these software tools is focused on the complementary concepts of burst and pause. A *burst* is a time span in which text was produced without interruptions. A *pause* is non-writing time span between two writing bursts.

Both software tools generate information about the pauses such as position in text and duration time. Document overall kinematic information on the written data such as duration of the experiment, velocity and writing distance is also displayed. The calculations can be exported in a spread sheet format.

The two systems differ in the way they collect gesture data. The Ductus software uses a digitizer to record the pen position and pressure. The Eye and Pen software uses a digitizing tablet and an eye tracker to synchronously record the pen position and the eye point of regard on the tablet, correlating the eye and pen movement during bursts and pauses. The user interface provides a player that displays the written text and eye gaze point recorded during the text production time. The user interface displays also bounding boxes surrounding each possible word. These bounding boxes are computed from the distance between traces using

a threshold that can be customized for each case. Each identified word can be associated with an ASCII string and extra information about it can be added.

2.3 XML Databases

With the popularity of XML use as a transactional data format, there was a natural increase in the use of XML databases. There are two types of XML databases, the XML enabled and the native XML databases.

The XML enabled databases follow a structure of a relational database. The structure of the XML file can be used to transform it into a common relational table set. Systems like Oracle object-relational DBMS can store XML files in XMLType which is a specific type to store XML files with built-in function that allow the management of the content in the files.

On other hand native XML databases can store the actual XML files without following any schema. The XML structured files enable the database to perform indexing which is the key for efficient queries in these systems. Most native XML databases have a set of built in functionalities that enable XML files management and database access.

The following paragraph has a description of some technologies embedded in most native XML databases. XQuery[4] is a language used to query XML databases, the advantage of using XQuery is the possibility of creating result sets based on the contents of the XML file. The XQuery Java API (XQJ) permits to create and submit XQuery expressions to the database and use the results set as Java objects. The XML Database[1] (XML:DB) API allow the management of XML databases disregarding the database system, this permits to create an environment capable of working and manage several databases.

There are several XML database systems distributed under commercial and free software license. Under commercial license the most complete systems are Oracle XML DB, Documentum xDB and MarkLogic Server. All these systems offer basic tools for querying using XPath, XQuery and support for XSL transformations. On free systems the most complete are BaseX, eXist[2] and Sedna. These systems use the same basic tools as commercial products for querying but the most complete system is eXist offering support for XML:DB API and XQuery Java API.

2.4 Javascript Frameworks

Developing web pages using JavaScript frameworks became a common practice due to the large set of pre-written widgets and functions that are offered. These frameworks enable developers to focus on the creation of new cross browser interactive contents. There are several JavaScript frameworks such as Dojo, jQuery, SmartClient and Prototype, these frameworks apart from design differences offer almost the same obligatory features which are described in the following paragraph.

To simplify the access to methods of Documents Object Model (DOM) objects they are wrapped into new objects with simple access methods. Offering a large set of customizable widgets tied to data sources, simplifies data bindings with the server.

3 HandSpy

HandSpy is a web based application to manage distributed and collaborative experiments on cognitive processes in writing. This system has the following distinctive features:

- an experiment management philosophy encompassing all the steps of the research in cognitive processes in writing;

- a multiuser web interface fostering collaboration among researchers and enabling remote work on the experiments;
- a cloud-based data management system providing central storage for all data collected in the experiments;
- an analysis process of the collected data, inspired in the state-of-art systems described in Section 2;
- the ability to select and synthesize collections of data based on different criteria;
- the use of standard XML based data formats to promote interoperability and cooperation among researchers in this community.

HandSpy follows a 3-tier architectural model as depicted in Figure 1 where the presentation tier (a web interface) is represented by the left box, the logic tier (a web server) is represented by the central box and the data tier (an XML database) is represented by the right box. This diagram represents also in three rows the data flows between these tiers.

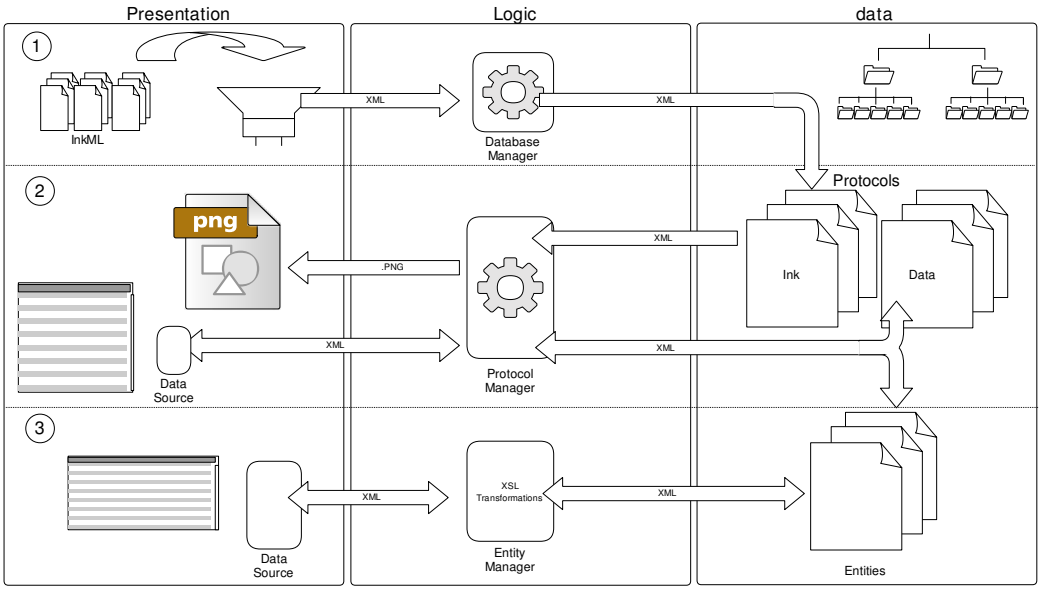


Figure 1 HandSpy application architecture.

In the top row marked with number one is represented the process of uploading data files in InkML format to the database through the web interface. On the server side the database manager module is responsible to organize the uploaded files to the respective collections based on the current user context.

The middle row represents the interaction with ink data. The two main components of HandSpy user interface are depicted in this row, an image viewer to display the written production of the protocol ink and a list grid to display calculations based on the protocol data. The server gets the ink of the selected protocol and generates an image file to feed the image viewer. The list grid is populated with a data source document containing the calculations results generated following the predefined settings on the interface. To optimize the system the main definitions on the HandSpy are classified and treated as entities. This generalization of data permits to manage it in the same way. All data showing objects are based on list grids which use XML data sources. In the third row of the model in

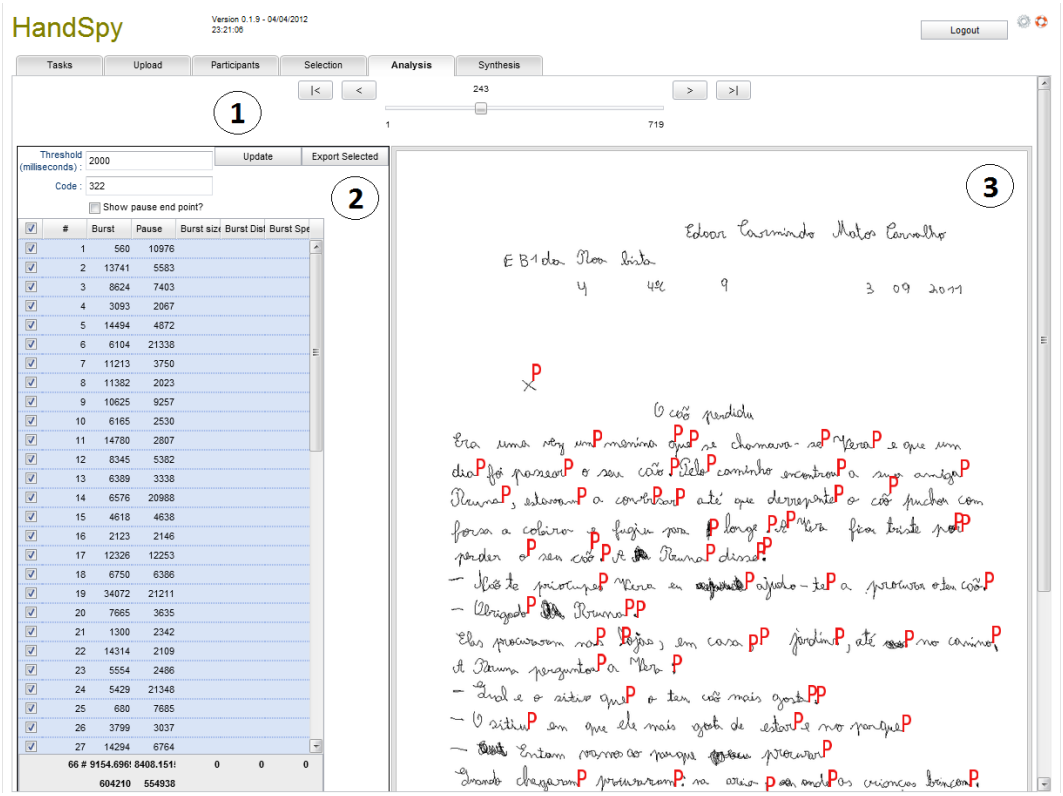


Figure 2 HandSpy interface: section 1 - slider to navigate through selection; section 2 - list with pauses duration, bursts duration, burst size (number of words in the burst), burst distance and burst average speed; section 3 - widget to display the protocol image with the selected pauses identified with a red P.

the Figure 1 is shown the Entity Manager that identifies the entity and uses the respective XSL transformer to transform the data stored in the data base into the client specific data source when the fetch operation is made. The operations of adding, updating and deleting entities entries use the correspondent XSL transformations to perform the operations and save the changes to the data base.

3.1 Design

3.1.1 Application Interface

The graphical user interface of HandSpy relies on a web application. The workspace is divided in six tabs covering the usual flow of an experiment in on cognitive processes in writing.

Tasks Identification of tasks to be performed by the participants during the experiment. For instance, an experiment may include a task where participants must write as much letters of the alphabet as they can in a fixed amount of time.

Upload Upload of the InkML files collected with specialized hardware (smart pens or digitizing tablets) to the system. At this stage the InkML data is connected with a task and a participant. The interface displays a collections of thumbnail images of the

uploaded files and allows the selection of particular file that is displays in its real size for better identification.

Participants Manage the participants in the current experiment. Display the features and the tasks completed by each participant. Custom features describing the participants, such as handiness or mother language, can be added to the participants. The participants features are useful for selecting them to a particular study. The list of participants can be imported and exported as a CSV (Comma Separated Values) file.

Selection Selection of protocols based on tasks and on features of the participants such as age, handedness and gender. The selection is a collection of conditions on protocols to be analyzed and synthesized. Selections set by different researchers are independent from each other, enabling researchers to analyze different collections of protocols simultaneously.

Analysis Figure 2 is a screenshot of HandSpy interface with the Analysis tab selected. The area identified as 1 is a slider to browse the current of protocol selection (set in the Selection tab). Area 2 has a form to define the parameters to calculate the pauses which are listed in the table below. Each row has a pause duration, a burst duration, a burst size is a number of words present on the burst, burst distance and the burst average speed. The footer of the table presents statistics on some of its columns, such as the average and standard deviation of durations, and the count of words. Area 3 displays the written production with red Ps (for Pause) marking the place where the pauses selected in area 2 start. Pause selection allows worthless parts (for instance, a part where the participant erased a word) to be removed from the analysis.

Synthesis Displays global statistics on the data of processed on the Analysis tab and bounded by the selection criteria. The statistics presented in Analysis tab table footer for each protocol are computed on this tab aggregating all the selected protocols. These results can be exported to other systems, such as spreadsheets or statistical analysis packages.

3.1.2 Data Repository

HandSpy processes data uploaded in XML files and stores it in a native XML database. The database structure model is represented in the Figure 3. Every experiment has a set of entities that store data about Tasks, Participants, Selection and Configurations. The data files containing the text production of one experiment are stored in the collection Ink in InkML format and remain unchanged. For every task added to the project a task collection is created to store data files containing calculations and other related data.

3.2 Implementation

As depicted in Figure 1, HandSpy is a composed of a presentation, logic and data layers. The presentation layer was implemented on a SmartClient JavaScript framework. The logic layer was implemented on the Tomcat servlet container and the data layer on the eXist XML database. The remainder of this subsection presents implementation regarding the use of these components.

The Isomorphic SmartClient LGPL platform[1] provided the web toolkit for the user interface. SmartClient provides sophisticated table editing widgets connected to data sources in XML formats that are appropriated to the data handled in HandSpy. These widgets have many built-in functions, such as sorting and grouping on every column, search fields

[1] `http://www.smartclient.com/`

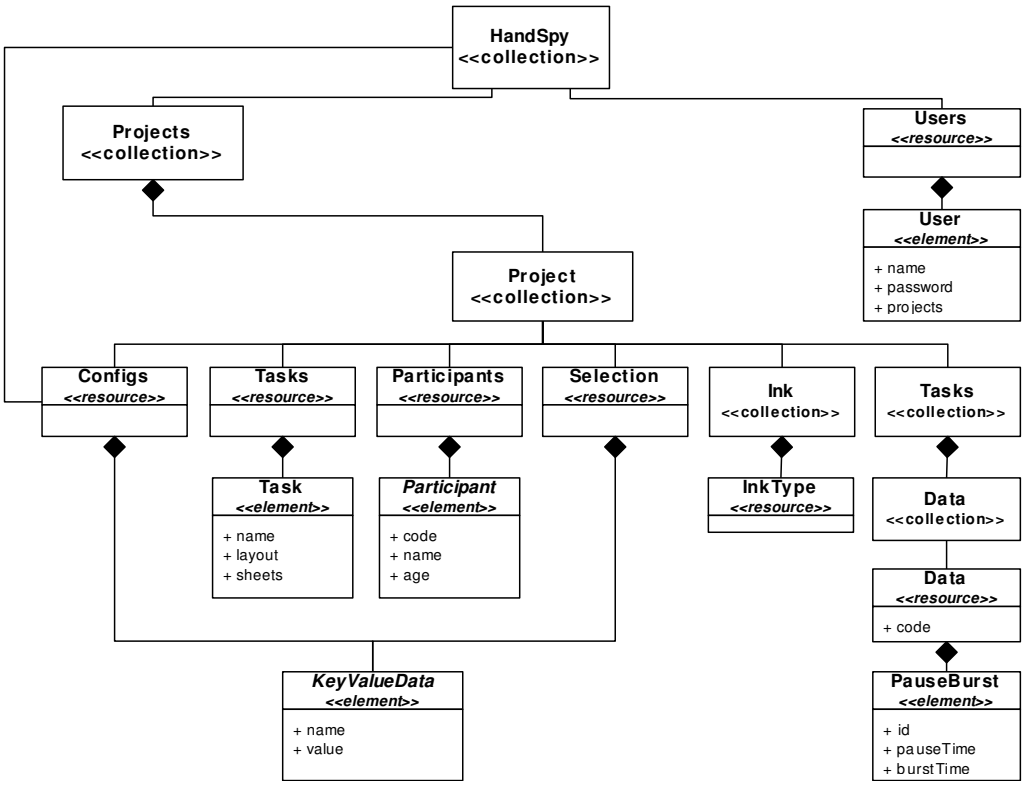


Figure 3 Database structure diagram.

and column customization. Data operations, such as fetching or querying, are built-in functions the data source object. Protocol images on different sizes are generated in the server is displayed in HTML canvas object and are embedded in a SmartClient HTML pane component. This enables the use of a native HTML control on the canvas for managing overlay image objects on the original image.

The server was developed on Tomcat[2] - a Java Servlet container to interact with the client interface and the data storage. For protocol image generation the InkML files are converted into Java objects to produce image files. For feeding SmartClient data sources the XML files in the database are converted into the XML format required by SmartClient data sources. The server runs a set of modules to manage the system some of the most important are listed below.

- `Process` is a Java Servlet, the only outside entry point in the system. Process the requests from the client and generate the responses.
- `DBconnection` is a singleton module responsible for making the connection with the data base using XML:DB API. The database system is implemented using eXist.
- `Selector` creates a XQuery expression from the user selection criteria to create a collection of protocols to be analyzed in the system.

[2] `https://tomcat.apache.org/`

- `EntitiesManager` uses XSL transformations to manage data transferences between the data base and the application client.

The database was implemented using eXist[3] database management system. The communication between the server and the database is made using XML:DB API. The implementation leverages on XML to create RESTful services that provide data sources to the web interface. The XML format required by data sources of SmartClient widgets is obtained by converting files stored in eXist using eXtensible Stylesheet Language Transformations (XSLT)[7]. Other uses of data stored in the database rely on the Java Architecture for XML Binding (JAXB)[9]. This API serializes Java Objects into XML files and vice versa, this enables total control for creating and editing XML files.

4 Case Study

HandSpy is being used to manage an experiment on cognitive processes in writing. This study is targeted to students in grades 2 to 7, totaling 560 children carrying out five different writing tasks. The data collection was made in the classroom with fifteen students at a time.

The device used to collect the data for this experiment was the Livescribe[4] smart pen. This smart pen is similar to a normal pen and has an infra-red camera capable of determining and recording its position over time on a special paper. This paper has a "patterns" of micro-dots that the camera is able to detect and use to determine its exact position.

There are several advantages of using smart pen to replace the digitizing tablets traditionally used for this kind of experiment. Firstly, the simplicity of setting up an experiment in a classroom, a place already familiar to the participants. Secondly, the fact of being a writing device similar to the pens normally used by your children in school. These features make the smart pen less intrusive that digitizing tablets. The cost of running the experiment with smart pens is also a relevant. The price of single digitizing tablet is equivalent to several pens, they are easy to carry, and a single researcher can supervise several participants at once.

The smart pen run a piece of software — a *penlet* – specially developed for this type of experiment. The penlet is an component developed using the Livescribe Java API which is based on the Java Micro Edition. The HandSpy penlet life cycle starts when the tip of the pen touches the HandSpy paper application and `initApp()` method is invoked. All the strokes events raised by the pen on this paper application are processed and additional information about the time of each point in the stroke is serialized to the penlet internal memory pool. The penlet life cycle ends when the pen is shut-down and destroy() method is invoked.

The sheet of paper with a micro-dots used by the pen to obtain its current position is called a *paper application*. The paper application is an Anoto Functionality Document (AFD) which has digital information about the physical paper such as active regions to raise events on the penlet. For this experiment were used three different page layouts to define each task using the same AFD. The main functionality of this paper application is added by an active region to define the start and end time of the narrative task.

Data stored in the smart pen is retrieved the HandSpy DataCapture application. This application was developed a C# using the Livescribe Desktop SDK. It is designed to browse

[3] `http://exist-db.org/`
[4] http://www.livescribe.com/

the Smart Pen AFD collection and use the additional data files, stored with the use of the HandSpy Penlet, in the memory of the Smart Pen to create an InkML file. DataCapture starts when the pen is attached to the USB powered connection dock. At this stage the collected data has already been uploaded to HandSpy and the association with the participants who produced the written data is done. The data is currently being analyzed.

5 Conclusions and Future Work

With the use of new devices capable of recording hand gestures, the use of digital handwriting as a transferable data is becoming more common. These new possibilities rise new ways of studying the cognitive processes involved in the handwriting process. In this paper is described the design and implementation of a new tool to support the study on cognitive processes in writing, HandSpy. This tool aims to extend the experience of conducting a study where large amounts of data need to be manage and where collaborative work speed up the analysis process of this data. Using a web platform, HandSpy is a powerful tool to be used as a cross platform environment and brings the cognitive study experience to a cloud environment.

HandSpy is currently being used to manage an experiment under a research project of the Faculty of Psychology and Educational Sciences of the University of Porto, successfully storing over a thousand written productions and related data. As future work we aim to improve the user experience by implementing missing features such as system management configurations, expand selection parameters, synthesis engine, written productions replay controls and other features that may arise with the current usage on the study case.

Acknowledgments This work is in part funded by the ERDF/COMPETE Programme and by FCT within project FCOMP-01-0124-FEDER-022701. The authors wish also to thank the reviewers for their helpful comments.

References

1 Application Programming Interface for XML Databases (XML:DB API). http://xmldb-org.sourceforge.net/.
2 eXist-db Open Source Native XML Database, 2011. http://exist-db.org.
3 D. Alamargot, D. Chesnet, C. Dansac, and C. Ros. Eye and pen: A new device for studying reading during writing. *Behavior Research Methods*, 2006.
4 S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language, 2010. http://www.w3.org/TR/xquery/.
5 T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language(XML)., 2008. http://www.w3.org/TR/xml/.
6 Y. Chee, K. Franke, M. Froumentin, S. Madhvanath, J. Magana, G. Pakosz, G. Russell, M. Selvaraj, G. Seni, C. Tremblay, and L. Yaeger. Ink Markup Language (InkML), 2011. http://www.w3.org/TR/InkML/.
7 J Clark. XSL Transformations (XSLT), 1999. http://www.w3.org/TR/xslt.
8 E.Guinet and S. Kandel. Ductus: A software package for the study of handwriting production. *Behavior Research Methods*, 2010.
9 E. Ort and B. Mehta. Java Architecture for XML Binding (JAXB), 2003. http://www.oracle.com/technetwork/articles/javase/index-140168.html.

Computing Semantic Relatedness using DBPedia

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

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
c0416098@alunos.dcc.fc.up.pt

3 **CRACS & INESC-Porto LA & DI-ESEIG/IPP**
Porto, Portugal
ricardo.queiros@eu.ipp.pt

Abstract

Extracting the semantic relatedness of terms is an important topic in several areas, including data mining, information retrieval and web recommendation. This paper presents an approach for computing the semantic relatedness of terms using the knowledge base of DBpedia — a community effort to extract structured information from Wikipedia. Several approaches to extract semantic relatedness from Wikipedia using bag-of-words vector models are already available in the literature. The research presented in this paper explores a novel approach using paths on an ontological graph extracted from DBpedia. It is based on an algorithm for finding and weighting a collection of paths connecting concept nodes. This algorithm was implemented on a tool called Shakti that extract relevant ontological data for a given domain from DBpedia using its SPARQL endpoint. To validate the proposed approach Shakti was used to recommend web pages on a Portuguese social site related to alternative music and the results of that experiment are reported in this paper.

1998 ACM Subject Classification H.3.3 Information Search and Retrieval

Keywords and phrases semantic similarity, processing wikipedia data, ontology generation, web recommendation

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.133

1 Introduction

Searching effectively on a comprehensive information source as the Web or just on the Wikipedia usually boils down to using the right search terms. Most search engines retrieve documents where the searched terms occur exactly. Although stemming search terms to obtain similar or related terms (e.g. synonyms) is a well known technique for a long time [15], it usually considered irrelevant in general and search engines of reference no longer use it [1].

Nevertheless, there are cases where semantic search, a search where the meaning of terms is taken in consideration, is in fact useful. For instance, to compare the similarity of genes and proteins in bio-informatics, to compare geographic features in geographical informatics, and to relate multi-word terms in computational linguistics.

The motivation for this research in semantic relatedness comes for another application area, recommendation. Most recommenders use statistical approaches, such as collaborative filtering, to make suggestions based on the choices of users with a similar choice pattern. For instance, an on-line library may recommend a book selected by other users that also bought

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 133–147

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

the books already in the shopping basket. This approach has a cold start issue: what should be recommended to someone that was not yet bought anything? to whom recommend a book that was just published and few people have bought?

An alternative approach is to base recommenders on an ontology of recommend items. An on-line library can take advantage from the structure of an existing book classification, such as the Library of Congress Classification system. However, in many cases such classification does not exist and the cost of creating and maintaining an ontology would be unbearable. This is specially the case if one intends to create an ontology on a unstructured collection of information, such as a folksonomy.

Consider a content-based web recommendation system for a social network, where multimedia content (e.g. photos, videos, songs) is classified by user-provided tags. One could simply recommend content with common tags but this approach would provide only a few recommendations since few content item share the exact same tags. In this case, to increment the number of results, one could search for related tags. For instance, consider that your content is related to music that users tag with names of artists and bands, instruments, music genres, and so forth. To compute the semantic relatedness among tags in such a site one needs a specific ontology adapted to this type of content.

It should be noticed that, although several ontologies on music already exist, in particular the Music Ontology Specification[1], they are not adjusted to this particular use. They have a comprehensive coverage of very broad music genres but lack most of the sub-genres pertinent to an alternative music site. To create and maintain an ontology adjusted to a very specific kind the best approach is to extract it from an existing source. The DBpedia[2] is a knowledge base that harvests the content of the Wikipedia and thus covers almost all imaginable sources. It is based on an ontology that classifies its subjects and on mapping rules that convert the content of Wikipedia info-boxes and tables into Resource Description Framework (RDF) triplets available from a SPARQL endpoint (SPARQL is a recursive acronym for SPARQL Protocol and RDF Query Language).

In this paper we present Shakti, a tool to extract an ontology for a given domain from DBPedia and use it to compute the semantic relatedness of terms defined as labels of concepts in that ontology. One of the main contribution of this paper is the algorithm used for computing relatedness. Most algorithms for computing semantic relatedness based on ontologies assume that these are taxonomies or at least direct acyclic graphs which is not the case with an ontology extracted from DBpedia. Also, these algorithms usually focus on a notion of distance. Instead the proposed algorithm is based on a notion of proximity. Proximity measures how connected two terms are, rather than how distant they are. A term may be at the same distance to other two terms but have more connections to one than the other. Terms with more connections are in a sense *closer* and thus have an higher proximity.

The rest of this paper is organized as follow. The following section presents related work on semantic relatedness algorithms and on the use of knowledge bases such as DBpedia. Section 3 is the main section as it presents the proposed algorithm and Shakti, a tool implementing it. The following section presents a use of Shakti to populate a proximity table of a recommender service that was used as validation of the proposed approach. The final section summarizes the contributions of this paper and highlights future directions of this research.

[1] `http://musicontology.com/`
[2] `http://dbpedia.org/About`

2 Related Work

This section summarizes the concepts and technologies that are typically used as basis for the computation of semantic relatedness of terms in the Web.

2.1 Knowledge Representation

Currently, the Web is a set of unstructured documents designed to be read by people, not machines. The semantic web — sponsored by W3C - aims to enrich the existing Web with a layer of machine-interpretable metadata on Web resources so that computer programs can predictably exchange and infer new information. This metadata is usually represented by a general purpose language called Resource Description Framework (RDF). Its specification [2] includes a data model and a XML binding. The data model of RDF is a collection of triples — subject, predicate and object — that can be viewed as a labeled directed multi-graph; a model well suited for knowledge representation. Ontologies formally represent knowledge as a set of concepts within a domain, and the relationships between those concepts. Ontology languages built on top of RDF provide a formal way to encode knowledge about specific domains, including reasoning rules to process that knowledge [4]. In particular, RDF Schema [3] provides a simple ontology language for RDF metadata that can be complemented with the more expressive constructs of OWL [12]. The triplestores can be queried and updated using the SPARQL Protocol and RDF Query Language (SPARQL).

2.2 Knowledge Bases

Knowledge bases are essentially information repositories that can be categorized as machine or human-readable information repositories. A human-readable knowledge base can be coupled with a machine-readable one, through replication or some real-time and automatic interface. In that case, client programs may use reasoning on computer-readable portion of data to provide, for instance, better search on human-readable texts. A great example is the machine-readable DBpedia extraction from human-readable Wikipedia.

Wikipedia articles consist mostly of free text. However, the joint efforts of human volunteers have recently obtained numerous facts from Wikipedia, storing them as machine-harvestable triplestores in Wikipedia infoboxes [17]. The DBpedia project extracts this structured information and combines this information into a huge, cross-domain knowledge base. DBpedia uses RDF as the data model for representing extracted information and for publishing it on the Web. Then, SPARQL can be used as the query language to extract information allowing users to query relationships and properties associated with many different Wikipedia resources.

2.3 Semantic Similarity

Extracting the semantic relatedness of terms is an important topic in several areas, including data mining, information retrieval and web recommendation. Typically there are two ways to compute semantic relatedness on data:

1. by defining a topological similarity using ontologies to define the distance between words (e.g. in a directed acyclic graph the minimal distance between two term nodes);
2. by using statistical means such as a vector space model to correlate words from a text corpus (co-occurrence).

Semantic similarity measures have been developed and applied in several domain ontologies such as in Computational Linguistics (e.g. Wordnet[3]) or Biomedical Informatics (e.g. Gene Ontology[4]). In order to calculate the topological similarity one can rely either on ontological concepts (edge-based or node-based) or ontological instances (pairwise or groupwise). A well-known node-based metric is the one developed by Resnik [13] which computes the probability of finding the concept (term or word) in a given corpus. It relies on the lowest common subsumer which has the shortest distance from the two concepts compared. This metric is usually applied on WordNet [6] a lexical database that encodes relations between words such as synonymy and hypernymy. A survey [14] between human and machine similarity judgments on a Wordnet taxonomy reveal highest correlation values on other topological metrics such the ones developed by Jiang [9] and Lin [10].

Statistical computation of semantic relatedness relies on algebraic models for representing text documents (and any objects, in general) as vectors of identifiers. Comparing text fragments as bags of words in vector space [1] is the simplest technique, but is restricted to learning from individual word occurrences. The semantic sensitivity is another issue where documents with similar context but different term vocabulary won't be associated, resulting in a "false negative match". Latent Semantic Analysis (LSA) [5] is a statistical technique, which leverage word co-occurrence information from a large unlabelled corpus of text [8].

Currently, Wikipedia has been used for information retrieval related tasks [16, 18, 7, 11]. This is due to the increasing amount of articles available and the associate semantic information (e.g. article and category links). One of these efforts is the Explicit Semantic Analysis(ESA), a novel method that represents the meaning of texts in a high-dimensional space of concepts derived from Wikipedia and the Open Directory Project (ODP). It uses machine learning techniques to represent the meaning of any text as a weighted vector of Wikipedia-based concepts. The relatedness of texts in this space is obtained by comparing the corresponding vectors using conventional metrics (e.g. cosine) [7].

3 Shakti

This section presents an approach to compute the semantic relatedness between terms using ontological information extracted from DBpedia for a given domain. The first subsection outlines the algorithm used for computing semantic relatedness as a measure of proximity between nodes in a graph. The second subsection presents the design of a system to extract the relevant part of DBPedia for a given domain, connect concept nodes on the ontological graph and compute their proximity. The last subsection describes the actual implementation of Shakti.

3.1 Algorithm

Concepts on DBPedia are represented by nodes. Take for instance the music domain used for the case study presented in section in section 4. Singers, bands, music genres, instruments or virtually any concept related to music is represented as a node in DBpedia. These nodes are connected by properties, such as `has genre` connecting singers to genres, and thus form a graph. This graph can retrieved in RDF format using the SPARQL endpoint of DBpedia.

[3] http://wordnet.princeton.edu/
[4] http://www.geneontology.org/

The core idea in the research presented in this paper is to use the RDF graph to compute the similarity between nodes. Actually, we are interest on the similarity between terms, but each node and arc of this graph has 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.

On a closer inspection this interpretation of relatedness as the inverse of distance reveals some problems. Consider the graph in Fig. 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 Lady Gaga or Madonna simply because they are both bands, which also should not be the case.

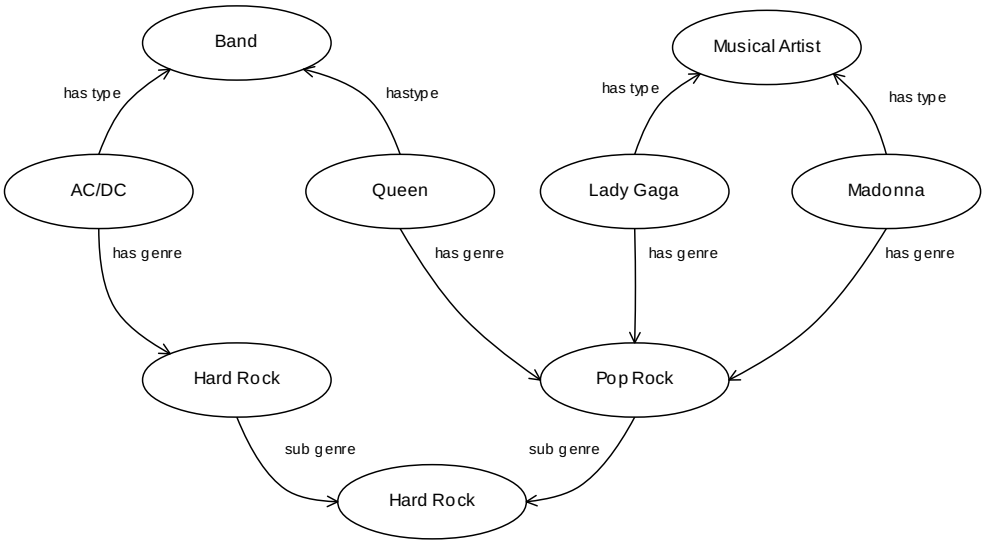


Figure 1 RDF graph for concepts in music domain.

In the proposed approach we consider *proximity* rather than distance as a measure of relatedness among nodes. By definition[5], 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 a metaphor consider the proximity between two persons. More than resulting from a single common interest, however strong, it results from a collection of common interests.

[5] https://en.wiktionary.org/wiki/proximity

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 2 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 maximum number of edges in a path, otherwise virtually every node in the graph will be in a path connecting the original nodes.

The proposed algorithm is formalized in Figure 1 using Java syntax[6]. The algorithm uses sets of paths segments catheterized by an intermediary node and its distance to the initial node. Each path segment is an instance of the class defined on the top of Figure 1. The algorithm starts by creating initial sets of path for each of the given terms. These sets have a single path with the node having as label each of the terms and the distance 0.

The algorithm proceeds recursively in successive sets. In each step the given sets of paths segments are intersected, and those found in the intersection form a complete path connecting the original node. They contribute to the proximity with the combined distance to the original nodes, divided by the cube of the step number. This way the longer the longer a path is, the less it contributes to proximity. Since this path is complete these path segments are removed from further processing. After processing the path segments in the intersection, if the maximum number of steps was not reached then each unused path segment is expanded before recursively adding the contribution to the proximity measure of the next step.

Figure 2 shows how the proximity algorithm proceeds to relate the nodes “Madonna” and “Britney Spears”. This examples omits the label nodes and starts with concept nodes associated with the relevant terms. We can see that each node is at the center of a pair of concentric circle. Each circle intersects a set of nodes that are reached from the center with a certain number of path segments. For instance, “Rock Music”, “Musical Artist” and “Pop Music” are all a path segment away from “Madonna”. A similar situation occurs with “Britney Spears” and some nodes are common to both circles, in this case “Musical Artist” and “Pop Music”. These two intermediary nodes contribute with two independent path connecting the original modes. The remaining nodes, “Rock music” for “Madonna” and “Dance Pop” for “Britney Spears” are used to continue unfolding the sets of nearby nodes connected to the original ones. In this case the node “Music genre” is common to both circles on the second level. This path is longer than the previous ones (i.e. has more path segments) and thus contributes less to proximity. At each level the contribution of new path is much less than those of the previous ones, although they are usually in greater number. After a few levels (typically 5) the algorithms stops.

The proximity algorithm can be extended to relate groups of concepts. This is relevant to relate two web pages, for instance. In this case one can extract are terms in a web pages and consider those that are labels to graph nodes. The proximity between the two node sets can be defined as the average, or the maximum, of all proximity pairs.

3.2 Implementation

The algorithm described in the previous section is implemented by a system called Shakti. This system is responsible for extracting data relevant to a given domain from DBpedia,

[6] For sake of clarity was used the Java 7 syntax.

Listing 1 The algorithm for computing proximity between nodes.

```
class Path {
  Node node = null;
  int distance = 0;
              // ...
}

public int proximity(String a, String b) {

  Set<Path> pathsA = new HashSet<>();
  Set<Path> pathsB = new HashSet<>();

  pathsA.add(new Path(nodeWithlabel(a), 0));
  pathsB.add(new Path(nodeWithlabel(b), 0));

  return proximity(pathsA, pathsB, 1);
}

private int proximity(Set<Path> pathsA, Set<Path> pathsB, int step) {
  int proximity = 0;

  if (pathsA.isEmpty() || pathsB.isEmpty())
    return proximity;

  Set<Path> unusedA = new HashSet<>(pathsA);
  Set<Path> unusedB = new HashSet<>(pathsB);

  for (Path pathA : pathsA)
    for (Path pathB : pathsB)
      if (pathA.node.equals(pathB.node)) {
        int step3 = step * step * step;
        proximity += (pathA.distance + pathB.distance) / step3;
        unusedA.remove(pathA);
        unusedB.remove(pathB);
      }

  if (step < MAX_STEP) {
    Set<Path> relatedA = related(unusedA);
    Set<Path> relatedB = related(unusedA);

    proximity += proximity(relatedA, relatedB, step + 1);
  }
  return proximity;
}

private Set<Path> related(Set<Path> paths) {
    Set<Path> related = new HashSet<>();

    for(Path path: paths)
      for(Arc arc: connectedTo(path.node)) {
        Property property = arc.property;
        Node target = arc.target;

        int distance =  path.distance + weigthOf(property);

        related.add( new Path(target,distance) );
      }
 }
```

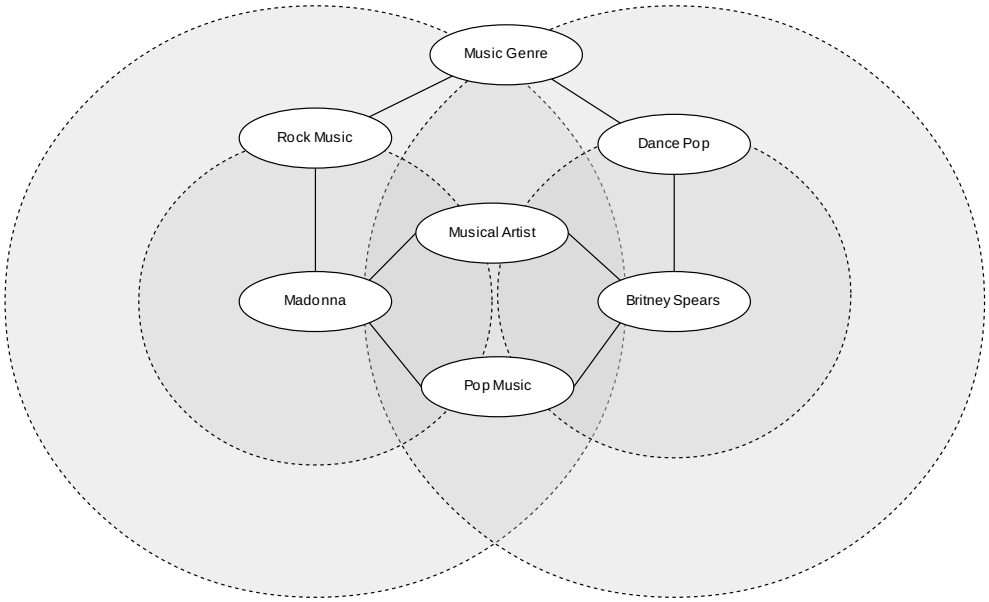


Figure 2 Using the proximity algorithm to relate "Madonna" and "Britney Spears."

to provide a measure of the proximity among concepts in that domain. This system is implemented in Java, an open-source semantic web toolbox called Jena[7] including application interfaces for RDF and OWL, a SPARQL engine, as well as parsers and serializers for RDF in several formats such as XML, N3 and N-Triples.

The overall architecture of a Shakti use case is described in the diagram in Figure 3. It shows that Shakti mediates between a client system and DBpedia, that in turn harvests its data from the Wikipedia. The system itself is composed of three main components:

controller is parametrized by a configuration file defining a domain and provides control over the other components;

extractor fetches data related to a domain from the DBpedia, pre-processes it and stores the graph in a local database;

proximity uses local graph to compute the proximity among terms in a pre-configured domain.

The purpose of the controller is twofold: to manage the processes of extracting data and computing proximities by proving configurations to the modules; and to abstract the domain required by client application. For instance, to use Shakti in a music domain it is necessary to identify the relevant classes on concepts, such as *musical artist*, *genre* or *instrument*, as well as the properties that connect them, such as type, *has genre* or *plays instrument*. To use Shakti in a different domain, say movies, it is necessary to reconfigure it.

The controller is parametrized by an XML configuration file formally defined by an XML Schema definition as depicted in Figure 4. The top level attributes in this definition configure general parameters, as the URL of the SPARQL endpoint, the natural languages of the labels

[7] https://jena.apache.org/

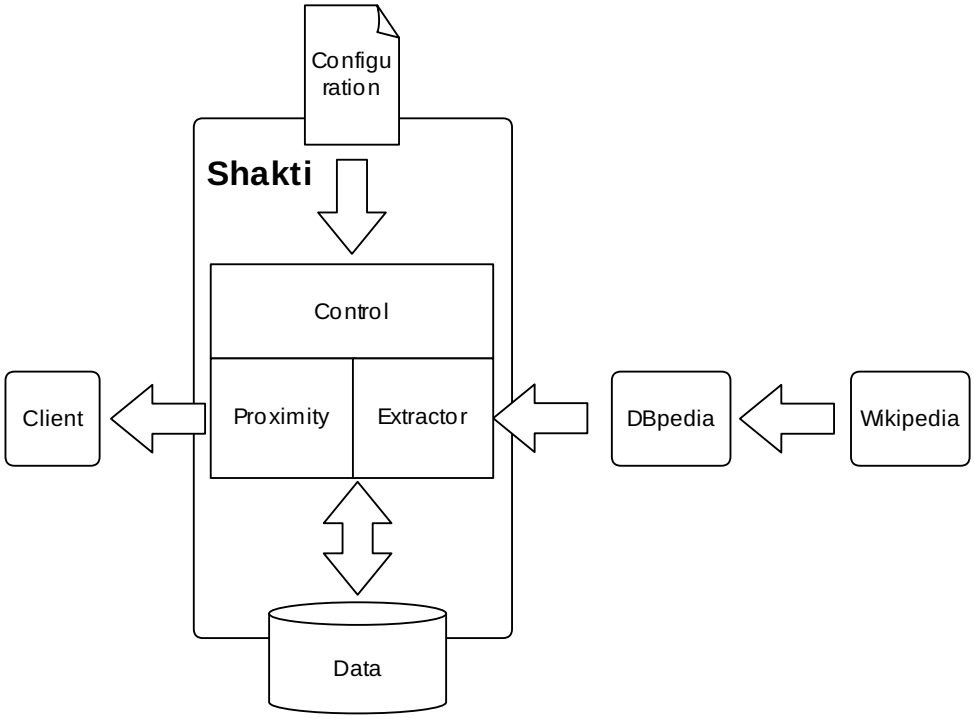


Figure 3 The architecture of Shakti.

(e.g. English, Portuguese), the maximum level used in the proximity algorithm, among others. The top level elements are used for defining prefixes, types and properties. XML prefixes are routinely used in RDF to shorten the URLs used to identify nodes. This configuration enables the declaration of prefixes used in SPARQL queries. The configuration file also enumerates the types (classes) of concepts required by a domain. This ensures that all the concepts with a declared type, having a label in the requested language are downloaded from DBpedia. The declaration of properties has a similar role but it also provides the weights assigned to path segments required by the algorithm. Each property definition includes a set of occurrences since the same name may be used to connect different types. That is, each property occurrence has a domain (source) and a range (target) and these must be one of the previously defined types. These definitions ensure that only the relevant occurrences of a property are effectively fetched from DBpedia.

The *extractor* retrieves data using the SPARQL endpoint of DBpedia. The extractor processes the configuration data provided by the controller and produces SPARQL queries that fetch a DBpedia sub-graph relevant for a given domain. Listing 2 shows an example of a SPARQL query to extract a type declared in the configuration file, where the string "[TYPE]" is replaced by each declared type. Similar queries are used for extracting properties.

Part of the data extracted this way, namely the labels, must be pre-processed. Firstly, multi-word labels are annotated incorrectly with language tags and must be fixed. For instance, a label such as `"Lady Gaga@en"` must be converted into `"Lady Gaga"@en`. Secondly all characters between parentheses must be removed. The Wikipedia, and consequently DBpedia, use parentheses to disambiguate concepts when needed. For instance, `"Queen`

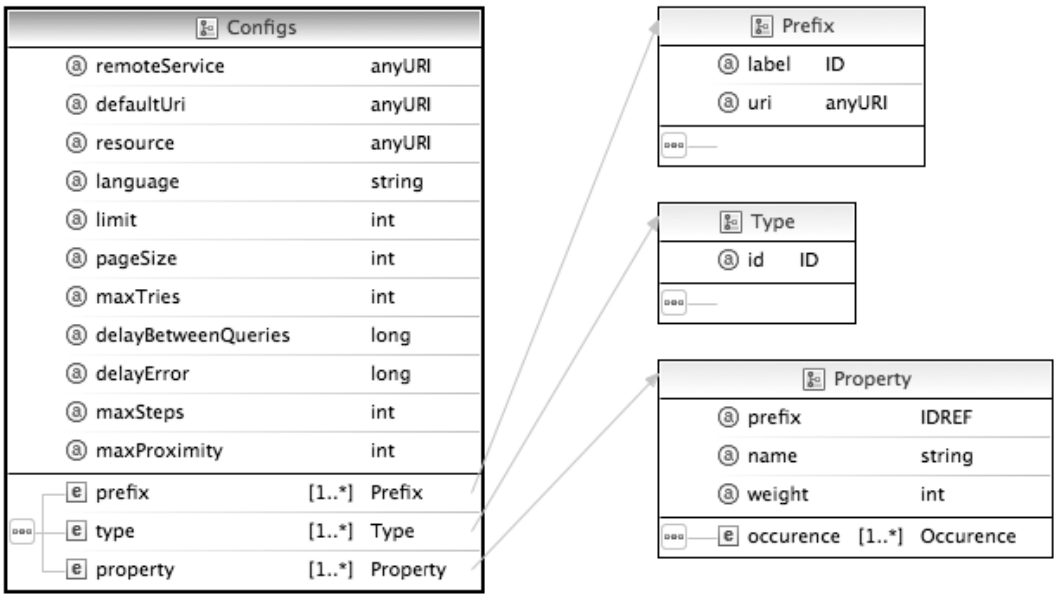


Figure 4 The XML Schema definition of Shakti configuration.

Listing 2 SPARQL query for extracting a type.

```
SELECT    ?R ?L
WHERE {
          ?R      rdf:type    dbpedia:[TYPE];
                  rdfs:label  ?L.
}
```

`(Band)"@en` is a different concept from `"Queen"@en` but in a music setting the term in brackets is not only irrelevant but would disable the identification with the term `"Queen"` when referring to the actual band. Also, concepts with short labels (less than 3 characters) or solely with digits (e.g. "23") are simply discarded.

The *proximity* module is responsible for computing the relatedness between two terms, or two bags-of-terms, from the graph extracted from DBpedia and already pre-processed. This module maintains a dictionary with all labels in the graph, implemented using a prefix tree, or *trie*. This data structure enables an efficient pre-search of terms, discarding the terms for which relatedness cannot be computed. Following this step, the implementation follows closely the algorithm describes in listing 1.

4 Evaluation

This section presents a use of Skati in the implementation of a recommender developed as part of the project Palco 3.0. This project was targeted to the re-development of an existing Portuguese social network — Palco Principal — whose main subject is alternative music.

The goals of this project include the automatic identification, classification and recommendation of site content. The recommendation service developed for this project is structured around recommenders — pluggable components that generate a recommendation for a certain request based on a given model. Most of the recommenders developed for this service use collaborative filtering. For instance, a typical recommender suggest songs to users in Palco Principal based on the recorded activity of other users. If a user shares a large set

of songs in his or her playlist with other users then it is likely that he or she will enjoy other songs in their playlist.

This approach is very effective and widely used but its main issue is cold start. If the system has no previous record of a new user then it will not be able to produce a recommendation. An alternative is to produce a content-based recommender. To implement such a recommender Shakti was used to find related content on the web site. This recommender can be used on words extracted from the web page itself, such as news articles or interviews, or on tags used to classify web pages, such as musics, photos of videos.

The remainder of this section describes the main steps to define a content recommender for Palco Principal using Shakti and how this experiment was used to evaluate this approach.

4.1 Proximity Based Recommendation

Palco Principal is a music website hence this is the domain that must be used in Shakti. This required selecting DBpedia classes and properties relevant to this domain, preparing DBpedia for extracting data from the Portuguese wikipedia to populate these classes, and configuring Shakti with the relevant types and properties to compute proximities.

DBpedia already has an extensive ontology covering most of the knowledge present in Wikipedia. This is certainly the case with the music domain and all the necessary classes and properties were already available. The DBpedia uses a collection of mapping to extract data present in the info boxes of Wikipedia. Unfortunately these mapping were only available for the English pages of Wikipedia and they had to be adapted for the pages in Portuguese. The DBpedia uses a wiki to maintain these mapping and new mappings of some classes had to be associated with the language label "pt".

In the Shakti it was necessary to configure the XML file to extract the selected classes and properties from DBpedia. These classes, whose mappings were created on DBpedia wiki for Portuguese pages, are:

MusicalArtist solo performers (e.g. Madonna, Sting);

Band groups of musicians performing as a band (e.g. Queen, Bon Jovi);

MusicGenre musical genres (e.g. rock, pop).

The properties associated with these classes that were considered relevant were also inserted in the configuration file and are enumerated in Table 1. This table defines also the weights assigned to properties, with values ranging from 1 to 10, needed for computing proximities. These weights were assigned based on the subjective perception of the authors on the proximity of different bands and artists. A sounder approach to weight calibration was left for future work.

Table 1 Properties.

Property	Type	Classes	Weight
Genre	MusicGenre	Band and MusicalArtist	7
Instrument	Label	Band and MusicalArtist	2
StylisticInfluences	Label	MusicGenre	4
AssociatedBand	Band	Band	10
AssociatedMusicaArtist	MusicalArtist	MusicalArtist	10
CurrentMember	Label	Band	5
PastMember	Label	Band	5

To integrate Shakti with the recommender it was necessary to implement a client application. This application is responsible populating a table with proximities among web pages recorded on the recommender service database. For each page this client application extract a bag-of-words, either the words on the actual page of the tags to classify it. For each pair of bags-of-words it computes a proximity using methods provided by Shakti.

4.2 Results Analysis

Shakti is currently being used in an experimental recommender. Thus, the recommendations are not yet available on the site the Palco Principal. For this reason a comprehensive analysis is not yet possible. This subsection presents some experimental results that are possible to obtain from the current setting.

For this experiment the recommender system computed proximities among news and events pages, which took about a day. In total 57,982 proximities relations among news pages were calculated, plus 59992 among event pages, performing a grand total of 69604805 relations.

Table 2 Proximity between pairs of news pages.

Resource ID	Resource ID	Proximity
3540	2623	0.22
3540	2431	0.21
3540	3000	0.15
3540	4115	0.15
3540	2691	0.15
3540	1892	0.15
3540	2676	0.14
3540	760	0.14
3540	3189	0.14
3540	4397	0.14

Table 2 displays the proximity table for news pages ordered by decreasing proximity. Each id code is a news item in the web site. For this particular entity the recommender searched for content regarding both terms from its text and tags.

To analyze the performance of Shakti the contents of the 2 most related pages — id 3540 (resource A) and id 2623 (resource B) — were compared in detail. The text and tags of this resource can be viewed in Figure 5. In order to calculate proximities, Shakti merge both fields and generates a group of concepts present in the RDF graph. Thus, from all the words of *text* and *tags* fields only the following bag-of-words are actually used to compute proximity: 38 Special, Lynyrd Skynyrd, Bret Michaels. For resource B the bag-of-words considered for computing compute proximity is: Lemmy, Myles Kennedy, Andrew Stockdale, Dave Grohl, Fergie, Ian Astbury, Kid Rock, M. Shadows, Rock, The Sword, Adam Levine, Ozzy Osbourne, Chris Cornell, Duff McKagan, Slash, Iggy Pop. Using these two bags-of-words Shakti computes a proximity of 0.22. The concepts are names of the bands appearing in news text, so the approach of using the this field to determine proximity seams promising.

Analyzing these news items one notices that they are on two musician artists with a musical genre in common, and both playing the guitar. This shows that the two news items are in fact related and a 0.22 proximity seams a reasonable figure. Note that proximities range between 0.0 (unrelated) to 1.0 (the same).

Figure 5 News piece generated fromabout resource A.

The proximity values computed for all pages vary between 0.1 and 0.22 and the average value of about 0.2. This value is lower than expected. Of course that these figures can be modified simply by reconfiguring the property weights. On the other hand, Shakti determined a non null proximity in 24,401 of a total of 33,616,804 possible relationships, about 0.07%, which is an unsatisfactory figure for a recommendation system.

One of the culprits for these poor results is the text encoding using HTML entities in the database of Palco Principal. For instance, the term "Guns N' Roses" (which is part of the text and tags of resource B) is written in the database in the format "Guns N&#039 Roses". This value is sent to the Shakti. As Shakti is not prepared to receive this type of formatting, it does not detect the word in the dictionary.

Nevertheless, the problems with text encoding alone do not justify the low number recommendations obtained in this experiment. Most probably the sub-ontology extracted from DBpedia does not cover satisfactory the domain of the Palco Principal.

5 Conclusions and Future Work

The goal of the research described in this paper is to measure the relatedness of two terms using the knowledge base of BDpedia. The motivation for this research is to use semantic relatedness in content-based recommenders, in particular in tags provided by users in social networks.

This paper proposes an algorithm to compute the semantic relatedness of two terms as proximity rather than distance, as in similar ontology based approaches. The algorithm ponders the collection on paths connecting the two terms rather using the weights associated to properties on the ontological graph. This algorithm was implemented in a system called Shakti. This system fetches a sub-graph of the ontology in DBpedia relevant to a certain domain and computes the relatedness of terms assigned as labels to concepts. To validate the proposed approach Shakti was used to populated a proximity table on a web recommended service of Palco Principal, a Portuguese social network whose subject is alternative music. The results are promising, although the ontology extracted form DBpedia is not yet covering satisfactory the terms contained on the pages of Palco Principal.

As part of the future work in this research we plan to experiment with larger ontologies, providing better coverage of the underlaying domain and validating scalability of Shakti. At this stage most of the effort of using Shakti is configuring this tool. We plan the development of a graphical user interface for assisting the tool users in defining the classes and properties to extract from DBpedia. There are two approaches being considered for this task. On the first approach a seed class is typed in and other related classes and properties in that

domain are suggested for possible inclusion. On the second approach Shakti is fed with a collection of example terms and DBpedia is searched for classes for those terms, as well as relates properties. Independently from the selected approach, the graphical user interface will also assist in the definition of property weights and other general configurations required by Shakti.

Acknowledgments This work is in part funded by the ERDF/COMPETE Programme and by FCT within project FCOMP-01-0124-FEDER-022701. The authors wish also to thank the reviewers for their helpful comments.

References

1 Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

2 D. Beckett and B. McBride. Resource description framework (RDF) model and syntax specification (revised). Technical report, W3C, 2004.

3 D. Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema. Technical report, W3C, 2004.

4 Óscar Corcho and Asunción Gómez-Pérez. A roadmap to ontology specification languages. In *Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, EKAW '00, pages 80–96, London, UK, UK, 2000. Springer-Verlag.

5 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*, 41:391–407, 1990.

6 Christiane Fellbaum, editor. *WordNet An Electronic Lexical Database*. The MIT Press, Cambridge, MA ; London, May 1998.

7 Evgeniy Gabrilovich. Feature generation for textual information retrieval using world knowledge. *SIGIR Forum*, 41(2):123–123, December 2007.

8 Evgeniy Gabrilovich and Shaul Markovitch. Overcoming the brittleness bottleneck using wikipedia: enhancing text categorization with encyclopedic knowledge. In *proceedings of the 21st national conference on Artificial intelligence - Volume 2*, AAAI'06, pages 1301–1306. AAAI Press, 2006.

9 J.J. Jiang and D.W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. In *Proc. of the Int'l. Conf. on Research in Computational Linguistics*, pages 19–33, 1997.

10 Dekang Lin. An information-theoretic definition of similarity. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, pages 296–304, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

11 David Milne and Ian H. Witten. Learning to link with wikipedia. In *Proceedings of the 17th ACM conference on Information and knowledge management*, CIKM '08, pages 509–518, New York, NY, USA, 2008. ACM.

12 M. Nilsson, M. Palmer, and J. Brase. The LOM RDF binding - principles and implementation. In *3rd Annual ARIADNE Conference*, 2003.

13 Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, IJCAI'95, pages 448–453, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

14 N. Seco, T. Veale, and J. Hayes. An intrinsic information content metric for semantic similarity in WordNet. *Proc. of ECAI*, 4:1089–1090, 2004.

15 I. Smirnov. Overview of stemming algorithms. *Mechanical Translation*, 2008.

16 Michael Strube and Simone Paolo Ponzetto. Wikirelate! computing semantic relatedness using wikipedia. In *proceedings of the 21st national conference on Artificial intelligence - Volume 2*, AAAI'06, pages 1419–1424. AAAI Press, 2006.

17 Fei Wu and Daniel S. Weld. Automatically refining the wikipedia infobox ontology. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 635–644, New York, NY, USA, 2008. ACM.

18 Torsten Zesch and Iryna Gurevych. Automatically creating datasets for measures of semantic relatedness. In *Proceedings of the Workshop on Linguistic Distances*, LD '06, pages 16–24, Stroudsburg, PA, USA, 2006. Association for Computational Linguistics.

Query Matching Evaluation in an Infobot for University Admissions Processing

Peter Hancox[1] **and Nikolaos Polatidis**[2]

1 **School of Computer Science, University of Birmingham**
Edgbaston, Birmingham, B15 2TT, United Kingdom
pjh@cs.bham.ac.uk
2 ***Formerly of:*** **School of Computer Science, University of Birmingham**
Edgbaston, Birmingham, B15 2TT, United Kingdom

Abstract

"Infobots" are small-scale natural language question answering systems drawing inspiration from ELIZA-type systems. Their key distinguishing feature is the extraction of meaning from users' queries without the use of syntactic or semantic representations. Two approaches to identifying the users' intended meanings were investigated: keyword-based systems and Jaro-based string similarity algorithms. These were measured against a corpus of queries contributed by users of a WWW-hosted infobot for responding to questions about applications to MSc courses. The most effective system was Jaro with stemmed input (78.57%). It also was able to process ungrammatical input and offer scalability.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases chatbot, infobot, question-answering, Jaro string similarity, Jaro-Winkler string similarity

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.149

1 Introduction

University student recruitment administration is an application where there is potential for a large volume of enquiries of a fairly routine and predictable nature from a world-wide pool of applicants. The costs of call centres (both in terms of running the centres and recruiting and retaining a knowledgeable workforce) make such ventures unattractive. On the other hand, it should be possible to implement a technological solution beyond adding over-large FAQs to web pages.

Student recruitment, particularly at graduate level, is international in outlook: in UK postgraduate computing degrees, it is not unusual for international students to outnumber UK students by two to one. Communicating with international applicants brings with it all the problems of understanding versions of English that can vary from the excellent to the less than adequate. Most applicants have a strong commitment to accessing the information that they need to be able to make a choice of what and where to study. Varying language capabilities notwithstanding, many applicants are very articulate (and so able to frame precise requests for information) and discriminating in making their choices based on that information. Although universities work hard at providing prospectus information that ranges from detail of the structure of courses to the accommodation packages and sports facilities available, many applicants still find themselves wanting to ask highly targeted questions about, for instance, module choice deadlines and visa requirements. Given the wide range of time zones that applicants come from and that 24 hour access to staff is not a

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 149–161

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

possibility, the work reported here emanated from projects to implement a web-based infobot application for answering applicants' enquiries.

A number of organizations (for instance in the UK, the Alliance & Leicester bank and the O_2 mobile phone company) have made chatbots available on their websites. Chatbots have a long history and the first implementation, ELIZA [8], is particularly famous. Recent bots owe much to the ALICE software which consists of a matching engine and an XML-based language for writing rules to transform input [1].

We argue that the use of traditional chatbots for informational systems is inappropriate. Writing of the first extensive script he prepared for ELIZA which parodied "the responses of a nondirective psychotherapist", Weizenbaum [8, p. 188] wrote "I chose this script because it enabled me to temporarily sidestep the problem of giving the program a database of real-world knowledge." ELIZA was a *conversational* agent: whatever happened and wherever the dialogue went, the aim was to keep the conversation going. Being a nondirective psychotherapist, information content was low, the main tactic being to use keywords from the user's input to fill slots in the system's output. In the following, ELIZA [8, p. 2] has used *boyfriend* and *depressed* to fill pre-prepared templates:

```
User:   Well, my boyfriend made me come here.
ELIZA:  Your boyfriend made you come here.
User:   He says I'm depressed much of the time.
ELIZA:  I am sorry to hear you are depressed
```

To avoid the conversation becoming too repetitive, ELIZA used two ways of introducing variety into its responses. For each keyword there could be a number of possible responses; each was used in turn to introduce some variation. Also previous topics were stored on a stack so that, should it be impossible to match a keyword with a template, a previous keyword could be revisited. This had the significant effect of making it seem as if there was some larger dialogue management taking place.

The ELIZA/ALICE model is essentially conversational: the chatbot attempts to maintain a dialogue exchange above all else. The communication of information is very much a secondary objective; hence Weizenbaum's choice of a nondirective psychotherapist.

Both the Alliance & Leicester and O_2 chatbots try to communicate information about products while trying to maintain a dialogue. While it might seem attractive from a marketing point of view to present the user with a "chatbot friend" in the hope they will bond with it, many users must be sufficiently ICT-literate and the chatbots so limited that the illusion of a conversational friend is shattered. However, behind such systems, the information content is equivalent to an over-large FAQ. This paper focuses on providing a natural language interface to a set of FAQ-like topics where the number of topics is too large for a conventional WWW-based FAQ and too small for a full database NL interface system. More specifically, the aims of the NL interface investigated here can be stated as:

1. *robustness* - capable of processing well-formed English or ill-formed either because the user's command of English is poor or because of ellipsis;
2. *low cost* - such a system should use relatively simple techniques to extract meaning from input and return outputs, thus reducing the cost of implementation and maintenance;
3. *low-skilled maintenance* - it is essential that adding to and modifying the knowledge base of the application should be as simple as possible, allowing changes to be made by IT literate rather than computer science trained colleagues.

As explained above, the context of this investigation was a system for responding to NL enquiries about applications to MSc courses. Such a system would consist of a WWW interface to a bank of 50-100 topics (i.e. too many for a manageable FAQ). Two main ways of accessing the bank of topics were chosen:

- *keywords* - keywords were manually assigned to each topic, together with a weight in the range 1...5 (where 1 was relatively insignificant and 5 extremely significant);
- *sentences* - one or more stereotypical interrogative sentences were assigned to each topic. No weights were assigned to these sentences.

In both cases, it would be relatively easy for non-computer scientists to annotate the topic banks. This system is termed an "infobot" to distinguish its informational and non-conversational functionality from that of chatbots.

Experiments were designed to assess the effectiveness of a number of methods of matching queries with either sets of keywords or sentences. (String similarity algorithms were used for the latter.)

2 Claims

The main claim made as a result of the experiments is that:

- A Jaro-based string similarity algorithm [3] is at least as effective as the less complex keyword-based methods tested and offers better scalability.

Sub-claims are:

- Abbreviated, terse queries (e.g. "cost of courses") and lengthy inputs have no significant effect on the performance of the best-performing matching algorithms.
- The best performing matching algorithms are robust when processing "non-native" English.

The methodology was first to establish a corpus of queries from users. This was used as the basis for building the keyword and sentence indexes. Then, each matching method was applied to the corpus to provide a basis of comparison.

3 Preparing a Corpus

To collect a sample of inputs, a simple keyword-based infobot for delivering admissions-related information in response to natural language queries was mounted on the WWW.

This infobot was implemented in SICStus Prolog with a PrologBeans interface to the Java front-end. Users' inputs were delivered to the Prolog application which extracted keywords or key-phrases and used these to match with keywords or key-phrases associated with pre-prepared text (Table. 1).

The system was made accessible via the WWW to applicants for MSc courses in the School of Computer Science, University of Birmingham [7] in two phases.

3.1 Phase 1: Initial Testing

This was a feasibility study in which 77 applicants (with surnames beginning with 'S' or 'T') were invited to use the system which contained templates based on email enquiries from the previous year's application round. 121 queries were submitted and outcomes analysed, allowing for the addition of further rules and, more particularly, the assignment of more keywords to texts.

Table 1 Rules and keywords from the simple chatbot.

Prepared text	Keywords
Our programmes begin on 4th October 2010. Next academic year begins on 26th September 2011.	begin beginning 'academic year' 'starting date'
The on-line application form is at: http://apply.bham.ac.uk/cp/home/loginf.	'online application'

3.2 Phase 2: Corpus Collection

The second phase was used to collect a reference sample of queries that might be used to evaluate later systems, to analyse the behaviour of users and to analyse the performance of this simple system. 573 applicants were invited to use the system (being applicants with surnames beginning with other than 'S' or 'T'). 357 queries were recorded of which 70 were repeats[1].

All inputs and responses were logged. Each input was manually annotated as one of:

- *Correct* - the input was judged to be grammatical, correctly spelled and the question appropriate to the domain.
- *Correct/spelling error* - an otherwise correct input that contains at least one spelling error.
  Examples: How long it takes to finish the *porgram*? How do I know if my online registration is *finnished*?
- *Correct/grammar error* - an otherwise correct input that contains at least one grammatical error.
  Examples: Do i require to attend an interview? Is there any part time programs?
- *Abbreviated* - an input that was too brief (usually lacking a verbal component) for keywords to be reliably identified.
  Examples: Registration? FAQ? why Birmingham?
- *Inappropriate* - the input was either judged to be grammatical, correctly spelled but the question inappropriate to the domain or the input was not English or not natural language.
  Examples: What time is it now? What is your name? Das ist ein scholarship! Mumble-Jumble,ISupposeThisIsATest, ????.

3.3 Analysis of Users' Inputs

In the email inviting applicants to take part in the trial, it was explained to them that this was a system under development that needed testing. An analysis of the input shows that a substantial number of the enquiries were well-formed and relevant English questions. Some applicants chose to use abbreviated enquiries such that they might use in a general search engine. Inevitably, in the context of a test where there was no identification of individual users, some chose to enter completely irrelevant (and thus inappropriate) queries (Fig. 1).

From the log of inputs it could be seen that some users immediately followed their original query with one or more repetitions of the input as if they believed that a repetition

[1] A repeat is defined as a user immediately entering an input identical to their previous input.

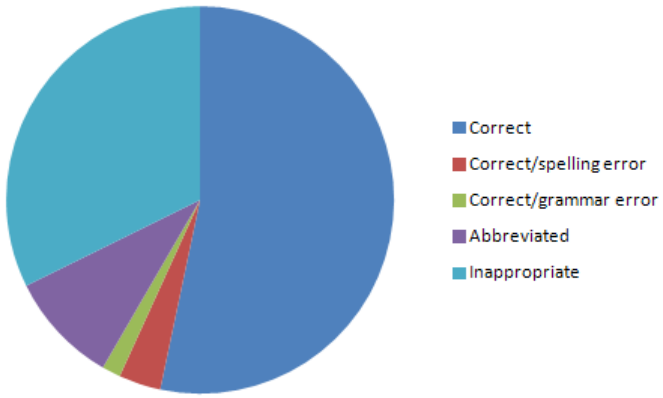


Figure 1 Classification of inputs.

Table 2 Classification of inputs.

Label	Original input		Repeated input		Total input	
	n	%	n	%	n	%
Correct	105	76.64	32	23.36	137	100.00
Correct/incorrect spelling	7	77.78	2	22.22	9	100.00
Correct/incorrect grammar	4	100.00	0	0.00	4	100.00
Abbreviated	22	91.67	2	8.33	24	100.00
Inappropriate	49	59.04	34	40.96	83	100.00

would, for some reason, return an alternative response (Table 2). It is very noticeable that users' willingness to repeat input was determined by the nature of their original input. 23.36% of correct inputs were repetitions, whereas only 8.33% of abbreviated queries were repeated, suggesting users realised that their input was too brief. The number of repetitions of inappropriate input was particularly large at 40.96%, perhaps suggesting that such users had a poor initial model of the system and were struggling to refine that model.

3.4 From Input to Corpus

To build a corpus as a tool for testing alternative designs, the inputs were selected as follows. All correct inputs were kept as were correct/grammar errors inputs. Correct inputs with spelling errors were corrected and (unless already present in the corpus) included. Abbreviated inputs were included where it was possible to glimpse some intended meaning. The corpus consisted of 154 queries, including well-formed and less well-formed questions as well as terse non-grammatical queries. Thus the corpus could claim to represent a real-life variety of English performance. The median length of queries was 6.19 words and the mode was 5 words.

Each query in the corpus was assigned to one of the infobot's responses. For instance, the input "how long does it take to pursue a master program?" was labelled as a "duration" so that the query would be given the response "Our MSc programmes last for one year". There were 69 distinct response classes. A few response classes were very closely related, for instance "birmingham_location" ("Where is Birmingham") and "location_university" ("Where is Birmingham University"). Such similarity would make the task of retrieval more difficult but reflected the practical difficulties of responding to some queries. Two topics dominated

others in the corpus: the cost of tuition fees and the availability of scholarships. There was a noticeable difference between the contents of emails previously sent to admissions tutors and infobot queries. When applicants realised they were communicating with a machine, they felt sufficiently uninhibited to ask about money issues.

4 Experiments on Matching Methods

The matching methods used fell into two groups: those that used keywords and those that used the stereotypical questions. The results of each experiment were classified into one of three categories:

1. *Correct* - the outcome matched the expected outcome given in the corpus;
2. *Incorrect* - the outcome did not match the expected outcome given in the corpus;
3. *No response* - no match was made, for instance because the user's query was irrelevant to the domain of the system.[2]

4.1 Keyword-based Matching

Words judged to be significant were manually added to the keyword set.[3] In the following queries from the corpus, the keywords have been underlined:

how many modules
what is the last date of submitting the recommendations

Weights were manually assigned to each keyword, with low weight attached to meaningful but commonly used keywords ("how many" = 1) to high weight to those keywords thought to carry the main content of their queries ("recommendations" = 4). Each keyword was associated with one or more *interpretations*; an interpretation here meaning the label of a particular response, for instance the duration example (Sec. 3.4). There were 152 keywords indexing 76 topics.

4.1.1 Simple Keyword Matching

This method of matching was not expected to be effective but was used to provide a baseline method against which all other methods could be compared. In the first experiment, weights were ignored. Competing interpretations were judged solely by the number of keywords found in the input. So, if the underlined words are keywords that shared the same interpretation (deadline_application):

what is the last date of submitting the recommendations

the score for the deadline_application interpretation was 3. Where there was a tie between two or more interpretations, the first occurring interpretation was selected.[4] Results are given in Table 3.

---

[2] In these experiments, the use of a corpus that excluded irrelevant queries meant that "no response" would be indicative of system failure rather than irrelevant input.
[3] Here "keyword" in understood to mean both single word and multi-word keywords, e.g. "part time".
[4] In a practical system, it would be necessary to employ some way of choosing between tied interpretations, for instance by allowing the user to choose the response best suited to their query. This, however, is an evaluation where the emphasis is on mechanically selecting the most appropriate interpretation.

4.1.2 Weighted Keyword Matching

Here the weights were summed. So, if the underlined words are keywords that shared the same interpretation (deadline_application):

what is the last date of submitting the recommendations

and their weights were:

what — deadline_application — 1
last date — deadline_application — 3
recommendations — deadline_application — 1

the sum was 5. Where there was a tie, the first occurring interpretation was selected. Results are given in Table 4.

4.1.3 Simple/Weighted Keyword Matching

The sum of the weights and the number of keywords found were summed. Again, using the example:

what is the last date of submitting the recommendations

where the simple keyword score was 3 and the weighted keyword score was 5, the simple weighted keyword was 8. Where there was a tie, the first occurring interpretation was selected. Results are given in Table 5. (It might seem more reasonable to calculate the average weight of keywords by dividing the summed weight by the number of keywords but this gave a slightly worse performance.)

4.2 Sentence-based Matching (string similarity)

One or more stereotypical queries were written for each interpretation. For instance, for the "duration" interpretation, the stereotypical queries were:

how long does a masters degree take?
how long does the program take?
how long does the programme take?
how long is the msc?
what is the duration of the course?
what is the duration of the program?
what is the duration of the programme?

The matching process was to compute the string similarity between input (here drawn from the corpus) and the stereotypical queries. There are a number of string similarity algorithms that could be used [2]. Those selected were:

- *Jaro proximity*[5] (comparing inputs/stereotypical questions forwards and backwards);
- *Jaro-Winkler proximity* (forwards and backwards).

[5] Confusingly, "proximity" and "distance" seem to be used interchangeable in the literature.

These algorithms were devised tor comparing strings such as personal names where strings would be short and errors likely to be transpositions over fairly short distances.

The Jaro algorithm compares two strings such as 'Martha' and 'Marhta'. One string is scanned, character-by-character. (In this example, 'Martha' is taken as the first string.) A moveable window is placed over the second string. The width of the windows is computed as half the length of the longer string - 1. The window moves in synchrony with the scanning of the first string. A match between a character in the first string can only occur within the window. In the example, the emboldened characters are matches while underlined characters are within the current window:

Martha	M**a**rtha	Ma**r**tha	Mar**t**ha	Mart**h**a	Martha
Marhta	M**a**rhta	Ma**r**hta	Mar**h**ta	Mar**h**ta	Marht**a**

In a second scan, the number of transpositions is counted. The calculation of Jaro proximity is:

$$\frac{1}{3} \times \frac{matches}{length(string_1)} + \frac{matches}{length(string_2)} + \frac{matches - (transpositions//2)}{matches} \tag{1}$$

(It should be said that the detailed implementation of transposition matching is not intuitive: "The number of transpositions . . . is computed somewhat differently from the obvious manner." [9, p. 10].)

The Jaro-Winkler algorithm is founded on the observation that transposition errors are less likely to occur in names or addresses within the initial n character positions (usually set to 4). Winkler extended the Jaro algorithm by adding a threshold of similarity (usually 0.70). For two strings with a Jaro proximity of 0.7 or more, the initial n characters are matched for absolute similarity (giving a 'match length'). Thus, Jaro-Winkler proximity is calculated as:

$$JaroProximity + (length(match) \times position \times (1.0 - JaroProximity)) \tag{2}$$

Jaro [3] and Jaro-Winkler [10] algorithms have a record of good performance [2]. Whilst developed for character-by-character processing of names, in these experiments the comparison was word-by-word and thus inputs in these experiments were relatively short and had a number of words comparable to the number of letters in names. It should be noted that the proportion of matching words (either directly aligned or transposed) was lower than the proportion of matching characters in a personal name [5].

4.2.1 Jaro Proximity String Similarity

The standard Jaro algorithm uses a matching window defined as:

$$\frac{max\left(length\left(string_1\right), length\left(string_2\right)\right)}{2} - 1 \tag{3}$$

A number of runs were tried to investigate the effect of longer window sizes, leading to the conclusion that Jaro's original window size was optimal.

Two experiments were carried out: searching from beginning to end of input/stereotypical sentences (Table 6); searching from end to beginning to end (Table 7).

4.2.2 Jaro-Winkler Proximity String Similarity

This modification of the Jaro algorithm rewards matches at the beginning of the two strings, specifically in the first four positions. It was used in these experiments because it seemed that the beginning of a query (e.g "how many ...", "are there any ...") was significant in the query's meaning. It was hypothesised that it would be more significant still for comparing the endings of queries because many questions in English begin with the same sequence of words, thus the endings of queries should be more discriminating. The results are presented in Tables 8 and 9.

Table 3 Simple keyword matching: results.

Outcome	n	%
Correct	105	68.18
Incorrect	49	31.82
No response	0	0.00
Total	154	100.00

Table 4 Weighted keyword matching: results.

Outcome	n	%
Correct	118	76.62
Incorrect	36	23.38
No response	0	0.00
Total	154	100.00

Table 5 Simple and weighted keyword matching: results.

Outcome	n	%
Correct	119	77.27
Incorrect	35	22.73
No response	0	0.00
Total	154	100.00

Table 6 Jaro (forward): results.

Outcome	n	%
Correct	118	76.62
Incorrect	26	23.38
No response	0	0.00
Total	154	100.00

Table 7 Jaro (backwards): results.

Outcome	n	%
Correct	105	68.18
Incorrect	49	31.82
No response	0	0.00
Total	154	100.00

Table 8 Jaro-Winkler (forward): results.

Outcome	n	%
Correct	117	75.97
Incorrect	37	24.03
No response	0	0.00
Total	154	100.00

Table 9 Jaro-Winkler (backwards): results.

Outcome	n	%
Correct	104	67.53
Incorrect	50	32.47
No response	0	0.00
Total	154	100.00

4.3 Jaro/Jaro-Winkler with Stemming

Both the forward and backwards versions of the Jaro and Jaro-Winkler algorithms were supplemented by stemming the input and stereotypical queries. The stemming algorithm used was the Porter algorithm [6]. The query:

what is the last date of submitting the recommendations

would be reduced to:

what i the last dat of submit the recommend

Results are given in Tables 10 to 13.

Table 10 Jaro (forward-stemmed): results.

Outcome	n	%
Correct	121	78.57
Incorrect	33	21.43
No response	0	0.00
Total	154	100.00

Table 11 Jaro (backwards-stemmed): results.

Outcome	n	%
Correct	106	68.83
Incorrect	48	31.17
No response	0	0.00
Total	154	100.00

Table 12 Jaro-Winkler (forward-stemmed): results.

Outcome	n	%
Correct	119	77.27
Incorrect	35	22.73
No response	0	0.00
Total	154	100.00

Table 13 Jaro-Winkler (backwards-stemmed): results.

Outcome	n	%
Correct	106	68.83
Incorrect	48	31.17
No response	0	0.00
Total	154	100.00

5 Interpretation of Results

Jaro-Winkler (backwards) was the worst-performing method. The range between the worst (67.53%) and the best-performing methods (78.57%) is surprisingly narrow.

There is little to choose between Jaro (forward-stemmed) (78.57%), simple/weighted keywords (77.27%) and Jaro-Winkler (forward-stemmed) (77.27%).

5.1 Simple Matching

The simple method (here used for baseline comparison) is almost the least reliable because its only strategy of choice is the number of keywords. So from Fig. 2, it can be seen that when word length rises beyond three, the simple method tends to perform less well. Essentially, given the restricted domain (and hence vocabulary of the application) the more words a query such as:

by when do I need to accept a course offer?

contains, the more probability there is that the query contains keywords for an inappropriate interpretation. The likelihood of incorrect interpretations was compounded by the lack of a principled way of selecting between alternatives.

5.2 Simple/Weighted Matching

The performance of simple/weighted matching was almost as good as the best method. There were some queries which it processed incorrectly but which the other methods processed correctly. Examples are:

why Birmingham University?
why study at Birmingham University?

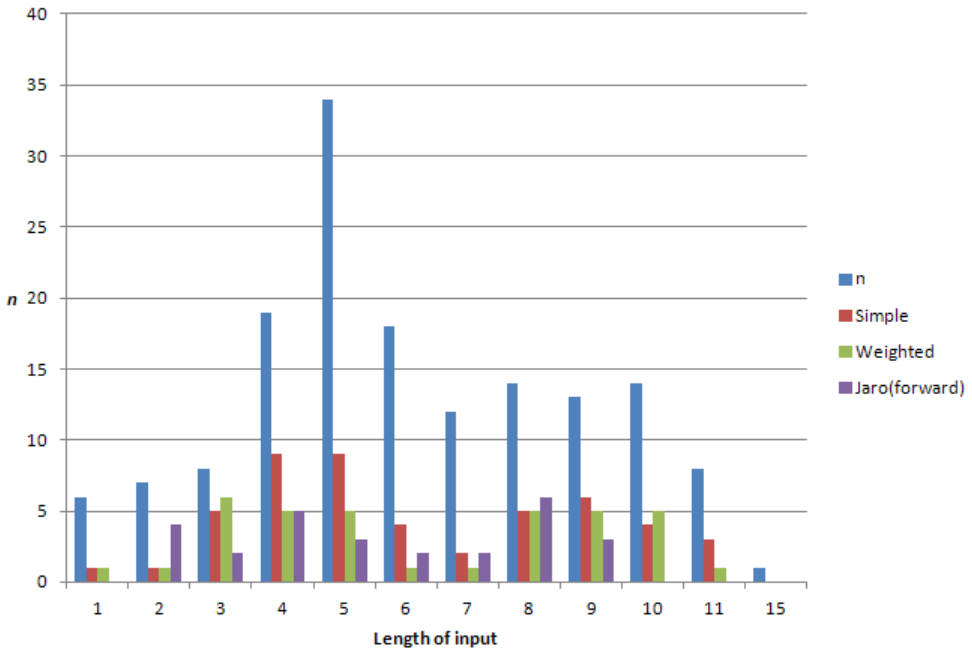


Figure 2 Incorrect interpretations by method of matching.

In both cases, it provided an interpretation for the very closely related query:

why should I come to Birmingham? [6]

It seems at this level of subtlety, the simple/weighted matching method was unable to distinguish satisfactorily between competing interpretations.

5.3 Jaro (forward-stemmed)

Of the eight Jaro-based methods, Jaro (forward-stemmed) was most effective (78.57%). It might have been expected to work even better. Fig. 2 fails to record any particular pattern to failures amongst three similarly scoring methods, for instance they did not mainly occur amongst queries of shorter lengths. Neither did the errors occur amongst longer queries. The stemming algorithm worked to reduce variability between users' expressions. Thus users' variability of expression became less significant and one stereotypical sentence would match with a larger number of users' queries. This is supported by an examination of queries which keyword methods could resolve correctly but which Jaro (forward-stemmed) failed. The most extreme was:

when do I need to finalize my course optional modules? [7]

Without a sufficiently similar stereotypical sentence, it would be more a matter of luck if the nearest matching stereotypical sentence had an appropriate interpretation.

[6] Where this sentence is understood to mean the city of Birmingham.
[7] A simpler way of expressing this might have been: "what is the deadline for choosing optional modules?"

5.4 Scalability

The corpus was, at 154 entries, small-scale. Nonetheless, it is possible to discern some problems of scalability even at this size. Simple/weighted matching failed where it had to choose between two closely related interpretations. An attempt to increase the success rate from 77.27% would require, in part, more keywords. This evaluation suggests that increasing keywords in a limited domain (with the likelihood that one keyword will index multiple interpretations) would bring a decrease in accuracy.

While it was difficult to see a consistent pattern of failure for Jaro (forward-stemmed) (78.57%), there is some evidence that a lack of stereotypical sentences was a cause of failure. Thus an increase in the coverage would, unlike simple/weighted matching, improve performance. In summary, Jaro (forward-stemmed) has the potential for scalability; simple/weighted matching does not.

5.5 Lack of Input *v.* correctness

It was hypothesised that it would be more difficult to answer shorter queries correctly. Fig. 2 gives only very limited evidence of this. At a query length of two, Jaro (forward) did badly; at query length of three words, keyword-based matching did less well. However, there is no clearly significant evidence and so the hypothesis can be neither confirmed nor denied.

5.6 Processing Ungrammatical Inputs

It was hypothesised that simple/weighted matching would outperform Jaro (forward-stemmed) in processing ungrammatical inputs. The proportion of ungrammatical inputs[8] (less than 10%) was small. Errors of grammar (usually number/person agreement failures) were either very local ("a courses") or longer distance:

when does the university starts?

For the keyword-based systems, there was no notion of agreement: each keyword was independent and so number/person agreement could not be enforced, even if desirable. Agreement is explicit in Jaro-based methods because, assuming stereotypical sentences will be well-formed, there would be no complete match between the users' inputs and the stereotypical sentences. However, for longer distance ungrammaticality to be possible, there has to be a relatively long input and so the Jaro score would be less reduced than it would be with very local ungrammaticality in short inputs.

Adding stemming worked against any effects of agreement. By reducing word forms to their stems, morpho-syntactic information was removed and so it played no part in the matching process. This had the effect of improving matching.

6 Conclusions and Further Work

A best correctness rate of 78.57% is not high enough for an effective system. The Jaro (forward-stemmed) method offers the possibility of further improvement because it is scalable, thus allowing more stereotypical sentences to be used. In particular, it performed well on clearly related sentences and less well on longer sentences not closely represented in the stereotypical sentence store. The target application of a postgraduate application enquiry

[8] Not to be confused with abbreviated input e.g. "registration deadline?"

system would be used by native and non-native English speakers. There is no evidence that ungrammatical queries led to serious deterioration of performance of the Jaro (forward-stemmed) string similarity algorithm.

The problem with the use of the Jaro (forward-stemmed) method is acquiring stereotypical sentences. To this end it is proposed that, in the target application, users be allowed to decide if the system has answered their question or not. If their response in positive, their query could be added to the store of stereotypical sentences. Thus the system would, in a limited way, be capable of learning. In this way, it would have a more limited learning capability than those systems (e.g. Jabberwacky [4]) that seek knowledge from the user.

There is further work that could explore the capabilities of keyword-based searches. First, a limited dictionary of synonyms could be used to normalise queries. So, instances of "MSc", "master", "masters", "MSc in", "MSc of", etc. could be normalised to one chosen form. This would reduce the number of keywords to be stored and make it easier to keep keywords and their weights consistent with other keywords and weights. Second, there is a possibility of returning to ELIZA-style templates for matching where the system has a number of patterns of the form:

```
what is the deadline for KEYWORD(S)?
```

However, this could overcomplicate the system, leading to poorer performance. It would require a more sophisticated keyword system, perhaps of a predicate/argument structure (e.g. deadline(option_choice)). This in turn would be more difficult to use with abbreviated ("Google-like") queries.

Acknowledgements We wish to thank those anonymous applicants who tested the systems for us.

References

1. ALICE Artificial Intelligence Foundation. *AIML: Artificial Intelligence Markup Language*. Available at: http://www.alicebot.org/aiml.html. [Accessed 9 March 2012].
2. William W. Cohen and Pradeep Ravikumar and Stephen E. Fienberg. *A comparison of string distance metrics for name-matching tasks*. In: Workshop on Information Integration on the Web (IIWeb-03). IJCAI, 2003. pp. 73–78.
3. Matthew A. Jaro. *Advances in record linkage methodology as applied to the 1985 census of Tampa Florida*. Journal of the American Statistical Society 84(406), 1989, pp. 414-–20.
4. Jiyou Jia. *CSIEC: a computer-assisted English learning chatbot based on textual knowledge and reasoning*. Knowledge-based Systems 22(4), 2009, pp. 249–255.
5. Nikolaos Polatidis. *Chatbot for admissions*. Unpublished MSc dissertation. Edgbaston: School of Computer Science, University of Birmingham, 2011.
6. M.F.Porter. *An algorithm for suffix stripping*. Program 14 (3), 1980, pp. 130-–137.
7. Rebecca Satterthwaite. *Prolog-Java chatbot for postgraduate admissions in the School of Computer Science*. Unpublished MSc dissertation. Edgbaston: School of Computer Science, University of Birmingham, 2010.
8. Joseph Weizenbaum. *Computer power and human reason: from judgement to calculation*. Penguin, Harmondsworth, 1984.
9. William E. Winkler. *Overview of record linkage and current research directions*. Washington, D.C.: Statistical Research Division, U.S. Census Bureau, 2006. (Research report series: Statistics 2006-2).
10. William E. Winkler. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*. Proceedings of the Section on Survey Research Methods. American Statistical Association, Washington, D.C., 1990. pp. 354—359.

Predicting Market Direction from Direct Speech by Business Leaders

Brett M. Drury[1] and José João Almeida[2]

1 **LIAAD-INESC**
Rua de Ceuta 188,6, Portugal
Brett.Drury@gmail.com
2 **University of Minho,**
Braga, Portugal
jj@di.uminho.pt

Abstract

Direct quotations from business leaders can communicate to the wider public the latent state of their organization as well as the beliefs of the organization's leaders. Candid quotes from business leaders can have dramatic effects upon the share price of their organization. For example, Gerald Ratner in 1991 stated that his company's products were *crap* and consequently his company (Ratners) lost in excess of 500 million pounds in market value. Information in quotes from business leaders can be used to make an estimation of the organization's immediate future financial prospects and therefore can form part of a trading strategy. This paper describes a contextual classification strategy to label direct quotes from business leaders contained in news stories. The quotes are labelled as either: 1. positive, 2. negative or 3. neutral. A trading strategy aggregates the quote classifications to issue a buy, sell or hold instruction. The quote based trading strategy is evaluated against trading strategies which are based upon whole news story classification on the NASDAQ market index. The evaluation shows a clear advantage for the quote classification strategy over the competing strategies.

1998 ACM Subject Classification I.2.1 Application and Expert Systems

Keywords and phrases Sentiment, Direct Speech, Trading, Business, Markets

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.163

1 Introduction

Direct quotations from business leaders can communicate to the wider public the latent state of their organization as well as the beliefs of its leaders. This information can be used to make an estimation of the organization's immediate future financial prospects. An infamous example of the effect of candid quotes on a company's market value was a speech by Gerald Ratner at the Institute of Directors (IOD) in 1992. He stated: *We also do cut-glass sherry decanters complete with six glasses on a silver-plated tray that your butler can serve you drinks on, all for £4.95. People say, How can you sell this for such a low price?, I say, because it's total crap.* and *We sell a pair of earrings for under £1,which was cheaper than a prawn sandwich from M&S, but probably wouldn't last as last as long.*. The Ratner Group company share price fell rapidly and as a direct result of the speech at the IOD the company lost 500 million pounds in market value. Gerald Ratner was forced to resign and the company changed its name to the Signet Group to disassociate itself from Ratner's speech [16]. This extreme example illustrates the communication of the latent negative attitudes of the company leadership towards its products and the associated market reaction.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 163–172

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

The identification of *actionable information* in quotes from business leaders is a non-trivial task because quotes can contain obscure linguistic features as well as a rich and varied lexicon [5]. Traditional classification techniques are not suitable for this task [8]. This paper proposes that a *contextual classification technique* [8] for labelling quotes from business leaders can be the basis of a successful stock trading strategy. This *quote based trading strategy* was evaluated on the NASDAQ market index against trading strategies which used whole news classification strategies to issue trading instructions. The evaluation used trading accuracy and points difference to gauge each strategy. The results show a clear advantage for using direct quotes as a basis of trading strategy when compared to the competing strategies. The remainder of the paper will discuss: 1. related work, 2. classification strategies, 3. experiments 4. conclusions.

1.1 Related Work

The use of quotes from business leaders as a basis of a trading strategy seems to be a unique research problem as the literature review failed to identify papers which used quotes for trading. There were a small number of papers which attempted to classify quotes into pre-assigned categories. Balahur et al [1] classified quotes from politicians by using general sentiment dictionaries. The value from the dictionaries for each sentiment word in a quote were summed together and a label applied to the quote. A positive quote has a score of greater than 0, a negative quote would have a score of less than 0 and a neutral quote would have a score of 0.

The research literature was richer when examined for strategies which used general news information to predict the direction of a stock market index. The research literature revealed two dominate strategies for classifying news stories: 1. dictionary based and 2. market alignment. Dictionary approaches use a pre-compiled dictionary which contain n-grams and an associated trading action or classification. The information in the dictionary is used to score information in a group of news stories on a single day. This score is used to issue a: 1. buy, 2. sell or 3. hold trading instruction. An example of a dictionary approach is a system developed by Wuthrich et al [19]. The dictionary which was constructed by Wuthrich contained tuples of words separated with an "AND" Boolean instruction, for example BOND and STRONG. The dictionary was used to analyse news stories published when the financial markets were closed. The number of stories in each category would be counted and sell or buy instruction would be generated for the index. They claim a 21% advantage over and above a trader who would trade by guessing based on an uniform distribution [19]. Another example of a trading system based upon a dictionary based approach was the News Categorization and Trading System *(NewsCATS)* [15]. NewsCATS analysed company press releases, and attempted to predict the company's share price upon information contained in the press release. The dictionary was not published, but the authors state that the dictionary was created by hand. The construction methodology was also not published. The function of the dictionary was to assist NewsCATS to categorise press releases into pre-designated categories. These categories were designed to indicate the influence (positive or negative) of the press release upon the share price. The system's authors claim that they significantly out perform a trader who buys on a random basis after press releases [15].

The market alignment strategies are used to "bootstrap" training data for a classifier. A news story would be labelled: 1. "negative" if it was published at the same time as a fall in a market's value, 2. "positive" if it was published at the same time as a rise in a market's value or 3. "neutral" it was published at the same time as a marginal change in a market's value. The classifier would then classify unlabelled news stories. A trading action

would be issued based upon the number of: positive, neutral and negative classification in a given time period. The AEnalyst [13] used market trends to construct language models to recommend news which would effect a share price. Drury et al [9] used market alignment and self-training to construct models to classify news stories into pre-defined categories. This information is used as a basis of a trading strategy.

2 Classification Strategies

The experiments for this compared a trading strategy which used information from quotes from business leaders against trading strategies which used news story classification methods. News story classification strategies classified a either a news story's: 1. headline or 2. story text. There were three competing news story classification techniques: 1. rule selected, 2. market alignment and 3. combination of rule selected and market alignment.

2.1 Quote Classification Strategy

The *quote classification method* for the trading strategy was the one proposed by Drury et al [8]. They proposed a two-step quote classification strategy. The strategy assumed that there was two types of speakers: 1. biased and 2. unbiased.

The *biased group* of speakers were assumed to be *unobjective* when talking about their organization because their job role forced them to be less than truthful [18]. A quote from a member of the biased group can not be taken on "face value", therefore classical sentiment analysis techniques will fail because quotes from this group were overwhelmingly "positive" [8]. Members of this group cannot always behave in this manner because on occasions they are legally compelled to present information about their company in an objective manner, for example profit announcements or warnings. Business leaders who actively lie or mislead about this information commit criminal offences which can lead to long terms of imprisonment [10].

The *unbiased group* consisted of speakers who had job roles where they were compelled to be objective, for example, researchers, traders or analysts. They are compelled to be objective because their job to disseminate objective information to clients who pay fees for this advice. The clients use this information to trade. Biased or untruthful information would impair the client's ability to trade.

The strategy grouped together quotes from members of the biased and unbiased groups, and applied a separate classification strategy to quotes from each group. The classification strategy for quotes from members of the "biased" strategy uses a market alignment strategy to label training data. The labelling procedure labelled quotes into two categories: 1. market moving and 2. non market moving. A quote is labelled as *market moving* if: 1. the share price of the speaker's employer moves significantly or 2. the trading volume of the share of the speaker's employer increases significantly. If the quote did not provoke a price change or trading volume increase it was labelled as "non market moving". The labelling process was manual and therefore time consuming. It was difficult to locate market moving quotes because there were 100 non market moving quotes to 1 market moving quote. A cluster processing was used to increase the number labelled quotes. An initial "seed set" of labelled quotes was clustered with unlabelled quotes. Clusters which contained a single class of labelled quotes had their label propagated to the unlabelled quotes in the cluster. The market moving quotes were labelled into two further categories: 1 negative and 2. positive. The labelling process was a manual process which aligned quotes with share price movements of the speaker's employer. A quote was labelled "positive" if the share price rose whereas a quote would be labelled as "negative" if the share price fell. There were two classifiers

induced. One model classified unlabelled quotes into "market moving" and "non market moving". The second model classified quotes labelled as "non market moving" into "positive" or "negative" categories.

Labelled data for the unbiased group was initially manually selected into: positive and negative categories. The quotes were selected by the affective information in the quote, i.e quotes which contained negative words were labelled as "negative" and quotes which contained positive words were labelled as "positive". Adjectives, nouns, verbs and adverbs were extracted from the initially labelled data and placed in a dictionary with a label. These terms were expanded with: synonyms and antonyms from Wordnet [11, 14]. The dictionary used linguistic rules to label unlabelled quotes. The linguistic rules labelled a quote "negative" if it contained at least three "negative terms" or positive if it contained at least three "positive terms". The linguistic rules were based upon Weibe's high confidence subjectivity classifier [17]. The newly labelled data was used to induce a classifier. Quotes from the unbiased group into "positive" or "negative" categories. A trading signal was issued by summing the quotes from both categories which were labelled as either "positive" or "negative".

2.2 Rule Selected Data Strategy

The rule selected data was a strategy which relied upon manually constructed linguistic rules to select and label data which was used to train a classifier. The economic literature suggests that events or sentiment reported in the mass-media can provoke a reaction in a market index or share price [4, 12]. The linguistic rules identified and scored event or sentiment phrases in news text. The linguistic rules relied upon dictionaries learnt from the news text. There were dictionaries for: economic actors, verbs, adverbs, adjectives and economic actor properties. Economic actors were either: companies, organizations, market indexes or business leaders. Economic actor properties were nouns associated with economic actors, for example: profits, costs, unemployment, etc. The verb dictionary contained "scored verbs" which linked an economic actor with economic actor properties, for example: "Microsoft (economic actor) profits (economic actor property) *rose* (verb) ". The sentiment dictionary contained "scored adjectives" which linked an economic actor with economic actor properties.

The linguistic rules modelled event or sentiment phrases as a triple: 1. economic actor, 2. verb and 3. economic actor property for event phrases and 2. 1. economic actor, 2. sentiment and 3. economic actor property for sentiment phrases. Event phrases were assigned a score based on the verb score. The event score was the sum of the verb scores in the event phrase. The verb score could be inverted if an economic actor property in the phrase was labelled as verb modifier. For example, the verb "rise" had a score of "1", however if "unemployment" was the economic actor property then "rise" was assigned a score of "-1". Sentiment phrases were scored with the *AVAC algorithm* [2]. The adverbs in the adverbs were used to modify the adjectives scores in the sentiment phrase. The adverbs can: 1. increase sentiment scores, 2. decrease sentiment scores and 3. invert sentiment scores. The rule construction strategy is described in full by Drury and Almeida [6].

The linguistic rules labelled news stories by assigning a score to its headline. A score was calculated for a headline by combining the scores returned by the event and sentiment linguistic rules. A news story was labelled: 1. *negative* if it was assigned a score of less than 0, 2. *positive* if it was assigned a score of greater than 0 and 3. *neutral* if the headline did not contain an entry from the verb or adjective dictionary. The classifier was induced by balancing the number of labelled documents from each category.

2.3 Market Aligned Data Strategy

The market aligned data strategy relied upon rise and falls in the value of a market index to label news stories. The strategy labelled news stories as: 1. "positive" if published on the same day as an increase in the value of a market index, 2. "negative" if published on the same day as a decrease in the value of a market index and "neutral" if published on the same day as a marginal change in the value of a market index. Sharp single day movements were chosen in preference to trends because financial markets can move on non news information and consequently trends can be illusionary [3] whereas single day market movements are more likely to occur is there is an underlying cause. Examples of single movements and their cause are in Table 1.

Table 1 Large Single Day Fluctuations in the FTSE.

Date	FTSE (+/-)	Reason
8th August 2011	-3.39%	Falls in US and Asian Markets
10th/12th Sept 2011	-2.73%	Terrorist Attacks
7th Sept 2008	-1.93%	Financial Crisis

The market alignment strategy evaluated the effect of *market index thresholds* in the labelling process. A market index threshold is the minimum market movement before a news story published on the same day can be labelled: 1. negative, 2. positive or 3. neutral. For example, market threshold of 1% and -1% would infer that stories published on the same day: 1. as an increase in value of a market index more than 1% would be labelled as "positive",2. as a decrease in value of a market index more than 1% would be labelled as "negative" and 3. an increase or decrease in value of a market index between 1% to -1% would be labelled as neutral. The tested market index thresholds were 1% to 9% for "positive" stories and -1% to -9% for "negative" stories.

2.4 Rules with Market Alignment Strategy

This strategy used the rule selected and market alignment strategies to label data. A story would be assigned a label if both strategies agreed on the same label. For example, a story would be labelled "negative" if both the rule selected and the market alignment strategy applied a negative label. The effect of *market index thresholds* was evaluated. The tested market index thresholds were 1% to 9% for "positive" stories and -1% to -9% for "negative" stories.

3 Experiments

The experiments were designed to evaluate the ability of each strategy to estimate the direction of the opening price of the NASDAQ market index when compared to the previous days closing price. This period was chosen because the NASDAQ was closed and the market could not react to the information contained in a quote or news story until the market opens the next day. The competing strategies had access to news stories published the day before until the opening of the market the next day.

The strategiesused a corpus of news stories which were published between October 2008 - April 2011. Headlines, story text, story published date and direct speech (quotations) were extracted from the news stories. The quotations are publicly available in *Minho Quotation*

Resource[1] [5]. The experiments used 300 randomly selected days from which news stories or quotations were used as labelled data. A 100 randomly selected days from from which news stories or quotations were used as testing data. The testing data was drawn from a period of time which was after the latest training day. This selection process was repeated 5 times. The competing strategies had access to the same sets of selected days.

The trading strategies classified a news story or quotation. If a news story or quotation was classified *negative* then it was assigned a score of -1 whereas a *positive* news story or quotation was assigned a score of 1. A neutral story or quotation was assigned a score of 0. A trading action was generated by summing the total news story scores for each day. A *buy* action was issued if the score for a specific day was greater than 0 whereas a *sell* action was issued if the score for a specific day was less than 0. A hold action was issued if the score for a day was 0. The effect of trading thresholds was tested. A trading threshold is the minimum day's score before a trading action could be issued. For example a threshold of 10 ensured that a day's score of 10 or more was required before a buy action was issued, and -10 or less before a sell action was issued. The range tested was from 0 to 90. The experiments also tested the effect of classifier confidence on the results of each strategy. The classifier confidence refers to the minimum confidence required before a classification of a news story or quotation would be accepted by a trading strategy. The tested range was from 0.5 to 1.0. The rules with market alignment and market alignment strategies also evaluated the *market index thresholds* for labelling data. Language models were used as a classifier in all of the experiments.

3.1 Points Difference and Trading Accuracy Evaluation

A measure for evaluation the competing strategies was the mean points difference gained or lost during the trading activities for each trading period. Table 2 holds the average points difference for all tested configurations. A successful strategy produces a positive points difference whereas an unsuccessful strategy produces a negative points difference. The quote strategy was clearly superior when all configurations were considered.

Table 2 Average Trading Profits (All Configurations).

Strategy	Points Difference	Accuracy
Rule Trained (news text)	-0.33±34.41	0.49±0.01
Rule Trained (headlines)	-18.74±54.99	0.47±0.03
Market Alignment (news text)	11.95±83.55	0.44±0.17
Market Alignment (headlines)	11.48±90.32	0.46±0.16
Rules + Market Alignment (news text)	0.57±66.59	0.45±0.16
Rules + Market Alignment (headlines)	-10.89±90.23	0.45±0.15
Quotes	86.98 ±67.08	0.55±0.03

The market based strategies used gains and losses from the NASDAQ to label quotes as either: positive, negative or neutral. The variation of the values used in this labelling strategy had an effect on the mean points difference and trading accuracy. Table3 holds the results for the market based strategies optimal configuration. The quotes strategy returned the highest points differences, but this difference was within the standard deviation of the competing strategies.

[1] Available at: `http://goo.gl/U6quN`

Table 3 Average (Optimal Market Configurations).

Strategy	Positive	Negative	Returns	Accuracy
Align. (text)	4	-3	78.47 ±53.18	0.52 ±0.02
Align. (headlines)	1	-8	75.63 ±68.77	0.55 ±0.05
Rules + Align. (text)	5	-3	51.97 ±49.16	0.53 ±0.02
Rules + Align. (headlines)	4	-3	32.23 ±120.10	0.48 ±0.07
Quotes	NA	NA	86.98 ±67.08	0.54 ±0.03
Rules (text)	NA	NA	-0.33 ±108.24	0.49 ±0.05
Rules (headlines)	NA	NA	-18.74 ±54.96	0.46 ±0.03

The final evaluation was to discover the optimal configurations of decision boundary and classification confidence for the quote and rules strategies, and market values, decision boundary and classification confidence for the market based strategies. Classifier confidence is the minimum confidence value a learner can return for a document before it is used in a trading decision. For example, a trading strategy which uses a classifier confidence of 0.90 will discard all documents classified with a confidence of 0.895 or lower. The decision boundary is the difference between the number of classifications before a trading decision is made. For example, a decision boundary of 90 ensures that the majority class must contain 90 more documents than either of the two minority classes. Table 4 demonstrates the trading accuracies and points differences for the optimal configurations for each strategy. The quote strategy returned the highest points difference and trading accuracy.

Table 4 Trading Returns (Optimal Configurations Confidence / Decision Boundary).

Strategy	Boundary	Confidence	Returns	Accuracy
Align. (text)	0	0.5 - 0.9	110.44 ±52.32	0.53 ±0.01
Align. (headlines)	0	0.5 - 0.9	83.44 ±77.86	0.55 ±0.04
Rules+ Align. (text)	0	0.5	109.86 ±63.24	0.53 ±0.02
Rules+ Align. (headlines)	90	0.5	83.96 ±112.42	0.51 ±0.05
Quotes	0	0.6	233.80 ±60.27	0.58 ±0.04
Rules (headline)	60	0.6	40.76 ±16.20	0.48 ±0.03
Rules (text)	90	1	56.64 ±96.39	0.50 ±0.06

4 Influence of Variables on Quote Strategy Performance

The experimental section described experiments which estimated points differences and trading accuracy for each strategy and for various configurations of the strategy. This section will discuss the configurations of the quote strategy. Visual evidence of the influence of classifier confidence and decision boundary are presented in Figures 1 and 2. The visual evidence suggests that the lower the decision boundary the higher the trading accuracy and points differences. The influence of classifier confidence is unclear. Pearson values were calculated for each of the variables with either points difference or trading accuracy. Pearson values are a value which represents the strength of a correlation between variables. Table5 holds the Pearson scores for each variable. The Person values confirm that there was an identifiable relationship between: 1. decision boundary and trading accuracy and 2. decision

boundary and points difference and no identifiable relationship between classifier confidence and trading accuracy or points difference.

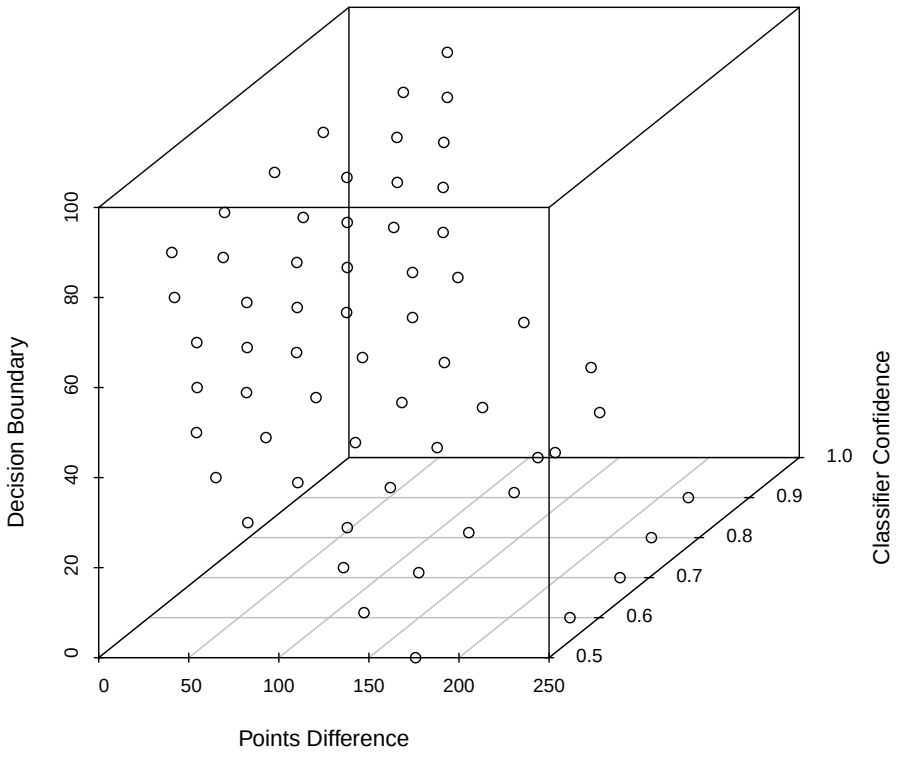


Figure 1 Point difference against classifier confidence and decision boundary.

Table 5 Correlation between variables and trading accuracy and points differences.

Variable 1	Variable 2	Pearson Value
Accuracy	Decision Boundary	-0.31
Trading Profits	Decision Boundary	-0.62
Accuracy	Confidence	0.00
Trading Profits	Confidence	-0.02

The negative correlation between decision boundary and trading accuracy and points difference may indicate that quotations are not equal. A consensus among speakers may not be required because quotes from single *important* speakers may outweigh numerous quotations for less influential people. For example, a single quote from Warren Buffett[2]

[2] Examples of his investment record can be found at `http://goo.gl/AETVm`

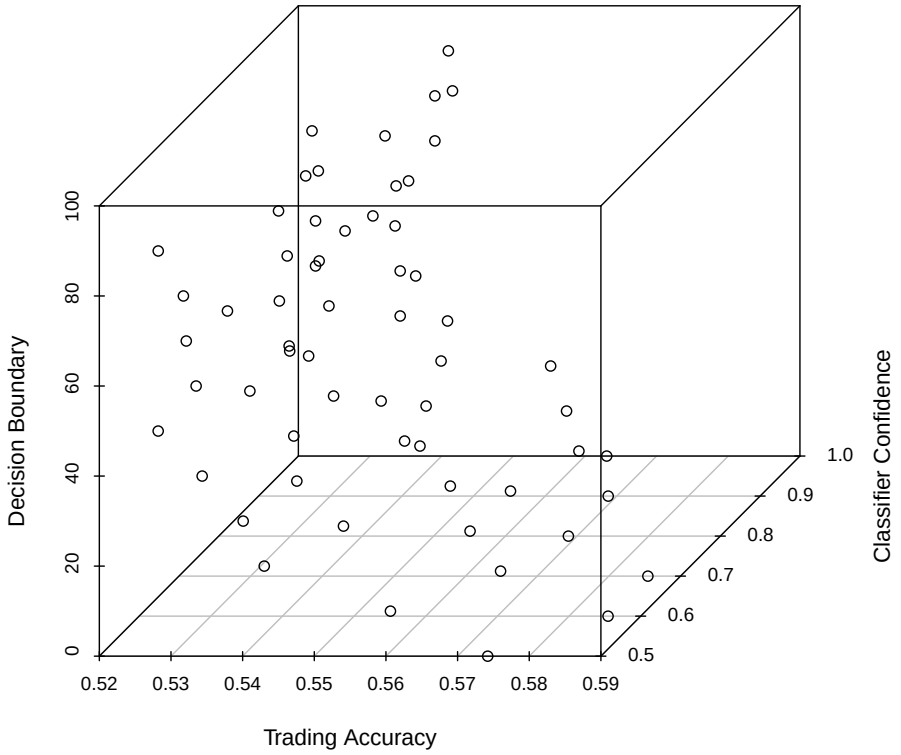


Figure 2 Trading Accuracy against classifier confidence and decision boundary.

can have a strong influence on stock prices, whereas numerous quotes from less well known analysts may have negligible or no effect on stock prices.

5 Conclusion

This paper presented a novel trading strategy which used quotations to estimate the direction of a market index (NASDAQ). It was superior to the competing strategies as it returned the highest points differences in all of the evaluation measures. It returned the highest trading accuracy for two of the three evaluations.

The work presented in this paper did not consider the influence of the speaker and market performance. A strategy which considers the influence of a speaker may achieve better results. It may be possible to identify and score influential speakers with a domain ontology [7]. In addition the results gained by the strategy will have to be tested against the possibility that these results could have been gained by chance. The work presented in this paper demonstrates that trading with *quotations* gains results comparable with traditional rule based and market alignment news classification strategies.

References

1 Alexandra Balahur, Ralf Steinberger, Mijail Kabadjov, Vanni Zavarella, Erik van der Goot, Matina Halkia, Bruno Pouliquen, and Jenya Belyaeva. Sentiment analysis in the news. In Nicoletta Calzolari, editor, *Proceedings of the Seventh conference on International Language Resources and Evaluation*, Valletta, Malta, may 2010.

2 Farah Benamara, Carmine Cesarano, Antonio Picariello, Diego Reforgiato, and V. S. Subrahmanian. Sentiment analysis: Adjectives and adverbs are better than adjectives alone. In *Proceedings of the International Conference on Weblogs and Social Media (ICWSM)*, 2007.

3 Wesley Chan. Stock price reaction to news and no-news. drift and reversal after headlines. *Journal of Financial Economics*, 70(2):223–260, 203.

4 Werner F. M. De Bondt and Thaler R. Does the stock market overreact? *The Journal of Finance*, 40(3):793–805, 1985.

5 Brett Drury and J.J. Almeida. The minho quotation resource. In *LREC*, 2012.

6 Brett Drury and José João Almeida. Identification of fine grained feature based event and sentiment phrases from business news stories. In *WIMS*, page 27, 2011.

7 Brett Drury, Jose Joao Almeida, and M.H. Moreira Morais. Construction and maintenance of a fuzzy temporal ontology from news stories. *International Journal of Metadata, Semantics and Ontologies*, 2012.

8 Brett Drury, Gaël Dias, and Luís Torgo. A contextual classification strategy for polarity analysis of direct quotations from financial news. In *RANLP*, pages 434–440, 2011.

9 Brett Drury, Luís Torgo, and José João Almeida. Classifying news stories with a constrained learning strategy to estimate the direction of a market index. *IJCSA*, 9(1):1–22, 2012.

10 Peter Elkind and Bethany McLean. *The Smartest Guys in the Room: The Amazing Rise and Scandalous Fall of Enron*. Penguin, 2004.

11 C Fellbaum. *WordNet: An Electronic Lexical Database*. The MIT Press, 1998.

12 Peter Ager Hafez. Construction of market sentiment indices using news sentiment. Technical report, Ravenpack, 2009.

13 Victor Lavrenko, Matt Schmill, Dawn Lawrie, Paul Ogilvie, David Jensen, and James Allan. Language models for financial news recommendation. In *In Proceedings of the Ninth International Conference on Information and Knowledge Management*, pages 389–396. ACM Press, 2000.

14 Bing Liu. *Web Data Mining: chapter(Opinion Mining)*. Springer, 2007.

15 Marc-Andre Mittermayer and Gerhard F. Knolmayer. Newscats: A news categorization and trading system. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pages 1002–1007. IEEE Computer Society, 2006.

16 Gerald Ratner. *The Rise and Fall... and Rise Again*. Wiley, J, 2007.

17 E. Riloff and J. Weibe. Learning extraction patterns for subjective expressions. In *In Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, pages 105–112, 2003.

18 James Spindler. Why shareholders want their ceos to lie more after dura pharmaceuticals, 2006.

19 Wuthrich, Cho, and Leung. Daily prediction of major stock indices from textual www data. In *4th ACM SIGKDD Int. Conference on Knowledge Discovery and Data Mining*, pages 364–368, 1998.

Part III

Short Communications

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz

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

Learning Spaces for Knowledge Generation

Nuno Oliveira[1], Maria João Varanda Pereira[2], Alda Lopes Gancarski[3], and Pedro Rangel Henriques[1]

1 **Universidade do Minho**
Braga, Portugal
{nunooliveira,prh}@di.uminho.pt
2 **Polytechnic Institute of Bragança**
Bragança, Portugal
mjp@ipb.pt
3 **Institut Télécom, Télécom Sudparis CNRS UMR Samovar**
Évry, France
alda.gancarski@it-sudparis.eu

Abstract

As the Internet is becoming the main point for information access, Libraries, Museums and similar Institutions are preserving their collections as digital object repositories. In that way, the important information associated with digital objects may be delivered as Internet content over portals equipped with modern interfaces and navigation features. This enables the virtualization of real information exhibition spaces rising new learning paradigms.

Geny is a project aiming at defining domain-specific languages and developing tools to generate web-based learning spaces from existent digital object repositories and associated semantic. The motto for Geny is "Generating learning spaces to generate knowledge". Our objective within this project is to use (*i*) ontologies—one to give semantics to the digital object repository and another to describe the information to exhibit—and (*ii*) special languages to define the exhibition space, to enable the automatic construction of the learning space supported by a web browser.

This paper presents the proposal of the Geny project along with a review of the state of the art concerning learning spaces and their virtualization. Geny is, currently, under appreciation by Fundação para a Ciência e a Tecnologia (FCT), the main Portuguese scientific funding institution.

1998 ACM Subject Classification H.3 Information Storage and Retrieval

Keywords and phrases Learning Spaces, Knowledge Acquisition, Digital Object Repositories, Ontology-based semantic Web-pages Generation

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.175

1 Introduction

Learning spaces are generally associated to classrooms within academia [4, 7]. They are commonly seen as physical places where groups of persons (typically students) discuss a theme and there exist someone (typically a teacher) that leads the discussion by organising the ideas and sums them up, creating knowledge. However, it is a fact that a great percentage of what a human knows is not learnt within a classroom. Discussions with other persons during a simple walk or on a break for a coffee and visits to museums or libraries are also means to learn. Therefore, any physical space where there is knowledge to be shared may be regarded as a learning space.

The advances in the internet and associated technologies made possible the port of physical learning spaces into virtualised versions. eLearning was the term coined to these spaces,

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 175–184

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

but again, intrinsically related with academia. However, with internet, the information is largely spread and accessible anywhere to anyone. Libraries, Museums and similar Institutions are preserving their collections as digital object repositories to make their associated information available as Internet content. Portals equipped with modern interfaces and navigation features enable the virtualisation of real information exhibition spaces and give rise to new learning paradigms. Summing up, these resources across the internet allow for a large range of persons—not only students—to generate knowledge; and eLearning shall not be applied in this generalised view to avoid misconceptions.

In this context, we regard a learning space as a site where a certain piece of information is exhibited in a way that the visitor learns with it. A web-based (virtual) learning space is similar to a traditional one but it is implemented as a web site where a collection of information pieces is exhibited so that the site visitors can learn with it.

To acquire new knowledge, a visitor needs to have access to organised data, but he also needs to be enrolled in the learning process. This is precisely the purpose of web-based learning spaces we intend to build automatically from their formal descriptions.

We propose Geny as a project aiming at defining domain-specific languages and developing tools to generate web-based learning spaces from existent digital object repositories and associated semantic. Geny's motto is "Generating learning spaces to generate knowledge"! Our objective within this project is to use (i) ontologies—one to give semantics to the digital object repository and another to describe the information to exhibit—and (ii) special languages to define the exhibition space, to enable the automatic construction/generation of the learning space supported by a web browser. With the obtained learning space we intend to give the visitor a more active role in the learning process. Quoting Saint Exupèry: "Each one that passes in our life takes a little from ourselves, but he also leaves something of himself", we argue that visitors should be participative actors by completing each learning space with their personal knowledge and assets for that theme. Moreover, learning spaces and the associated technological gadgets and widgets shall (*i*) stimulate the visitor for knowledge acquisition; (*ii*) trigger the visitor's imagination and his will to contribute and (*iii*) capture the visitors' feedback.

We dream with learning spaces where the global story (the message to pass there) is made up from smaller stories; a democratic location where not only the story-teller has permission to talk.

Outline. In this paper we briefly discuss a project proposal and focus on the related work/state of the art. In this context, section 2 presents the state of the art with respect to the Geny project. We discuss several projects and compare their features with those we propose. Then, in section 3 we briefly present our proposal for Geny, by showing the general architecture and workflow with the intended features. Finally, in section 4 we close the paper, by providing a summary of the discussion undertaken in the other sections.

Notice. Geny is, currently, being reviewed by Fundação para a Ciência e a Tecnologia (FCT), a scientific funding institution. This article is a summary of the Geny project proposal, where we deposit the main ideas to improve the current state of the art on virtual learning spaces. No further outcomes are available.

2 State of the Art

Several projects have been proposed and accomplished which addressed the analysis and creation of virtual spaces where learning is enabled for a broader range of persons rather

than just for students as happen with eLearning. The following paragraphs describe some of these projects and relate them with Geny.

The Spaces for Knowledge Generation project [11][1] develops a broad study on learning spaces for students within universities. Learning spaces, in this perspective, are seen as physical places where students generate knowledge. The main idea of the project is to provide principles for the creation of learning spaces capable of being reconfigurable according to the students needs. These principles include comfort, aesthetics, flow, equity, blending, affordances and repurposing. The project members stress the importance of a social interaction by blending students, teachers and technology, for a successful knowledge generation within learning spaces.

This project diverges in a large part from Geny. Its focus is set on both physical learning spaces within universities, where students may collaboratively and socially generate knowledge, and in the principles used to improve these spaces. The aimed public is limited. However, the ideas discussed in the project's produced book [14] for physical learning spaces may clearly be taken into account, within Geny, for generating virtual learning spaces, which are thought to serve a broader range of potential users.

Recent emergence of virtual imitation of the real world, e. g, with Second Life®, makes possible a new way of providing virtual learning spaces[3, 10, 9]. In these cases, information is modeled in 3D and delivered as part of virtual worlds where users (by means of avatars) may learn new things by freely (with no cost or risk) move around such artifacts. In [9], authors explore the advantages and benefits of virtual worlds as immersive and social learning spaces. Their point is to allow teachers (or teaching institutions, in general) for transferring real world learnt lessons to the virtual cost- and risk-free worlds (and vice-versa). This enables students to acquire knowledge about certain aspects difficult to explore in other settings.

The generation of learning spaces within these virtual worlds is mainly a manual task. Although a semantic coherence may be defined in the availability of the informational resources, users may not navigate through them following conceptual and semantic approaches. Moreover, users may be easily distracted by other objectives (e.g. online social acquaintance of other users) rather than focusing on learning the provided thematic. More discrete approaches (like that of Geny) make more sense for capturing the visitor's attention to the single objective of learning. These approaches for 3D learning spaces may also be limited in two aspects: first the aimed public may be reduced due to the used technology—clearly, it is more suitable for younger users—and second, it reduces the possibility of collaborative enrichment of the knowledge embodied in such learning spaces. Both aspects are intended to be treated in Geny.

The Domus Naturae project aims at developing a web-based virtual museum application taking advantage of tools for managing heterogeneous and structured knowledge. Such knowledge is represented on ontologies, using the emerging Semantic Web standard technologies. In [6], the authors propose the use of a ontology for constraining, expressing and analyzing the meaning of concepts and relations in the domain of knowledge subjacent to the project. The authors stress out that using such ontology upon a collection of digital objects, the user may navigate through the virtual exhibition taking advantage of queries closer to the domain of discourse.

This project is similar to Geny, in the sense that ontologies are superimposed over a repository of digital objects allowing for navigation upon such objects. However, in the context of Geny, a ontology is to be used also as a means to make the automatic generation of

[1] http://www.skgproject.com/

a learning space closer to the knowledge the visitor intends to get. Moreover, the combination of Social and Semantic Web in Geny is in order to provide a new and sustainable experience on learning about a theme and completing it with visitor's knowledge.

PATHS: Personalised Access To cultural Heritage Spaces [15][2] is a recently approved FP7 with strong similarities to the project we are proposing. The PATHS project would rely on the European Digital Library (Europeana) contents to create a system that will act as an interactive and personalized tour guide through the Europeana's collections. The idea behind this system is to create virtual thematic paths through the objects of all available collections in Europeana. The paths may be either pre-defined or user-defined. In this context, users have a very important role in this system as they may create their own path through the digital objects. In fact, this is the authors' claim of innovation in the user-driven information access experience.

Little information is available at this moment for the PATHS project. However, it is easy to trace similarities to our own. From the description produced at this moment, a path (considered the perspective to learning spaces) is a user creation by selecting a set of nodes—pointing to repository objects—and annotating and linking them to create a coherent narrative. Although our approach is similar, we do not intend to create paths based on a selection of objects (which may be semantically unrelated). Rather, our learning spaces are defined taking into account the semantic relations between the repository objects, enabling a single learning space to involve all the preserved objects. Moreover, visitors will have the ability to perfect the learning spaces, the ontological layer upon the repository and the repository itself based on their own previous knowledge and private assets.

In [8], MuseumFinland is presented as a semantic portal for publishing heterogeneous museum collections based on the Semantic Web. The system is based in a set of ontologies, showing that it is possible to make semantically interoperable collections and provide visitors with intelligent content-based search and browsing services to the global collection base. Some technologies like XML, RDF, OntoViews and Prolog (logic predicates for reasoning over static information) are used. Since MuseumFinland is based on a central repository and uses ontologies to navigate through meta-information, it is possible to mix information from different resources (including different museums). Moreover, the system also allows for showing implicit relations between contents and semantic browsing. It also enables the create of end-user's viewpoints. The collection items are represented as web pages, linked with each other through semantic associations. The system layout is based on different views and facets.

Geny follows a similar approach. However we add value in two fronts. First we intend to allow for feeding the knowledge base with information collected from users and their own knowledge on the theme; secondly, we intend to enable the automatic generation of virtual learning spaces, which are not focused on museum assets.

The Art Project[3], powered by Google, is a website that allows for an interactive and completely virtual exploration of some well known museums around the world. This experience provides access to art collections through a rich and intuitive interface. Besides browsing art collections, the user is invited to actually explore the museum, navigating in a 3D world that mirrors the physical building, following the same approach as in Google Street View.

This project has many social advantages. It allows visitors for accessing art and related

[2] http://www.path-project.eu
[3] http://www.googleartproject.com/

information, without the need of having to actually going there, which can be very expensive and not affordable in many cases. Also it enables the user to access information at his own time and need, since the information is always available and users are more prone to get interested. With beautiful and appealing interfaces, this project is sure to attract the attention of people that would normally not spend their time visiting museums and exploring art collections.

These advantages clearly emphasize the use and benefits of virtual learning spaces and exhibitions that can be explored by anyone, anywhere. Although this project approach produces rich interfaces for information exploration, it lacks a social-based feedback and user-centered approach for improving these virtual learning spaces. In particular, it lacks the possibility for users to access the information to create personalized and focused views of these learning spaces. Although showing the museums as they are is the objective of this Google's project, it could take more advantage on the exploration of the information technologies to create more dynamic and interactive user-experience within such learning spaces, re-inventing the way museums are explored. Our approach, not centered on museum objects and exhibits, intends to provide the visitors a way of navigating through the objects in a repository via their semantic relations and aspects, allowing for focused acquaintance of knowledge on a desired thematic.

A more systematic and amusing way of information access via on-the-fly generation of virtual learning-spaces is Qwiki[4]. Qwiki is a search engine web application, which presents a summary of interactive information about millions of topics. The interactive summary (the learning-space), also called qwiki, is automatically generated, without human intervention, from static content available in several sources and machines like Google, Wikipedia, Fotopedia and Youtube. The generated content of a qwiki is a subtitled and narrated movie where text, images, maps, videos and any other multimedia content are merged together, focusing on the most important details about the searched topic, in order to produce a readable and meaningful summary.

Qwiki offers three main functionalities: (i) Movie visualization by topic search and related topics. During the movie, the subtitles present links for related qwikis and the user is also allowed to interact with the multimedia content by viewing photo details, movies or navigating in maps. (ii) Static content visualization. In this functionality, the user may read the text used as subtitles or browse the multimedia artifacts, where some of them point to related qwikis. (iii) Movie construction collaboration. This feature allows the user to submit new photos or movies to complete a qwiki, and also lets the user rate the narration voice. Qwiki is also connected to social web applications from where users can comment and provide feedback about the system.

Qwiki authors are starting to provide means (API and associated framework) for authoring qwikis. This way, companies and individuals would be able to create their own qwikis.

Qwiki presents an interesting approach for information visualization from where we may borrow some ideas. Nevertheless, the features we are aiming at providing are beyond Qwiki's. Qwiki uses the content or some meta-data information of displayed objects to decide about their relevance to the topic. In Geny, we plan to use resources semantic descriptions to guide the user during the knowledge acquisition process, taking advantage of Semantic Web technologies. Another weak point of Qwiki is that its social component is limited. We envisage to allow users for providing more feedback and changing the ontological layer upon

[4] http://www.qwiki.com/

the repository and the contents of the digital objects repository via Web 2.0 and Web 3.0 mechanisms.

Summing up, the projects discussed above are representative of what exits in this area of learning spaces as virtual places in the web for knowledge generation. While some projects already resort to modern technologies and mechanisms, all the presented projects lack some features that we, with Geny, intend to fulfill. The main aspects where Geny will innovate is on (*i*) the automatic generation of learning spaces from space specification, taking into account an ontology that organizes the information persisted in a repository and (*ii*) the social and semantic features for content navigation and for spaces enrichment with the visitor's knowledge.

3 Geny

We count on a long experience working from Language Processing techniques (grammar-based formal specifications, and automatic code generation from the specifications) to the area of annotated documents (or, in more general terms, to digital objects) and after that working with archives and museums. A careful review of the state of the art in the area of virtual exhibition and teaching spaces was here carried out to launch us into a still fresh research area.

This study led us to propose Geny and identify four main topics to which we intend to contribute with Geny's approach and tools:

- C1 - a Ontology driven Semantic Digital Object Repository (SemDOR), that aggregates heterogeneous information sources using an ontology;
- C2 - a Domain Specific Language to describe Learning Spaces (GenySL);
- C3 - a Learning Spaces Generator (GenyEngine), that automatically creates the learning spaces from a GenySL specification (a description of the space and its navigability), and resorting to the SemDOR;
- C4 - the Learning Spaces generated with capabilities for user feedback and improvements.

Figure 1 presents the main workflow and general architecture of Geny approach, where the four outcomes highlighted are embodied.

3.1 Methodology

To reach the objectives we decided to rely on a set of well known and sound approaches that we are used to apply in the context of language processing and eLearning systems. We overview the main frameworks, underlying each contribution.

3.1.1 Ontology Driven Approach to Knowledge Representation

Knowledge has known several ways of being represented in a computer. However, ontologies have been accepted as the most generalized approach for such requirement. It allows for constraining, expressing and analyzing the meaning of concepts and relations in the domain of knowledge subjacent to a given discourse. Moreover, when superimposed upon a collection of digital objects (as we intend to do in Geny) it gives semantic to the repository and therefore may allow for a more intuitive navigation on the virtual exhibition taking advantage of queries closer to the domain of discourse. This ontological approach for knowledge representation would also enable the automatic generation of learning spaces closer to the knowledge the visitor is looking for.

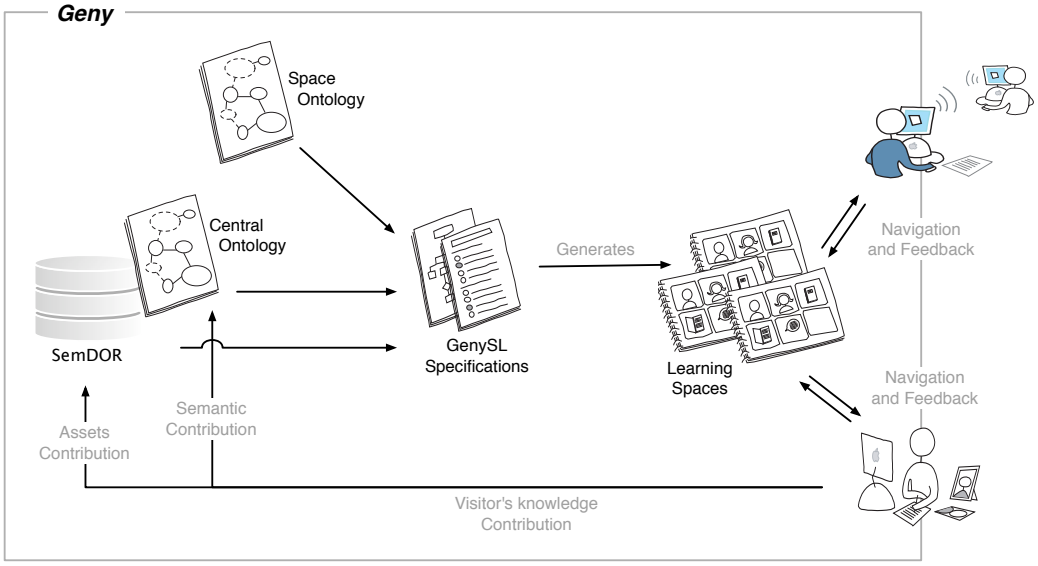


Figure 1 Geny's Workflow and General Architecture.

3.1.2 Domain Specific Languages and Generative Programming Approaches

GenySL is biased to a well-defined domain, which enables the use of domain specific languages (DSL) development approaches to define its design goals. We enrich such approach with the notion of templates for a more beneficial framework.

The major pros of such framework are that the expressive power and ease of use of DSLs reduces the time and background required to write programs that solve problems in a specific domain, when compared with a general purpose programming language [16, 12]. And the use of templates along with DSLs increase system efficiency (less things to be described), and increases the number of heterogeneous results [1].

A similar approach using template-aware DSL and a generative approach for creating embedded applications has already proven useful in Navegante, where a DSL allows the specification of an application written in a small number of statements, based on a previous created template [5].

In the overall, the adoption of such approach will raise several benefits. The following list presents some of them:

- a previously created set of templates can be shared by many applications, which means that improvements made to templates, improve all applications;
- learning spaces may be created automatically, once the DSL program is written;
- learning spaces may be easily created and maintained even for someone without the required knowledge to perform such task.

3.1.3 Social and Semantic Web Development Approaches

In the context of Geny, we want to take advantage of Web 2.0 (Social) and Web 3.0 (Semantic) to enable social interaction and structured information exchange. Web 2.0 offers new opportunities for collaboration between users having the main role the user as producer. O'Reilly in [13] argues that the network effects have an important role in the user

involvement (in our case in virtual exhibitions and learning spaces) in terms of collaboration and knowledge exchange. To support automatic information processing by computers, Tim Berners-Lee [2] proposes the Semantic Web, where machines can read Web pages much as humans do. Intelligent agents have a key role in the transition from Web 2.0 to Web 3.0, transforming the unstructured information of Web 2.0 in structured and interrelated semantic information, which is the aim of Web 3.0. In the context of Geny, we will take profit of these technologies for user interaction in two aspects. On the one hand to allow for the introduction of new knowledge to the system and, on the other hand, to enable sophisticated navigation based on semantics.

3.2 Plans

Considering the workflow and the general architecture of Geny, presented in figure 1, and after describing the methodological support, we decided to partition the work into four technical tasks plus one for validation:

- T1 - Construction of a Central Ontology to describe the knowledge base embodied in the Digital Object Repository;
- T2 - Design of a Domain Specific Language, GenySL, to specify Learning Spaces;
- T3 - Automatic Construction of Learning Spaces based on GenySL specifications;
- T4 - Feeding the Knowledge Base with the information collected from the Learners and sharing learning experiences using social media;
- T5 - Approach Validation using real Case Studies.

Task T1 is responsible to deliver contribution C1; task T2 clearly will produce the DSL that we identify above as contribution C2; task T3 is aimed at building the tool that we consider as the main result of Geny project, referred to as contribution C3; contribution C4 is obtainable from the tool produced by T3 and enriched with the outcome of task T4. Task T5 has a crucial influence in all the four contributions; it provides the motivation for project, the requirements for C1 and C2, the usability tests for C3, and the effectiveness assessment of the last contribution C4.

Due to the fact that T5 has as main duty the pragmatic (or case-study based) validation of all the project deliverables, it is the tail of the list of tasks. But actually T5 will be the provider of requirements for all the other tasks.

Clearly T1 involves the analysis of a specific DOR and the design and implementation of a case-oriented Ontology. Notice that the ontology is composed of two parts: a semantic network with the concepts and relations (taxonomic or not) among them describing the case domain; and the instances network, mapping the concepts into their occurrences, i.e., the objects stores in the DOR. To create a concrete ontology, ontology representation schemas must be studied compared and one out of them must be selected.

T2 is also basilar for the remaining tasks. T2 deals with the design of GenySL which embodies two main concerns: the capability to refer (in conceptual terms) to the digital objects that will be exposed and explored in the learning spaces; and the capability to described the space itself, how it is structured and what objects will fit in each part. For that a space ontology will be defined.

Task T3 is concerned with the development of the learning spaces Generator. This work completely relies on the GenySL. The objective of this task is the definition of an approach, and the work on its implementation, to build a website that realizes a virtual learning space, given a definition of that learning space written in the GenySL specification language

(outcoming from T2). The basic idea is to plan and develop a language processor (similar to a compiler) that transforms a GenySL specification into that website.

The aim of T4 is to improve the Learning Spaces specified with GenySL and generated with GenyEngine (created in task T3) with gadgets to increase the interactions of the visitor with the space and the collection of feedback that can enrich the DOR or the Central Ontology.

4 Summary

A learning space is a framework equipped with information resources to educate all sorts of people (not necessarily "students" in the classical sense); it is not restricted to a classroom nor to eLearning. Based in this statement, several works study the learning spaces virtualization. This paper presents a representative state-of-the-art concerning virtual learning spaces creation for knowledge generation. The main drawbacks of the discussed works are the weak user participation in the enrichment of assets repository and the lack of semantic exploitation of the digital resources. In this sense, we propose a novel project, named Geny, for virtual learning spaces creation where semantics and user feedback are the main concerns. Due to Geny's general application, we believe that it can help in the preservation of cultural heritage, namely oral history, clearly contributing for the well-being of different human communities.

Acknowledgements The Geny project discussion, definition and proposal would not be possible without the remaining members of the team. Hereby we thank them for their support and enthusiast work: Daniela da Cruz, José João Almeida, Nuno Carvalho and Paulo Alves.

References

1 R. Barrett. *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial Mathematics, 1994.

2 Tim Berners-Lee and Mark Fischetti. *Weaving the Web : The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. Harper San Francisco, Sep. 1999.

3 Anja L. Blanc, Jonathan Bunt, Jim Petch, and Yien Kwok. The virtual learning space: an interactive 3D environment. In *Proceedings of the tenth international conference on 3D Web technology*, Web3D '05, pages 93–102, New York, NY, USA, 2005. ACM.

4 Malcom Brown. *Learning Spaces*, pages 12.2–12.22. EDUCAUSE e-Books, 2005.

5 Nuno Carvalho, Alberto Sim??es, Jos?? Jo??o Almeida, Pedro Rangel Henriques, and Maria Jo??o Varanda Pereira. PFTL: A systematic approach for describing filesystem tree processors. In Raul Barbosa and Luis Caires, editors, *INForum'11 — Simp??sio de Inform??tica (CoRTA2011 track)*, pages 222–233, Coimbra, Portugal, Setembro 2011. Dep. de Eng. Inform??tica da Universidade de Coimbra.

6 Cristina Ghiselli, Alberto Trombetta, Bozzato Loris, and Elisabetta Binaghi. Semantic web meets virtual museums: The domus naturae project. In *In Proceedings of Digital Culture and Heritage (ICHIM05)*, Paris, France, 2005. Archives & Museum Informatics Europe (AMIE).

7 M. Goos. Creating learning spaces. In *The Annual Clements/Foyster Lecture*, 2006.

8 Eero Hyvönen, Eetu Mäkelä, Mirva Salminen, Arttu Valo, Kim Viljanen, Samppa Saarela, Miikka Junnila, and Suvi Kettula. MuseumFinland-finnish museums on the semantic web. *Web Semant.*, 3(2-3):224–241, Oct. 2005.

9 Laurence Johnson and Alan Levine. Virtual worlds: Inherently immersive, highly social learning spaces. *Theory Into Practice*, 47(2):161–170, 2008.

10 Maged N. Kamel Boulos, Lee Hetherington, and Steve Wheeler. Second life: an overview of the potential of 3-D virtual worlds in medical and health education. *Health Information and Libraries Journal*, 24(4):233–245, Dec. 2007.

11 Mike Keppell, Kay Souter, and Matthew Riddle. *Physical and Virtual Learning Spaces in Higher Education: Concepts for the Modern Learning Environment*. IGI Global, 1 edition, Jul. 2011.

12 Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

13 Tim O Reilly and O Reilly Media. What is web 2.0: Design patterns and business models for the next generation of software. *Design*, 65(65):17–37, 2007.

14 Kay Souter, Matthew Riddle, Warren Sellers, and Mike Keppell. Spaces for knowledge generation. Technical report, La Trobe University, 2011.

15 Mark Stevenson and Kate Fernie. Personalised access to cultural heritage spaces - PATHS annual report. Technical report, University of Shefield, 2012.

16 Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN NOTICES*, 35:26–36, 2000.

Automatic Test Generation for Space

Ulisses Araújo Costa[1], Daniela da Cruz[2], and Pedro Rangel Henriques[3]

1 **VisionSpace Technologies**
Rua Alfredo Cunha, 37, Matosinhos, Portugal
ucosta@visionspace.com
2 **Department of Informatics, University of Minho**
Campus de Gualtar, 4710-057, Braga, Portugal
danieladacruz@gmail.com
3 **Department of Informatics, University of Minho**
Campus de Gualtar, 4710-057, Braga, Portugal
pedrorangelhenriques@gmail.com

Abstract

The European Space Agency (ESA) uses an engine to perform tests in the Ground Segment infrastructure, specially the Operational Simulator. This engine uses many different tools to ensure the development of regression testing infrastructure and these tests perform black-box testing to the C++ simulator implementation. VST (VisionSpace Technologies) is one of the companies that provides these services to ESA and they need a tool to infer automatically tests from the existing C++ code, instead of writing manually scripts to perform tests. With this motivation in mind, this paper explores automatic testing approaches and tools in order to propose a system that satisfies VST needs.

1998 ACM Subject Classification D.2.5 Software Engineering, Testing and Debugging

Keywords and phrases Automatic Test Generation, UML/OCL, White-box testing, Black-box testing

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.185

1 Introduction

Since ever, every industry use testing methods to discover problems in early stages of the development process to improve the products quality, and software industry is not an exception. Miller [22] describe the utility of software testing as:

> *The general aim of testing is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances.*

In the most recent period of software history the integration of software testing as an important step in the process of software development opened up to the origin of *xUnit* [7] tools and Agile software development. Also, ESA started to use manual written tests as a part of their software development processes.

Using manual written tests is tedious, time consuming and error-prone. Lots of functions/methods need full code coverage and this practice leads to incomplete test suites; as it is hard to create tests that cover specific code paths, many hidden bugs can be left. Many times a supervision leaded by the developer is needed to assure that the right paths in the code are being tested, specially regarding black-box testing.

Nowadays we start to observe a rapid increase in the automatic test generation field.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 185–203

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

1.1 Goals

This document correspond to the first milestone in the author's dissertation (developed under a partnership agreement between UM and VST) aimed at producing a tool that is able to automatically generate interesting testcases for the C++ ESA's Operational Simulator.

This document reviews the most studied techniques and the tools that implement them in order to choose the best set of suitable techniques to incorporate in an automatic testing generator to the Ground Segment infrastructure, specially the Operational Simulator at ESA.

Two different techniques emerge for different purposes, Structural Techniques and Functional Techniques, known respectively as White-box [8] testing and Black-box [9] testing. Functional testing is the most common at ESA, because of the calculation complexity behind the Operational Simulators.

A brief discussion will be presented regarding White-box testing vs. Black-box testing and then some automatic generation techniques will be discussed in more detail. Furthermore the potential of the described tools will be explained, and how they can help on solving the problem VST has nowadays. First of all an explanation about the Operational Simulator Infrastructure will be provided.

1.2 Operational Simulator Infrastructure

ESA's Operational Simulator called Simulation Infrastructure for the Modeling of SATellites (SIMSAT) is a satellite simulator that model and simulate the behavior of satellites in order to allow operators[1] train more effectively and help them to define the satellites' operational processes.

The simulator consists of operational models of the various internal components of the satellite from their main computer to its payload (instruments aboard the satellite), which interact with each other and thus define the behavior of the satellite. VST has participated in the development of tests to validate the operational simulator. The development of these simulators is based on operating rules simulation of ESA – Simulation Model Portability (SMP)[2], as well as in infrastructure SIMSAT simulation. This standard is infrastructure agnostic of any space specific model, so any other needs of simulation can be used, such as defense, transport, energy, etc. Here is a brief description of each component in SIMSAT:[3]

SIMSAT Kernel this is a generic simulation infrastructure providing the framework for the running of space systems simulators.

SIMSAT Man-Machine Interface (MMI) this is a generic Graphical User Interface enabling the user interaction with the simulator's components.

Ground Models this is a family of SIMSAT compatible models enabling a realistic simulations of all ground systems between the spacecraft (or spacecraft model) and the control centre at European Space Operations Centre (ESOC).

---

[1] Operators are responsible for the operation of the satellite after its launch.

[2] SMP is based on the ideas of component-based design and Model Driven Architecture (MDA) as promoted by the Object Management Group (OMG) and is based on the open standards of UML and XML. One of the basic principles is the separation of the platform specific and platform independent aspects of the simulation model. This protects the investments in the model from changes in technology by defining the model in a platform independent way, which can then be mapped into different technologies. Further the SMP specification provides standardised interfaces between the simulation models and the simulation run-time environment for common simulation services as well as a number of mechanisms to support inter-model communication. [1, 2, 3, 4, 5]

[3] Information in: http://www.egos.esa.int/portal/egos-web/products/Simulators/simsat/intro-sim.html

Emulator Suite On-board Processor Emulators support the execution in satellite simulators of the real flight software.

Generic Models a set of generic space models that ease the developments of the spacecraft models used in operational simulators.

Ground Systems Test and Validation Applications (GSTV) this is a family of test simulators that are based on the generic simulators infrastructure components listed above and are able to support the different levels of testing of ground infrastructure systems.

Moreover the SIMSAT Kernel is made up of several components:[4]

Scheduler is responsible for the co-ordination and processing of all events within the Simulation Kernel. An event on the schedule identifies an action that needs to be performed at a specified point in simulated time.

Mode Manager is the simulation state machine. The Simulation has a number of operational modes, which control the operation of the simulation.

Time-Manager is responsible for maintaining and providing models and the MMI with the correct simulation-Time. It provides time in four formats, Simulation-Time, Epoch-Time, Zulu-Time and Correlated Zulu-Time. this is a family of SIMSAT compatible models enabling a realistic simulations

Logger supports the recording of Kernel or model events that occur during a simulation. The log in which the current simulation messages are written is called the active log. The logger also provides a view of the simulation event history in an MMI during a simulation session.

Visualization manager is responsible for making the values of both model and Kernel data items available for display in an MMI.

State-vector manager is responsible for the saving and restoring of the state of the simulation. Its main purpose is to allow the Simulation State, at any point in the simulation, to be saved. This allows the user to return to an earlier simulation scenario.

Command handler is responsible for the reception and execution of Kernel and user defined commands.a set of generic space models that ease the developments of the spacecraft models used in operational.

Command procedure interpreter is responsible for the interpretation of command procedures. A command procedure contains Kernel and User defined simulator commands and supports a procedural language to control the flow of these commands. The execution of command procedures is controlled directly from the MMI.

Right now, to be able to perform tests in the Operational Simulator, in order to validate SIMSAT, VST Engineers need to write scripts that perform simulations and validate the results using GUI interfaces (SIMSAT MMI). This job can be tedious and difficult to replicate.

So a first solution will have to go through a preliminary study of the tools that currently exist with which we can generate tests automatically. By studding these tools we do not hope to find the perfect solution, but combine techniques to obtain an optimal solution to improve VST work.

[4] More information in: http://www.egos.esa.int/portal/egos-web/products/Simulators/SIMSAT/

1.3 White-box vs Black-box testing

In this subsection is discussed the two most common approaches for testing: White-box and Black-box testing.

In White-box testing the tester needs to understand the internals of the code to be able to write tests for it. The goal of selecting test cases that test specific parts of the code is to cause the execution of specific spots in the software, such as statements, branches or paths. This technique consists in analyzing statically a program, by reading the program code and using symbolic execution techniques to simulate abstract program executions in order to attempt to compute inputs to drive the program along specific execution paths or branches, without ever executing the program. Control Flow based testing approach can be useful to analyze all the possible paths in the code and write unit tests to cover multiple paths. The CFG (Control Flow Graph) of the program can be built, test inputs can be generated to make any path execute regarding a given criterion: Select all paths; Select paths to achieve complete statement coverage [8, 24]; Select paths to achieve complete branch coverage [28, 8]; or Select paths to achieve predicate coverage [8, 24].

Data Flow Testing is designed into looking at the life cycle (creation, usage and destruction) of a particular piece of data and observe how it is used along the CFG, this ensures that the number of paths is always finite [27].

Opposite to White-box testing, Black-box testing is based on functionality, so the tester observes a system based on its functional contracts and writes the pairs of inputs and the expected outputs. This approach is used for unit testing of single methods/functions, integration testing of combinations of the methods/functions, or even final system testing.

This document is organized as follows. In section 2 the important testing approaches in use—Specification-based testing and Constraint-based generation—are briefly revisited and, for each one, the most relevant tools are identified. In section 3 some of the tools referred are experimented in order to be compared. Our proposal for a test generation system is introduced in section 4. The document is concluded in section 5.

2 Testing Tools Approaches

In this section, a study of the most recent tools that use Specification-based, Constraint-based, Grammar-based and Random-based tests generation approaches for the most popular languages - C, JAVA and C# will be presented.

2.1 Specification-based Generation Testing

Specification Based Testing refers to the process of testing a program based on what its specification or model says its behavior should be. In particular, can be generated test cases based on the specification of the program's behavior, without seeing an implementation of the program. So this clearly a way of Black-box testing.

With this technique the testing phase and development phase can be started in parallel, we do not need the implementation to start the development of test cases. The only thing needed is the functional contracts and/or oracles[5] for each function/method.

Since the 90's there have been some effort into using specifications to try to generate test cases such as Z specifications [19, 30], UML statecharts [25],VDM [6] or ADL specifications [29]. These specifications typically do not consider structurally complex inputs and

[5] A test oracle determines whether or not the results of a test execution are correct [26].

these tools do not generate JUnit test cases. Nowadays there are some tools out there that can perform Specification-based Testing approach:

Conformiq is a commercial Tool Suite that generates human-readable test plans and executable test scripts from Java code, state charts and UML[6].

MaTeLo stands for Markov Test Logic and is a commercial tool that generates test sequences from a collection of states, transitions, classes of equivalence, types, sequences, global variables and test oracles using their user interface[7].

Smartesting CertifyIt is a commercial tool that generates test cases from a functional model, as UML[8].

T-Vec is a commercial tool that generates test cases from modeling tools available from T-VEC or third-party vendors[9].

Rational Tau is an IBM commercial tool that provides automated error checking, rules-based model checking, and a model-based explorer using UML[10].

The relevant ones or the recent open-source ones will be discussed.

2.1.1 Spec Explorer

This is a Microsoft model-based testing that uses one software modeling languages, the AsmL (Abstract State Machine Language). This modeling language provides the foundations of the Spec Explorer[11] tool and Spec# that is a formal language for API contracts (influenced by JML, AsmL, and Eiffel), which extends C# with constructs for non-null types, pre-conditions, post-conditions, and object invariants[12]. These tool is already available to users and is in a very mature phase.

The user of Spec Explorer writes a model of the system and sets the possible values for some properties in his code, furthermore the user also provides a scenario. These scenarios are simple sets of calls to methods without their parameters (remember that this is Spec Explorer job). Then Spec Explorer will generate a visual graph where each node represents a state of the system and the arrows represent a call to some method. It searches throw all possible sequences of methods invocation that do not violate the contracts (pre, pos conditions) and that are relevant to a user-specified set of test properties. After that we can generate from this visual graphs the unit tests (the arrows) and the test cases (a graph).

2.1.2 JMLUnit

JMLUnit [15] is a tool that automates the generation of oracles for JAVA testing classes. This tool monitors the specified behavior of the method being tested to decide whether the test passed or failed. This monitoring is done using the formal specification language runtime assertion checker. The main idea behind these tools is to translate the pre- and post-conditions methods into the code of the testing method.

[6] See more at: `http://www.conformiq.com/products.php`
[7] See more at: `http://www.all4tec.net/index.php/All4tec/matelo-product.html`
[8] See more at: `http://www.smartesting.com/index.php/cms/en/product/certify-it`
[9] See more at: `http://www.t-vec.com/`
[10] See more at: `http://www-01.ibm.com/software/awdtools/tau/`
[11] See more at: `http://research.microsoft.com/en-us/projects/specexplorer/`
[12] See more at: `http://research.microsoft.com/en-us/projects/specsharp/`

The pre-conditions became the criteria for selecting test inputs, and the post-conditions provided the properties to check for test results. So, the post-conditions became the test oracles.

This tool uses the JML [12] specification language to annotate JAVA methods code with pre- and post-conditions and automatically generate JUnit test classes from JML specifications.

2.1.3 TestEra

TestEra [21] can be used to perform automated specification-based testing of JAVA programs. This framework requires as input a JAVA method, a formal specification[13] of the pre and post-conditions of that method, and a bound that limits the size of the test cases to be generated.

With the pre-condition it automatically generates all non-isomorphic test inputs up to the given bound. It executes the method on each test input, and uses the method post-condition as an oracle to check the correctness of each output. This tool uses Alloy's[14] SAT system to analyze first-order formulae. The authors claim that have used TestEra to check several JAVA programs including an architecture for dynamic networks, the Alloy-alpha analyzer, a fault-tree analyzer, and methods from the JAVA Collection Framework.

2.1.4 Korat

Korat [11] is a mature framework for automated testing structurally complex inputs of JAVA programs. Given a formal specification for a method, Korat[15] uses the method pre-condition to automatically generate all (non-isomorphic) test cases up to a given small size. Korat then executes the method on each test case, and uses the method post-condition as a test oracle to check the correctness of each output.

To be able to generate test cases for a method, Korat uses a predicate and a bound on the size of its inputs, Korat generates all (non-isomorphic) inputs for which the predicate returns *true*. Korat generates all the possible input spaces regarding the predicate and monitor the predicate's executions to be able to prune large portions of the search space.

The writing of a predicate is done using JAVA language and in most cases can be written the first thing that cames to programmer's head to restrict the input space. But for more complex structures it is better to understand how the matching algorithm work to be able to write a fast verifiable predicate.

Unfortunately the test derivation tool using Korat (that also uses JML) is not available to the public.

2.2 Constraint-based Generation Testing

Constraint Based Testing [18] can be used to select test cases satisfying specific constraints by solving a set of constraints over a set of variables. The system is described using constraints and these can be solved by SAT solvers.

---

[13] Specifications are first-order logic formulae.

[14] Alloy is a first-order declarative language based on sets and relations. The Alloy Analyzer is a fully automatic tool that finds instances of Alloy specifications: an instance assigns values to the sets and relations in the specification such that all formulae in the specification evaluate to true.

[15] See more at: `http://korat.sourceforge.net/`

Constraint programming can be combined with symbolic execution, regarding this approach a program is executed symbolically, collecting data constraints over different paths in the CFG, and then solving the constraints and producing test cases from there. There are some tools out there, like:

Euclide for verifying safety properties over C code using ACSL annotations, CPBPV for program verification.
OSMOSE a tool that uses concolic execution and path-based techniques over machine code.
GATeL for Lustre language to generate test sequences[16].

Here two tools will be explained, one proprietary and other academic.

2.2.1 Pex

Pex [32] is an automatic white-box test generation tool for .NET. Starting from a method that takes parameters, Pex performs path-bounded model-checking by repeatedly executing the program and solving constraint systems to obtain inputs that will steer the program along different execution paths. This uses the idea of dynamic symbolic execution [33]. Pex uses the theorem prover and constraint solver Z3[17] to reason about the feasibility of execution paths, and to obtain ground models for constraint systems.

Pex came with Moles that helps to generate unit tests. These tools together are able to understand the input (by analyzing branches in the code: declarations, all exceptions throws operations, if statements, asserts and .net Contracts). With this information Pex uses Z3 constraint solver to produce new test inputs which exercise diferent program behavior.

The result is an automatically generated small test suite which often achieves high code coverage.

Pex can be used in a project, class or method (which makes it a very helpful and versatile tool). After the analysis process the "Pex Explorarion Results" shows the *input* × *output* pairs selected for each test case for the method, here it also shows the percentage of the test coverage.

2.2.2 PathCrawler

This is an academic tool based on dynamic and static analysis [34], it uses constraint logic programming to generate the Test-cases. PathCrawler[18] executes an instrumented function for each function under test with the generated inputs, it preserves this information to not cover the same path.

This tool supports assertions in any point in the code and pre-conditions regarding the input values.

2.3 Grammar-based Generation Testing

In this approach inputs to a system under test are defined by a context-free grammar. The language of the grammar contains all possible test cases. Using this approach to describe the syntax of the input to the system under test proves to be very helpful to test network protocols [31, 20] and parsers and compilers [13, 14].

[16] See more at: `http://www-list.cea.fr/labos/gb/LSL/test/gatel/index.html`
[17] See more at: `http://research.microsoft.com/en-us/um/redmond/projects/z3/`
[18] See more at: `http://www-list.cea.fr/labos/gb/LSL/test/pathcrawler/index.html`

2.3.1 ASTGen

ASTGen [17] is a JAVA framework that automates testing of refactoring engines: generation of test inputs and checking of test outputs. The main technique is an iterative generation of structurally complex test inputs. ASTGen[19] allows developers to write imperative generators whose executions produce input programs for refactoring engines. More precisely, ASTGen offers a library of generic, reusable, and composable generators that produce abstract syntax trees (ASTs).

So, ASTGen ensures the production of test inputs instead of the developer produce them. The developer needs to write a generator whose execution produces thousands of programs with structural properties that are relevant for the specific refactoring being tested. This tool has found 21 bugs in Eclipse and 26 bugs in Netbeans applications.

2.4 Random-based Generation Testing

In the random testing approach, test inputs are selected randomly from the input domain of the system. To have a random testing suite first we must identify the input domain, after that select test inputs independently from the domain, then the system under test is executed on these inputs, the results are compared to the system specification, an oracle.

Random testing gives us an advantage of easily estimating software reliability from test outcomes. Test inputs are randomly generated according to an operational profile, and failure times are recorded. The data obtained from random testing can then be used to find bugs or non expected behaviors.

The main problem regarding random generation is the problem of the coverage, it is possible that it will not be broad enough. And furthermore it can be too sparse to actually test specifics parts of the program. Either way, this technique proves to be very effective for testing compilers.

2.4.1 Csmith

Csmith [36] is a black-box random tests generator that is able to generate C programs conform to the C99[20] standard. This is a very recent tool that already discover more than 195 bugs in LLVM and 79 bugs in GCC. With Csmith we are able to generate random programs with unambiguous meanings (undefined behavior or unspecified behavior). Does not attempt to generate terminating program, so they use timeouts for long time consuming generated programs. And the main supported features right now are: Arithmetic, logical, and bit operations on integers, Loops, Conditionals, Function calls, Const and volatile, Structs and Bitfields, Pointers and arrays, Goto, Break and continue. The generation of code regarding this features can be tuned using the command line program.

2.4.2 QuickCheck for JAVA

QuickCheck was originally a combinator library for the Haskell[21] programming language [16]. Later on QuickCheck philosophy spread to other programming languages like: JAVA, Erlang, Perl, Ruby and JavaScript.

---

[19] See more at: `http://mir.cs.illinois.edu/astgen/`
[20] See more at: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf`
[21] See more at haskell.org

QuickCheck works by generating high amounts of data (within the method domain) and checking it against a given property, it is expected to create a wide range of the input domain, thus increasing the chances of giving more test coverage.

3 Using the Tools

After introducing the theory and the techniques that support each tool, some of the tools will be demonstrated in action, resorting to small but illustrative examples on how each tool can help us to find good test cases.

3.1 PathCrawler

Concerning the first case a simple example will be used based on a function that performs a multiplication (see listing 1), creating a simple branch on the code.

Listing 1 2D point struxture and multiplication function.

```
typedef struct s {
    int x;
    int y;
}Point;

int Multiply(Point p) {
    if(p.x * p.y == 42) return 1;
    else return 0;
}
```

Pointers were tried instead of coping the structure as a parameter to *Multiply* function, but PathCrawler was not able to run.

Nevertheless, PathCrawler was able to give a full coverage for this simple function as you can see in Table 1.

Table 1 Output Table for *Multiply* function using PathCrawler.

Result	p	return value
✓	Point{x=1,y=42}	1
✓	Point{x=177407,y=109471}	0

Regarding our second example a function that performs a binary search (listing 2) in order to find if a number is in a given range (between two bounds).

A function that PathCrawler gives to us has been used: *pathcrawler_assert*, this function can be used at any location in the program under test, and will force PathCrawler to generate test cases to cover both the case where its argument is true and the case where it is false. This feature may be seen as another way to write an oracle.

The results were interesting: 31 covered paths and 44 infeasible paths and the test was interrupted by PathCrawler, because PathCrawler reach the maximal test session time (the user can increase this number, but for this example is left the default value).

A further analysis of the results demonstrated that 28 out of the 44 infeasible paths discovered appeared when PathCrawler tried to do the assertion in line 8. No pre-condition

Listing 2 Binary search.

```
int BSearch(int x, int n) {
    return BinarySearch(x, 0, n);
}

int BinarySearch(int x, int lo, int hi) {
    while (lo < hi) {
        int mid = (lo+hi)/2;
        pathcrawler_assert(mid >= lo && mid < hi);
        if (x < mid) { hi = mid; }
                else { lo = mid+1; }
    }
    return lo;
}
```

was written, so PathCrawler does not know that this is a pre-condition for *BinarySearch* function: $lo \leq x < hi$. In Table 2 is shown some of the test inputs generated for this example.

Table 2 Output Table for *BSearch* function using PathCrawler.

Result	x	n	return value
✓	-189424	-140714	0
✓	157819	0	0
✓	1	1610612736	2
✓	2	805306368	3
✓	11	1610612736	12

PathCrawler was tried with the function on listing 3, that calculates the year of the n^{th} day after 1980-01-01.

The result was unexpectedly *unknown*. PathCrawler was unable to trace even one path in our code, the number of k-path's could be increased but with no success for this example.

3.2 Pex

Regarding Pex, we used the same examples shown previously adapted to C# language. Because C# is a more expressive language than C our examples will be improved with some other OO and C# specific features like Exceptions and Debug.Assert calls. In fact Pex can also support a lot more features that are present in C# language like .NET Contracts and many more.

Listing 4 is the simple implementation of a 2D *Point* class that has been created to have special behavior, under a certain condition $x \times y \equiv 42$ it is supposed to throw an exception.

So, as was described earlier, Pex will try to generate such input as it is possible (in a given amount of time) to traverse all the paths inside the code. The output table can be seen in Table 3, with the inputs and outputs that Pex found to ensure a full coverage of the code.

Listing 3 Calculate year for n^{th} day after 1980-01-01.

```
int IsLeapYear(int year) {
  return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
}
int FromDayToYear(int day) {
  int year = 1980;

  while (day > 365) {
    if (IsLeapYear(year)) {
      if (day > 366) {
        day -= 366;
        year += 1;
      }
    } else {
      day -= 365;
      year += 1;
    }
  }
  return year;
}
```

Listing 4 2D point class and multiplication method that throws an exception.

```
public class Point {
  public readonly int X, Y;
  public Point(int x, int y) { X = x; Y = y; }
}

public class Multiply {
  public static void multiply(Point p) {
    if (p.X * p.Y == 42)
        throw new Exception("hidden bug!");
  }
}
```

Table 3 Output Table for *multiply* method using Pex.

Result	p	Output/Exception	Error Message
✗	null	NullReferenceException	Object ref. not set to an instance of an object.
✓	new Point{X=0,Y=0}		
✗	new Point{X=3,Y=14}	Exception	hidden bug!

Pex was successful to reach the *Exception* path inside the code. Of course this is not always possible, since sometimes the functions inside the *if* statement does not have inverse function.

Pex can also be very helpful checking assertions and contracts in .net code. A binary

Listing 5 Binary search with debug assertion.

```
public class Program {
  public static int BSearch(int x, int n) {
    return BinarySearch(x, 0, n);
  }
  static int BinarySearch(int x, int lo, int hi) {
    while (lo < hi) {
      int mid = (lo+hi)/2;
      Debug.Assert(mid >= lo && mid < hi);
      if (x < mid) { hi = mid; } else { lo = mid+1; }
    }
    return lo;
  }
}
```

search algorithm was written and an assertion was also written in the middle of our code. Pex was able to generate an input that could not pass in the assertion inerted in our code, as can be seen in Table 4.

Table 4 Output Table for *BSearch* method using Pex.

Result	x	n	result	Output/Exception
✓	0	0	0	
✓	0	1	1	
✓	0	3	1	
✗	1073741888	1719676992		TraceAssertionException
✓	1	6	2	
✓	50	96	51	

Now we have a more complex example, a function that returns the year of the n^{th} day after 1980-01-01. Pex was able to generate some important test cases, but it has reached the limit amount of time to calculate interesting paths in the code, this boundary prevents Pex from getting stuck when the program goes into an infinite loop.

Pex was unable to discover the year for day 366 and 7671 as we can see in Table 5. This problem occurred because Pex by default has a maximum number of conditions, this avoids never ending functions and still has a result from Pex. In this particular case we could increment the number of $MaxConditions$: $[PexMethod(MaxConditions = 10000)]$.

3.3 Korat

Like was explained before, Korat generates a graphical representation of the structure instances that validates the property *repOK*. This property was written using JAVA code.

In order to test the freelly available version of Korat, a Doubly Linked List structure was created in JAVA.

Now the *repOK* predicate method must be defined. This predicate method will check that the tree doesn't have any cycles and that the number of nodes traversed from root

Listing 6 Calculate year for n^{th} day after 1980-01-01 (Java).

```
public class Program {
  private static bool IsLeapYear(int year) {
    return (year % 4 == 0) && ((year % 100 != 0) || (year % 400 == 0));
  }
  public static void FromDayToYear(int day, out int year) {
    year = 1980;
    while (day > 365) {
      if (IsLeapYear(year)) {
        if (day > 366) {
          day -= 366;
          year += 1;
        }
      } else {
        day -= 365;
        year += 1;
      }
    }
  }
}
```

Table 5 Output Table for $FromDayToYear$ method using Pex.

Result	day	out year	Output/Exception
✓	0	1980	
✓	367	1981	
⚠	366		path bounds exceeded
✓	1023	1982	
✓	2561	1987	
✓	7874	2001	
⚠	7671		path bounds exceeded

matches the value of the field size. First was defined the properties about this data structure. The most relevant ones are property 5 in Figure 1 that ensures the structure and property 6 that ensures our doubly linked list does not have repeated elements.
Consider $e, e_1, e_2 \in LinkedListElement$ and i the index function: $i : LinkedListElement \rightarrow int$, that receives an element of $LinkedList$ and returns the position of that element in the structure. Consider also three new functions:

1. $Head(l)$ being l of type $LinkedList$ and meaning in Java code $l.Head$.
2. $Tail(l)$ being l of type $LinkedList$ and meaning in Java code $l.Tail$.
3. $size(l)$ being l of type $LinkedList$ and meaning in Java code $l.size$.

As a matter of avoiding verbosity two symbols were defined ($\Subset$ and $\Subset\!\!\Subset$, these symbols are used to define the $LinkedList$ invariants in Figure 1):

1. $a \Subset l$ being a of type $LinkedListElement$ and meaning that a is an element of the $LinkedList$ l.
2. $\{a, \ldots, z\} \Subset\!\!\Subset l$ meaning $a \Subset l \wedge \ldots \wedge z \Subset l$.

Listing 7 Linked list implementation.

```
public class LinkedList<T> {
  public static class LinkedListElement<T> {
    public T Data;
    public LinkedListElement<T> Prev;
    public LinkedListElement<T> Next;
  }
  private LinkedListElement<T> Head;
  private LinkedListElement<T> Tail;
  private int size;
}
```

We took the properties described in Figure 1 and use them to restrict the generation of structures as we can see in the following Java implementation code. Note that we using short-circuiting, so we return $false$ as soon as we can. This way Korat will be able to generate faster the instances matching our criteria.

The last step was defining the finitization method, this way we tell Korat how to bound the input space.

The properties in Figure 1 were taken and used to restrict the generation of structures using Java. So the *repOK* method that receives a *LinkedList* structure and returns *Bool* whenever this structure follows the invariants in 1 was defined. Using this specification Korat generated the 2 structures shown in Figure 2. In Figure 2a with 2 elements and in Figure 2b an instance with 5 elements.

3.4 Summary

After the experimental study of the selected tools, reported in the previous subsections, it was found that PathCrawler and Pex have different approaches regarding testcase generation. PathCrawler seems to be a very efficient tool to discover multiple infeasible paths in C code, because it uses a mix between static and dynamic analysis. When it finds a suitable input for a function it tries to execute collecting all the executed paths in the code. Pex on the other side just uses static execution and it is very efficient discovering all the feasible paths in C# methods. Pex was also used to perform testcase generation in C# classes, but the generated instances are too simple to perform more interesting tests. The *LinkedList* class

$$\langle \forall\, l : l \in LinkedList \;:\; Head(l) \equiv null \vee Tail(l) \equiv null \Leftrightarrow size(l) \equiv 0 \;\rangle \tag{1}$$

$$\langle \forall\, l : l \in LinkedList \;:\; Tail(l).Next \equiv null \;\rangle \tag{2}$$

$$\langle \forall\, l : l \in LinkedList \;:\; Head(l).Prev \equiv null \;\rangle \tag{3}$$

$$\langle \forall\, l : l \in LinkedList \;:\; size(l) \equiv 1 \Leftrightarrow Head(l) \equiv Tail(l) \;\rangle \tag{4}$$

$$\langle \forall\, l : l \in LinkedList \;:\; \langle \forall\, e_1, e_2 : \{e_1, e_2\} \subseteq\!\subseteq l \;:\; \langle \exists\, e : e \notin\!\notin l \;:\; e_1.Next \equiv e \wedge e_2.Prev \equiv e \;\rangle \;\rangle \;\rangle \tag{5}$$

$$\langle \forall\, l : l \in LinkedList \;:\; \langle \forall\, e_1, e_2 : \{e_1, e_2\} \subseteq\!\subseteq l \;:\; e_1 \equiv e_2 \Rightarrow i(e_1) \equiv i(e_2) \;\rangle \;\rangle \tag{6}$$

Figure 1 Invariants for class *LinkedList*.

Listing 8 Structures generation.

```
public boolean repOK() {
  if(Head == null || Tail == null)
    return size == 0;
  if(size == 1) return Head == Tail;
  if(Head.Prev != null) return false;
  if(Tail.Next != null) return false;
  LinkedListElement<T> last = Head;
  Set visited = new HashSet();
  LinkedList workList = new LinkedList();
  visited.add(Head);
  workList.add(Head);
  while (!workList.isEmpty()) {
    LinkedListElement<T> current =
                      (LinkedListElement<T>) workList.removeFirst();
    if (current.Next != null) {
      if (!visited.add(current.Next))
            return false;
      workList.add(current.Next);
      if(current.Next.Prev != current) return false;
      last = current.Next;
    }
  }
  if(last != Tail)
    return false;
  return (visited.size() == size);
}
```

was written in C# with many management methods implemented (Add, Remove, Find,...). Pex generated very simple *LinkedList*'s structures to perform automatic test generation for each implemented method. The problem is that the generated structures does not meet the properties about Doubly Linked Lists as it can be seen in Figure 3. Concerning Korat, this is The tool to generate complex data structures. The freely available part of Korat show potential in expressing rules to hedge the automatic generation of data structures.

In Table 6 we can see a brief comparison between all the experimented and mentioned tools, a more detailed conclusion is addressed in Chapter 5.

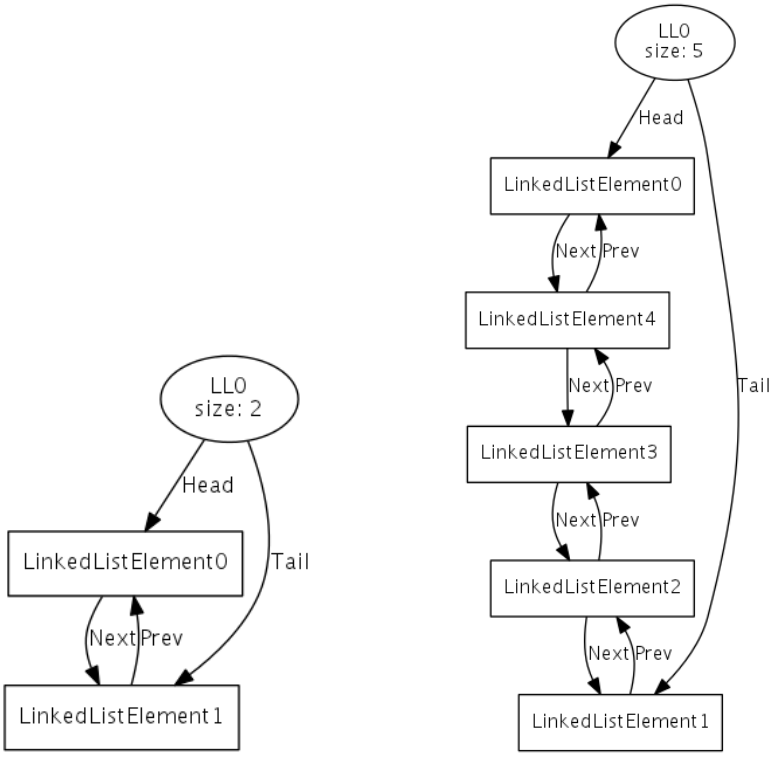
4 Generate Tests from Code+OCL

Since the Operational Simulator code is not familiar to us, regarding its implementation, it was decided to start solving this problem by inferring the UML+OCL from the existing code to be able to work on a more abstract level rather than the implementation. The idea is to extract tests from the inferred OCL, using the Partition Analysis described in [10] and at the same time generate tests directly from the code, using symbolic execution to complement the specification-based generation from OCL. The main goal is to extract as many tests as possible from a model and from the implementation to provide information to a feedback loop [35] test generation framework with two test prespectives, functional and structural, and from there be able to get a more refined set of tests.

A combination of both, symbolic execution from Pex and complex data generation from

Listing 9 Finitization method implementation.

```
public static IFinitization finLL(int nodesNum,int minSize,int maxSize)
{
  IFinitization f = FinitizationFactory.create(LL.class);
  IObjSet nodes = f.createObjSet(LinkedListElement.class,
                                 nodesNum, true);
  f.set("Head", nodes);
  f.set("Tail", nodes);
  f.set("size", f.createIntSet(minSize, maxSize));
  f.set("LinkedListElement.Next", nodes);
  f.set("LinkedListElement.Prev", nodes);
  return f;
}
```



(a) Instance with 2 elements for *LinkedList*

(b) Instance with 5 elements for *LinkedList*

Figure 2 Examples of generated instances from Korat for *LinkedList* class.

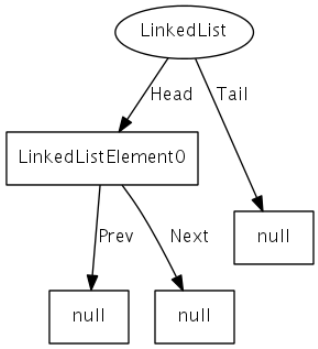
Korat, it will be designed and implemented to generate more interesting inputs for the methods under testing.

5 Conclusion

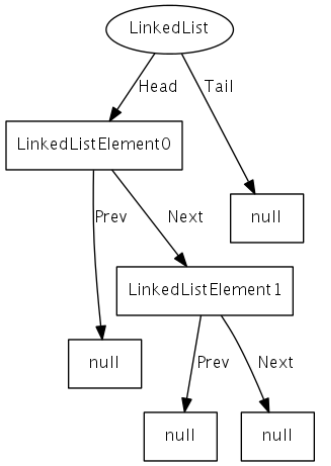
Looking for an efficient solution to automatically generate complete test sets for complex and critical C++ software, the state-of-the-art approaches in the area were studied and along

Table 6 Comparison of experimented and mentioned tools.

Name	Target Language	Black/White-box	Additional Input	Output	Comments
PathCrawler	C	White-box (symbolic execution)	Test vectors	Constraints about the executed paths	Too Complex
Pex	C#	White-box (symbolic execution)	–	Unit Tests	Poor generated data instances (objects)
Korat	JAVA	Black-box	Invariants written in JAVA	Graphical form of data structures (using Alloy-GraphViz)	Powerful generating valid data instances



(a) Example of Pex generated *LinkedList* instance to test *Remove* method



(b) Example of Pex generated *LinkedList* instance to test *Find* method

Figure 3 Examples of generated instances from Pex for *LinkedList* class.

the document some tools were introduced from methodological and experimental perspectives. Pex has proved to be a very powerful tool, aimed at offering a full coverage. However, the incapability for generating calling-methods sequences was a bit disappointing. With Microsoft's SpecExplorer we can already manually call sequences of methods; maybe a combination of this feature with Pex would make Pex a perfect all-in-one testing tool regarding .NET automatic testing tools. Concerning Korat, the expected improvement is just to write the invariants for a class instead of the *repOK* method, or maybe infer these invariants from the existing code. Writing the *repOK* method for very complex data structures requires some previous experience with Korat, but we think this is not a weakness, since the tester quickly gets used to write the *repOK* method in Korat. The only problem is that right now we can not fully automate the process without human help.

Considering the studied tools and thinking about a full automated test generation tool,

a clever composition among between Pex to ensure the maximum possible coverage, Korat to generate all the valid data structures and an automatic tool to generate calls to methods combinations would be the perfect tool.

At the end, it was proposed an approach based on the inference of tests from a Code+OCL. Concerning the OCL inference from C++ code, work will now be done on a tool that implements it. For that purpose, Frama-C will be explored, as it is well known that this tool is able to infer pre- and post-conditions [23] and interesting safety conditions from C source code.

References

1 European Space Agency. In *ECSS-E-TM-40-07 Volume 1A – Simulation modelling platform – Volume 1: Principles and requirements*, 2011 January.

2 European Space Agency. In *ECSS-E-TM-40-07 Volume 2A – Simulation modelling platform – Volume 2: Metamodel*, 2011 January.

3 European Space Agency. In *ECSS-E-TM-40-07 Volume 3A – Simulation modelling platform – Volume 3: Component model*, 2011 January.

4 European Space Agency. In *ECSS-E-TM-40-07 Volume 4A – Simulation modelling platform – Volume 4: C++ Mapping*, 2011 January.

5 European Space Agency. In *ECSS-E-TM-40-07 Volume A5 – Simulation modelling platform – Volume 5: SMP usage*, 2011 January.

6 Bernhard K. Aichernig. Automated black-box testing with abstract vdm oracles. In *Computer Safety, Reliability and Security: proceedings of the 18th International Conference, SAFECOMP'99*, pages 250–259. Springer, 1999.

7 Kent Beck. Simple Smalltalk Testing: With Patterns. Technical report, First Class Software, Inc., 1989.

8 Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

9 Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

10 Mohammed Benattou, Jean-Michel Bruel, and Nabil Hameurlain. Generating test data from ocl specification, 2002.

11 Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on java predicates. In *IN PROC. INTERNATIONAL SYMPOSIUM ON SOFTWARE TESTING AND ANALYSIS (ISSTA*, pages 123–133. ACM Press, 2002.

12 Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications, 2003.

13 C. J. Burgess. The automated generation of test cases for compilers. *Sofwtware testing, Verification and Reliability*, 4(2):81–99, June 1994.

14 C J Burgess and M Saidi. The automatic generation of test cases for optimizing fortran compilers. *Information and Software Technology*, 38(2):111–119, 1996.

15 Yoonsik Cheon, Yoonsik Cheon, Gary T. Leavens, and Gary T. Leavens. The jml and junit way of unit testing and its implementation. Technical report, 2004.

16 Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

17 Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 185–194, New York, NY, USA, 2007. ACM.

18 Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 17(9):900–910, 1991.

19 Hans-Martin Horcher. Improving software tests using z specifications. In *In Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*, pages 152–166. Springer-Verlag, 1995.

20 R. Kaksonen and Valtion teknillinen tutkimuskeskus. *A functional method for assessing protocol implementation security*. VTT publications. Technical Research Centre of Finland, 2001.

21 Sarfraz Khurshid and Darko Marinov. Testera: Specification-based testing of java programs using sat. *Automated Software Engg.*, 11:403–434, October 2004.

22 F. Miller. *Introduction to Software Testing Technology*, pages 4–16. Tutorial: Software Testing and Validation Techniques. IEEE Computer Society Press, 1981.

23 Yannick Moy. *Automatic Modular Static Safety Checking for C Programs.* PhD thesis, Université Paris-Sud, 2009.

24 S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, 14:868–874, June 1988.

25 Jeff Offutt and Aynur Abdurazik. Generating tests from uml specifications. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 416–429, Berlin, Heidelberg, 1999. Springer-Verlag.

26 Dennis Peters. Generating a test oracle from program documentation, 1995.

27 S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11:367–375, 1985.

28 Marc Roper. *Software Testing.* The International Software Engineering Series. McGraw-Hill, 1994.

29 Sriram Sankar, Roger Hayes, Sriram Sankar, and Roger Hayes. Specifying and testing software components using adl, 1994.

30 Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22:777–793, November 1996.

31 Oded Tal, Scott Knight, and Tom Dean. Syntax-based vulnerability testing of frame-based network protocols. In *PST'04*, pages 155–160, 2004.

32 Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg, 2008. Springer-Verlag.

33 Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23:2006, 2006.

34 Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: automatic generation of path tests by combining static and dynamic analysis. In *In: Proc. European Dependable Computing Conference. Volume 3463 of LNCS (2005) 281–292*, pages 281–292. Springer. ISBN, 2005.

35 Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In *In Proc. 3rd International Workshop on Formal Approaches to Testing of Software, volume 2931 of LNCS*, pages 60–69, 2003.

36 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46:283–294, June 2011.

Interoperability in eLearning Contexts. Interaction between LMS and PLE *

Miguel A. Conde[1], Francisco J. García-Peñalvo[1], Jordi Piguillem[2], María J. Casany[2], and Marc Alier[2]

1 **Computer Science Department. Science Education Research Institute (IUCE). GRIAL Research Group. University of Salamanca.**
Salamanca, Spain
mconde@usal.es,fgarcia@usal.es

2 **Services and Information Systems Engineering Department, UPC - Campus Nord, building Omega.**
Barcelona, Spain.
jpiguillem@essi.upc.edu,mjcasany@essi.upc.edu,marc.alier@upc.edu

Abstract

The emergence of the Information and Communication Technologies and its application in several areas with varying success, implies the definition of a great number of software systems. Such systems are implemented in very different programming languages, using distinct types of resources, etc. Learning and Teaching is one of those application areas, where there are different learning platforms, repositories, tools, types of content, etc. These systems should interoperate among them to provide better and more useful learning services to students and teachers, and to do so web services and interoperability specifications are needed. This paper presents a service-based framework approach to facilitate the interoperability between Learning Management Systems and Personal Learning Environments, which has been implemented as a proof of concept and evaluated through several pilot experiences. From such experiences it is possible to see that interoperability among the personal and institutional environments it is possible and, in this way, learners can learn independently without accessing to the institutional site and teachers have information about learning that happens in informal activities.

1998 ACM Subject Classification D.2.12 Interoperability

Keywords and phrases interoperability specifications, web services, LMS, PLE, personalization, BLTI

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.205

1 Introduction

Since its appearance, the Information and Communication Technologies (ICT) and their application in different areas have evolved very quickly. This application is more or less useful depending on the target area but in any case implies the definition of new tools, new contexts, new communication ways, etc., something that is especially remarkable with the spread of 2.0 tools. Such diversity of systems facilitates to address the necessities of institutions, users, etc.; but it also means a great deal of systems (based on different technologies) and

* This work is partially supported by the Ministry of Industry, Tourism and Trade of Spain (project IST-020302-2009-35), the Ministry of Education and Science of Spain (project TIN2010-21 695-C02) and the Government of Castilla y León through the project GR47.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 205–223

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

stakeholders that should be able to communicate with each other. In order to make this possible it is necessary to follow interoperability strategies. But, what can be understood by interoperability? European union defines it as "the capacity of ICT systems and the business processes that they support, to exchange data and knowledge" [45]. And how to achieve such interoperability? In order to do this there are different ways. Two of the most common ways to do this are Service Oriented Architectures (SOA) and Interoperability Specifications.

Service Oriented Architectures are software architectures based on creating a set of services of different granularity, between business processes and applications [5]. These architectures main aims are 1) Model the business logic as services; 2) Provide access to functionality without knowing the underlying technology; and 3) Minimize technological dependencies between the business layer and the application layer for both ones to be able to change independently [38]. The problems of these solutions are the cost to adapt the existing systems to a specific SOA and that they will not necessarily work for other configurations or distribution of the components. Proper solutions should be implemented to allow communication between different systems and this can be achieved by using interoperability specifications and standards.

Interoperability specifications define ways to exchange information and/or interaction between systems, they can be focused on the exchange of a specific type of information or address interoperability in a global way. The drawbacks of these initiatives are their acceptation and the difficulty to implement solutions based on such kind of interoperability specifications.

In the eLearning context interoperability is also necessary. The application of ICT to teaching and learning processes has had also an important influence, as the support channel to make eLearning possible [25]. Such application leads to the development of different systems as LMS (Learning Management Systems), learning tools, contents formats, etc. However ICT application did not have so much success as it was supposed to, among other reasons because of: 1) The institutional resistance to change motivated by the policy of the institutions and difficulty to integrate new systems [29, 34]; 2) The need for provide proper support to digital immigrants and the younger pupil generations that are digital natives [9, 36]; 3) The lack of connection between the formal, non-formal and informal environments makes difficult to improve learning processes and the centralization of the activity in only one context [21]; and 4) Moreover, lot of technological applications and tools are defined without taking into account the final user, which means that adopting and using them can be difficult.

In order to address these problems, learning institutions need to change their strategies. They must provide environments more adapted to the student and open to include the new set of Web 2.0 tools that are under the student's control [17]. This implies to open the existing learning environments and the definition of Personal Learning Environments (PLE), which facilitate the user learning process by allowing them to use those tools they want to use and not joining them to a specific institutional context or learning period [1].

The openness of the existing LMS, the definition of new learning environments, and the existing technological learning tool diversity make necessary to find ways in which all those systems can interoperate between them and this is what is explored along this paper, with special attention to the interoperability between LMS and PLE.

This paper proposes an approach to do so, that is, to define a possible way to facilitate the interoperability between the LMS and other external tools that could be integrated on a PLE. The approach is based on services and interoperability specifications in order it can be flexible enough to change the involved LMS, the tools or the communication ways employed.

In order to understand properly the problem, the following section describes the current

landscape of learning tools interoperability and how interoperability problems are addressed in the specific context of the LMS-PLE. After that, section 3 presents related works about how to deal with interoperability. Later (section 4) a service framework to facilitate this is posed, how it can be implemented and an example of an specific scenario a its evaluation. Finally some conclusions are provided.

2 Interoperability in eLearning Ecosystems

Since its origins eLearning implies the definition of different tools and systems, and the use of other with the aim to improve student's learning. Currently there are lot of tools employed to learn in eLearning contexts. There are different LMS i.e: Moodle (`http://www.moodle.org`), Blackboard (`http://www.blackboard.com`), Sakai (`http://sakaiproject.org`), ATutor (`http://atutor.ca/`), Desire2Learn (`http://www.desire2learn.com/`), etc.; portfolio system to gather the learning experiences i.e: Mahara (`http://www.mahara.org/`), Elgg (`http://www.elgg.com`), Desire2Learn(`http://www.desire2learn.com/`); contents and learning object repositories i.e: CAREO (`http://www.careo.org`), FREE (`http://www.free.ed.gov/`), MERLOT (`http://www.merlot.org/`); tools that can be used with learning aims i.e: Wikipedia (`http://www.wikipedia.com`), Youtube (`http://www.youtube.com`), Slideshare (`http://www.slideshare.com`), etc.

These tools are implemented with different programming languages and can be used separately or together when they are involved in learning activities.

To achieve that interoperability, learning systems and tools should be able to interoperate between them independently of the underlying technology, in order to do this previously have been described two possibilities:

- Service Oriented Architectures (SOA). They allow the interaction among different systems independently of the underlying technology. Some examples of its use in eLearning could be: 1) Use SOA to provide information from an LMS to external context, i.e: LUISA project (`http://luisa.atosorigin.es`); 2) Small adaptation of learning platforms to other applications, such as authentication services and backoffice and administrative communication tools [32]; 3) The application in interoperability specifications (such some of the described below); and 4) LMS Adaptions, that require to extract specific funcionalies from Moodle such as Moodbile (`http://www.moodbile.org`).

  The main drawbacks of these solutions in learning contexts are: the difficulty and cost of the integration of the existing systems in this kind of architectures.
- Interoperability Specifications and Standards. This implies the definition of common ways to exchange information and interaction. For example SCORM facilitates content exchange between platforms, IMS LD the Exchange of learning designs, LEAP2A to exchange user information between portfolio systems and LMS and so on. However the most difficult thing is to define standards and specifications that widen the set of possible interactions between systems. That is to say, a standard or a set of standards that facilitates interaction between systems in different senses, a simple authentication action, content transfer, information transfer, logging transfer, outcomes and so on. There are some specifications and initiatives like these and they are defined as interoperability specifications. Some of the most representative are [4, 43]:
  - Powerlinks [10]. Interoperability specification owned by WebCT and now by Blackboard. It allows discovering, launching and information provisioning of LMS services. It uses Web Services to manage users and courses, mail, calendar, tasks, notes and files. It is only available to the users of the previously mentioned platforms.

- WSRP (http://www.oasis-open.org/committees/wsrp/ – Web Services for Remote Portlets). Specification that defines an interface to represent the information provided by web services. It is not linked to a specific portlet implementation technology, such as JSR 168, but facilitates information related to how portlets can be added to portals. This specification requires a provider in the portlet container and a consumer in the portal. With these specifications the integration of applications is easier because they will be integrated into the systems not only as a service but also with a graphical representation.
- WSRP2.0 (http://www.oasis-open.org/committees/download.php/18617/wsrp-2.0-spec-pr-01.html – Web Service for Remote Portlets). Second version of the specifications that provides connections between portlets and the possibility to interact by using technologies such as AJAX or REST.
- IMS TI (http://www.imsglobal.org/ti/index.html – IMS Tools Interoperability) 1.0. Specification similar to PowerLinks. It facilitates the provisioning, launching and execution of external applications into other such as the LMS. Its implementation requires the integration of a runtime-environment into the LMS and this platform should also define deployment, configuration, execution and outcomes services for the integrated tools. On the other hand it requires the inclusion of a runtime mechanism in each of the external tools to integrate. It uses SOAP style for the implementation of Web services.
- IMS LTI (http://www.imsglobal.org/toolsinteroperability2.cfm – Learning Tools for Interoperability). It is an evolution of the previous specification, such its predecessor, provides a standard way to integrate learning tools in LMS, portals and other systems. It facilitates the launching, single-sign-on, application configuration and resources and outcomes management. In order to do this, a provider should be included in each tool and a consumer in the LMS. Its main problem is the complexity and the lack of implementations of the specification.
- IMS BLTI (http://www.imsglobal.org/lti/index.html – Basic Learning Tools for Interoperability). Reduced version of the previous specification to integrate tools in the LMS. In this case only launching and authentication services are considered. It is being extended to include outcomes. BLTI has gained greater acceptance from LMS providers. IMS Global has announced the fusion of LTI and BLTI in a new specification (http://www.imsglobal.org/lti/).
- OSIDs (Open Service Interface Definitions). Included in the Open Knowledge Initiative OKI (Open Knowledge Initiative – http://www.okiproject.org), it is a specification to define services for the integration of learning tools in SOA architectures. Such as in the other specifications a consumer and a provider are used in order to isolate the definition of services (such as authentication, course configuration, file access, etc.) from the underlying technology.

Taken into account these specifications it is necessary to think about how interoperability is addressed in the LMS/PLE context. LMS and PLE should coexist. Despite of all the benefits provided by the PLE and the shift towards the student that has happened in eLearning context, this does not mean the demise of the LMS [1]. LMS have been highly successful in stimulating online engagement of teachers and learners and, besides, they are widespread and big amounts of money have been invested on them [41]. Both systems are going to interact, tools from the LMS must be included in the PLE, and tools of the PLE can be included in the LMS; activities carried out in the PLE should be reported to the institutional environment as a way to measure the informal activity, etc.

However, it is very difficult to achieve this goal because of: 1) The problems to incorporate interoperability standards in the LMS [41]; 2) Problems of user activity traceability in the PLE and, therefore, also in the formal environment [35]; 3) Single-sign-on implementation problems [43]; and 5) Information security problems [14].

Given this situation Wilson and others proposed three possible scenarios of interoperability [52]:

- PLE and LMS exist in parallel, as informal and formal environments respectively. There are several initiatives on this sense but they are outside the scope of integration problem.
- The second scenario refers to open the LMS through the inclusion of web services and interoperability initiatives. In this scenario may be included: iGoogle based initiatives [15], social networks connected with LMS [44], the LMS that supports interoperability specifications implementation [27], PLE with specific communication protocols [47] or integration systems based on service-oriented architectures - SOA [33]. Main difficulties of these initiatives are: the institutional barriers to the opening of formal environments and the fact that those initiatives focus on information exportation and not on interaction exchange.
- The third scenario is based on the integration of external tools into the LMS. In these initiatives user might not decide which tools she is going to use and is limited to institutional decisions. Some examples of this scenario are: LMS defined for the integration of external tools [12], Google Wave Gadgets integrated into Moodle [53], PLE introducing tools based on log analysis [48], initiatives based on tools integration driven by learning design activities [19], etc. These initiatives have several problems such as: integration problems between tools, context integration difficulties, stiffness for customization by the student and so on. Those that best overcome these problems are the ones that define a learning platform starting from scratch or from a previous institutional development, however, it will greatly limit the scope of use of the solution that will be applied to very specific context.

In order to understand better the third scenario that is more focused on the use of interoperability specifications, the following section reviews the existing initiatives related with this issue in the interoperability.

3 Works related to the use of Interoperability Specifications in the LMS and PLE Contexts

The interoperability concept appears connected to the PLE since its definition in Oleg and Olivier publication [31]. In this paper the communication between institutional environments is essential and can be achieved by using specifications and standards. Despite the acceptance of the relevance of interoperability, even among authors with such different PLE perspectives as Van Harmelen [47], Wilson et al. [51], Downes [20], Schaffert y Hilzensauer [40], Wild et al. [50], this area has not been properly exploited.

This section tries to address the existing initiatives to guarantee interoperability between PLE-LMS based on specifications and standards. Before describing the interoperability specifications, some special cases should be taken in to account. In some cases interoperability among systems is achieved using specifications and standards not specifically defined with this aim; for example, in Colloquia [31] by using IMS Enterprise to add and authorize users, IMS LIP to exchange information about users, and IMS CP and SCORM to exchange content. These specifications are considering interoperability from very specific perspectives (content, user and group configuration, etc.) and in most cases take into account only information and

not interaction exchange. Other examples can be found in [8, 19, 37, 50]. However, there are other specifications more focused on interaction; representative examples are described below.

The first interoperability specification to be considered is WSRP (Web Services for Remote Portlets - `http://www.oasis-open.org/committees/wsrp/`) previously described.

Examples of WSRP applications in learning contexts can be found in the WAFFLE (Wide Area Freely Federated Learning Environment) which provides a communication bus model to define a service oriented learning architecture [11] and in the integration of a set of learning tools in the LMS Sakai [54]. The main problem of this specification is the lack of adoption of the specification by LMS developers [3, 43, 54]. Moreover, in the PLE-LMS context, WSRP provides an interface to represent portlets that can be used in the PLE but not a way to communicate with the LMS.

Other specifications such as IMS TI (IMS Tools Interoperability - `http://www.imsglobal.org/ti/index.html`) facilitate the integration of external tools in the LMS by using web services and web proxies. There are some implementations of it such as those of carried out by Wang [49] or [16], that needed to integrate an LMS (MINE LMS) and a set of collaborative tools called Learning Blog (LBlog) defining a iPLE; the Campus Project [39] uses IMS TI to launch and deploy the modules based on OKI OSIDs (this specification is discussed later); and Al-Smadi and Gutl [2] propose the definition of an online activity evaluation system based on a service-oriented architecture and IMS TI.

It is evident that very few implementations of the specification have been developed which is one of its main drawbacks, and also the difficulty to implement the specification due to its complexity [22, 12]. Given the adoption issues IMS TI, in 2008 IMS Global Learning Consortium in collaboration with eLearning and editorial companies decide to evolve TI into IMS (Learning Tools for Interoperability - `http://www.imsglobal.org/toolsinteroperability2.cfm`) and a reduced version to implement so prototypes called Simple LTI (`http://simplelti.appspot.com/`). This last solution was very popular because it provided an easy way to integrate applications so it evolved into an official subset of LTI, called Basic LTI. These two kinds of implementations have had different uptake in Learning contexts. IMS LTI has been implemented in none or very few LMS, while the majority of the LMS adopt IMS BLTI (`http://www.imsglobal.org/cc/statuschart.html`). In this situation IMS has decided to unify both proposals including BLTI features with the possibility to include outcomes and to evolve to allow the inclusion of other services. The main problem with IMS LTI is again its complex implementation which means that there are very few implementations of it. On the contrary several examples could be found related with BLTI such as Google Summer of Code implementations [42]; Campus Project [39]; use and the integration of Wordpress and Mediawiki tools in the University Oberta of Catalonia [46]; integration of tools and games in learning systems based on Interactive-TV [23]; integration of educational tools such programming problem solving [28]; Google Docs integration [7].

Another specification to take into account is that proposed by the Open Knowledge Initiative (OKI) that describes how learning components can exchange information and how to integrate with others. Examples of this specification are Campus Project (`http://www.campusproject.org`) is an example of the implementation of OKI OSIDs, developed by a set of universities and companies to define an open virtual campus in which is easy to integrate functionalities from different tools (defining a iPLE in this way) [39]; implementations based on the aggregation of tools such as content repositories, wikis, blogs, etc.[24]; and Agoravirtual, an open learning platform based on teachers' experiences and flexible enough to adapt to their specific necessities to carry out an activity [6].

This specification is a very complete one for systems defined from scratch, and it can facilitate the definition of interoperable systems, but the adaptation of existing LMS and tools to it is very complex, due to the great quantity of services to consider, that is the reason for so few implementations of it. Given this context and taking into account the advantages and drawbacks of these specifications an approach to facilitate the interoperability between LMS and PLE is presented.

4 A Service-Based Framework Solution

As metioned above, there is a need of new learning environments focused on the student, and the PLE is the best representative of those environments. They should interact with the institutional learning environments in such a way that institutional functionalities can be included into the PLE, and the activity carried out in PLEs integrated in the LMS. In addition it is also needed that the existing system could be used used and not only solutions defined from scratch.

But the definition of a complete, scalable, flexible and portable approach which satisfies the students' needs and institutional requirements, while allowing to reuse the existing learning platforms, is a very difficult goal to achieve. It requires of interoperability between the LMS and the PLE. This interoperability facilitates students the definition of their own PLE in a seamless way, so that they only need to access the LMS for a minimum set of indispensable activities. Moreover, interoperability also gives teachers more information about what the students do in the external environments and give them a more broad set of tools for the proposal of learning activities. All these tools may heavily contribute to the evolution of the LMS.

To achieve this, second and third Wilson's interoperability scenarios are mixed in the approach. That is, the exportation of functionalities is considered, specially by using the web service layers that most LMS provides and the use interoperability specifications to integrate what is happening outside the institution.

4.1 Components of the Service-Based Framework Approach

When talking about the definition of a service-based framework in this proposal, it is clear that its main goal is to facilitate the communication and interaction between the institutional (represented by one or several LMS) and personal learning environments (a specific PLE). That communication will be based on the use of services and standards so as to guarantee the independence of the solution from the underlying technology (that means independence of the different LMS, PLEs or online tools), the scalability (it should be easy to add other tools or LMSs) and the portability of the approach (to other contexts such as mobile devices).

The proposal consists of three main elements: the institutional context, the personalized context and the communication channels. Besides, some other elements, such as mediator elements (to facilitate the communication between specific instances of the LMS and the online tools included into the PLE) and/or the representation of these elements in other contexts (such as mobile devices), may be used. These elements can be seen in Figure 1.

The institutional contexts can include one or several different LMS in which the student performs her academic activities. This element represents the institutional learning environments that the student uses, focused mostly in the course and not in the user. The institutional context can be represented as in Figure 1 as one or several nodes with instances of different LMS. In order to make possible the interoperability between learning tools and such learning environments it is necessary that each LMS implements: a web service interface

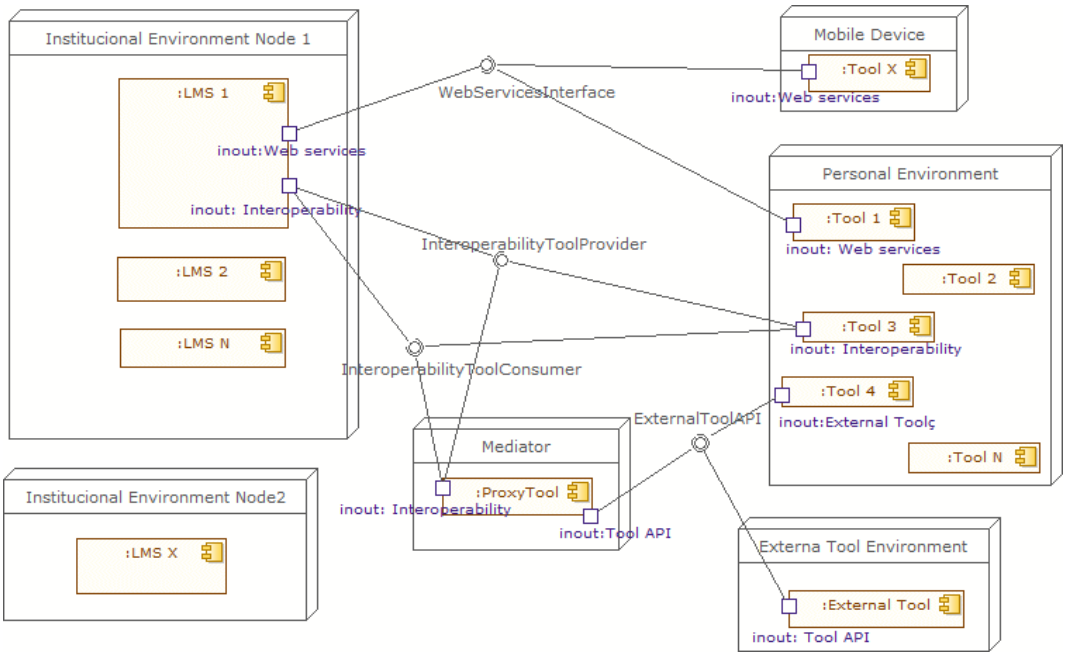


Figure 1 Deployment diagram of the reference framework approach.

to facilitate the access to the learning platform information and functionality and an interoperability interface to consume external tools (InteroperabilityToolconsumer). Moreover these learning platforms require an interoperability interface to gather information from the external tool (InteroperabilityToolProvider).

On the other hand there is a personalized environment focused on the learner which facilitates informal learning. It should allow the learner to add all kind of tools she uses to learn, including institutional tools. In order to do this, each tool should be able to work independently but into a context that acts as a container. These tools, as can be seen in the Figure 1, could have no interaction with the PLE (Tool2), use web services (Tool 1) or use interoperability specifications to communicate with the learning environment (Tool 3 and 4), sometimes they are based on external tools and require additional interfaces and intermediate components to facilitate the communication with the LMS (ExternalTool). Moreover these tools could be included in other contexts such as mobile devices (Tool X).

The other important element in the framework is the one related to communication channels. Communication channels should provide standard and independent ways to exchange in a bi-directional way (from the LMS to the PLE and from the PLE to the LMS) information and interaction. There are three kind of interfaces (Figure 1).

- Web service interface. It allows the communication between the external tools and the LMS independently of the underlying technology. These interfaces are implemented by the LMS and provide a way to access to the LMS information and functionality. They are specific for each LMS so the solutions should be adapted. Anyway some basic services should be included such as authentication, users managements, courses management, activity management and resources management.
- Interoperability interfaces. They are defined as ways to establish information and interaction channels between the LMS and the tools included into the PLE. As most of the interoperability specifications it implies the definition of a ToolConsumer (TC) in the

LMS and a ToolProvider (TP) in each Tool. The TC implements an InteroperabilityToolConsumer interface used to launch, instantiate and set up activities from the LMS based on external tools. The TP implements an InteroperabilityToolConsumer interface that is used by the LMS in order to recover information about the Tool and the activities carried out into it. In this way the teacher can define activities based on such tools in the LMS, that are going to be performed by students in the PLE, and which outcomes can be returned to the LMS to be taken into account during students' evaluation.

- External tool interfaces. In the PLE can be represented not only customized tools but also other external well-known tools which may be used in learning activities, such as GoogleDocs, Slideshare and so on. Such tools are not accessible to be installed into a PLE but they provide APIs to access to their functionality. These APIs are used by PLE Tools and also by intermediate tools.

It is also possible that the framework includes mediator elements to perform activities related to the adaptation of the transferred functionality and information. They are mainly used to facilitate the integration of proprietary and/or not educational tools. For example, the mediator may interact with a proprietary tool, which cannot be adapted to the framework and/or provide evaluation interfaces, to help to use in learning contexts.

With these components and interfaces it is possible to define solutions that not only export information from the LMS but also funcionality, and that allow monitoring the learning activities activities that are carried out by the user in the PLE. But to provide a real interoperability between these components by using the interfaces some scenarios can be considered. They are described in the following section.

4.2 Interoperability Scenarios

Given the previous architectural approach a set of interoperability scenarios can be defined. They present possible ways to facilitate the exportation of functionalities outside the LMS and the integration into the LMS of students' activities performed in other tools. The idea of these scenarios is to enable the student to learn not only in the LMS but also in her personalized environment and to enable the teacher to work in the institutional one. Both contexts should communicate between them. This communication is distributed in four possible scenarios:

- Scenario 1 - Exportation of institutional functionalities to personalized environments. This scenario aims to the export of functionalities from a LMS to other environments controlled by the user. In order to export that functionality, the LMS web service layer is used. In that scenario the tool connects with the learning platform by using the web services to access the functionality. This means that the student may use a functionality from the LMS in the PLE. The teacher can also follow the student activity as if she was answering from the LMS, so she can be also assessed. Thus, teachers and students use their respective environments while having knowledge about what is happening in the other context. The scenario is open to include other tools and to export the functionality to other contexts different than the PLE.
- Scenario 2 - Taking into account the use of external learning tools from the institutional environment. In this scenario no interoperability between the LMS and the PLE is proposed. It takes into consideration the students' activity into the PLE from the institutional environment but such activity should be assessed by the teacher who would access to the context in which is the tool used by the student in order to check the activity that she has carried out. For example, a student accesses an online tool from the PLE, and performs (in agreement with the teacher) a task by using it; then, the teacher should

enter into the online tool or the PLE, check her activity and perform her assessment from the LMS. This scenario is quite common in different institutions and it requires an extra effort from the teacher.

- Scenario 3 - Use of external online educational tools (with evaluation support) in the PLE, and recover information from LMS. In this scenario the activity is done in the external educational tool but it is integrated in the LMS. The teacher defines an instance of the educational tool into the LMS: this will create a context only accessible by teachers and through which they can see the results of the task completed by the student; the student accesses her personalized environment and can use, among others, the educational tool adapted to return information about the student's activity to the LMS. The tool should be able to assess the activity carried out by the student or provide the interfaces needed by the teachers to do that (i.e. a tool to carry out quizzes, a simulator, a serious game, etc.). The interoperability described in this scenario is based on the use of Interoperability specifications, so it will need the TC, TP and interoperability interfaces described in the previous sections. The application of the specification minimizes teachers' effort in order to check the activity outside the LMS (because they do not need to access to other environments to check it).
- Scenario 4 - Use of external online tools (not defined as educational and thus without an evaluation interface) in the PLE and recover the information from the LMS. This scenario aims to gather the students' activity in online tools included in the PLE. Those tools are not necessarily educational tools so they are not going to provide an interface to assess the students' outcomes. The teacher defines an instance of the online tool into the LMS; this will create a context that only teachers can access and through which the results of the activity performed by the student can be returned or the evaluation could be facilitated. The student accesses to her personalized environment and can use, among others, the online tools adapted to return information about the student activity to the LMS. The tool in this case is not necessarily created with a learning objective, so they do not include assessment interfaces, something that is needed to grade the student's activity. This assessment interface will be provided by the mediator (or proxy tool), which interacts with the online tool and with the LMS. The implementation of the scenario also requires the use of Interoperability specifications in order to return the activity from the PLE to the LMS. As in the previous scenario, this involves including a TC in the LMS and a TP in the Tool, the TP could be included in the mediator because in many of those tools it is not possible to have access to the code and introduce the TP (in example in Google Docs).

4.3 Example of Interoperability Implementation by using IMS BLTI

The components and interoperability scenarios have been implemented has a proof of concept and later evaluated in education environments (in the University of Salamanca subjects). In order to carry out such implementation the LMS is represented by Moodle, due to its widespread (`http://moodle.org/stats/`) and because it includes an open web service layer [18] that is used in the interoperability scenario 1. In addition to this LMS any other which provide a web service layer could be employed. The tools of the PLE are represented following W3C widget recommendation (`http://www.w3.org/TR/widgets/`) and the PLE by using Apache Wookie (Incubating) [52], this is because, by following this specification it is possible to facilitate the portability of the solution to other environments and contexts. As interoperability specification it was decided to use is IMS BLTI, this is because of its uptake in the LMS context (`http://www.imsglobal.org/cc/statuschart.cfm`) and because despite

of the few quantity of services it provides, it is possible to use different extensions as outcomes and memberships (which facilitates to gather the grades achieved in a tool and the roster of students involved in an activity) that make able the implementation of the interoperability scenarios. Other reason to use BLTI is the existence of a BLTI Tool Consumer integrated with Moodle as a module (`http://code.google.com/p/basiclti4moodle/`), so it is not necessary to define one from the scratch. It is also necessary to consider that the scenarios need to include different adapted tools to be represented as widgets into the PLE. During the proof of concept these tools are: Moodle Forum (in scenario1), Wordpress and Flickr adaptations (in scenario 2), an ad-hoc quiz tool as an educational external tool (in scenario 3) and GoogleDocs as a collaboration external tool (in scenario 4).

In order to illustrate the use of the interoperability approach the Scenario 3 is described. In such scenario there are two roles involved, teachers and students and it requires the use of several components.

One of the components is Moodle and the integrated BLTI ToolConsumer that uses an interface to configure and launch the external tool (the tool provider interface) and implements an interface called ToolConsumerInterface that is going to be used to return information to the LMS. The BLTI Tool provider should be configured in Moodle to link an external tool and after that it will be possible to create learning activity instances based on such tool in the LMS.

The other component to consider is the external tool, that in this scenario should be an educational tool. This means that the tool should be able to evaluate students' activity or provide an evaluation interface (i.e: a simulator, a case tool, a quiz tool, etc.). A quiz tool has been created for the proof of concept. It allows the teacher to define a self-evaluation quizzes, that students can carry out in a web environment or in a widget integrated into the PLE. The quiz tool includes a BLTI ToolProvider that facilitates the communication with the LMS, implementing a ToolProviderInterface and using a the ToolConsumerInterface that provides Moodle.

Given this context the teacher can enter into her Moodle course, create an activity based on the quiz tool and launch it. The launching implies the definition of a quiz in the external tool that would be available for the students of the course. In order to do so it is necessary the BLTILaunching service to set up the activity, and the Memberships BLTI extension to recover the id of al the students that participates on the course; once created the activity and associated the learners the view of the quiz is return to the ToolConsumer (Figure 2). The students could carry out the activity in the widget included in the PLE or in the quiz tool web environment. When the activity was finished the teacher is able to recover the grades achieved by the students in the external tool from the LMS, to do so Outcomes BLTI extension is employed.

This is not the traditional way to use BLTI, which use to integrate the tool into the LMS and there is used by teachers and students. In this case the teacher can access to an activity view in Moodle from which she can recover the students' grades, however the students do not need to carry out the activity through the institutional environment and can do it in the PLE (Figure 3). In this way an interoperability channel has been established that allows the teacher to control what is happening in an activity included in the PLE and the learner can combine an institutional activity with other tools she uses to learn.

4.4 Interoperability Scenarios Validation

The interoperability scenarios has been evaluated through several pilot experiences carried out with students of the University of Salamanca and a set of semi-structured interviews

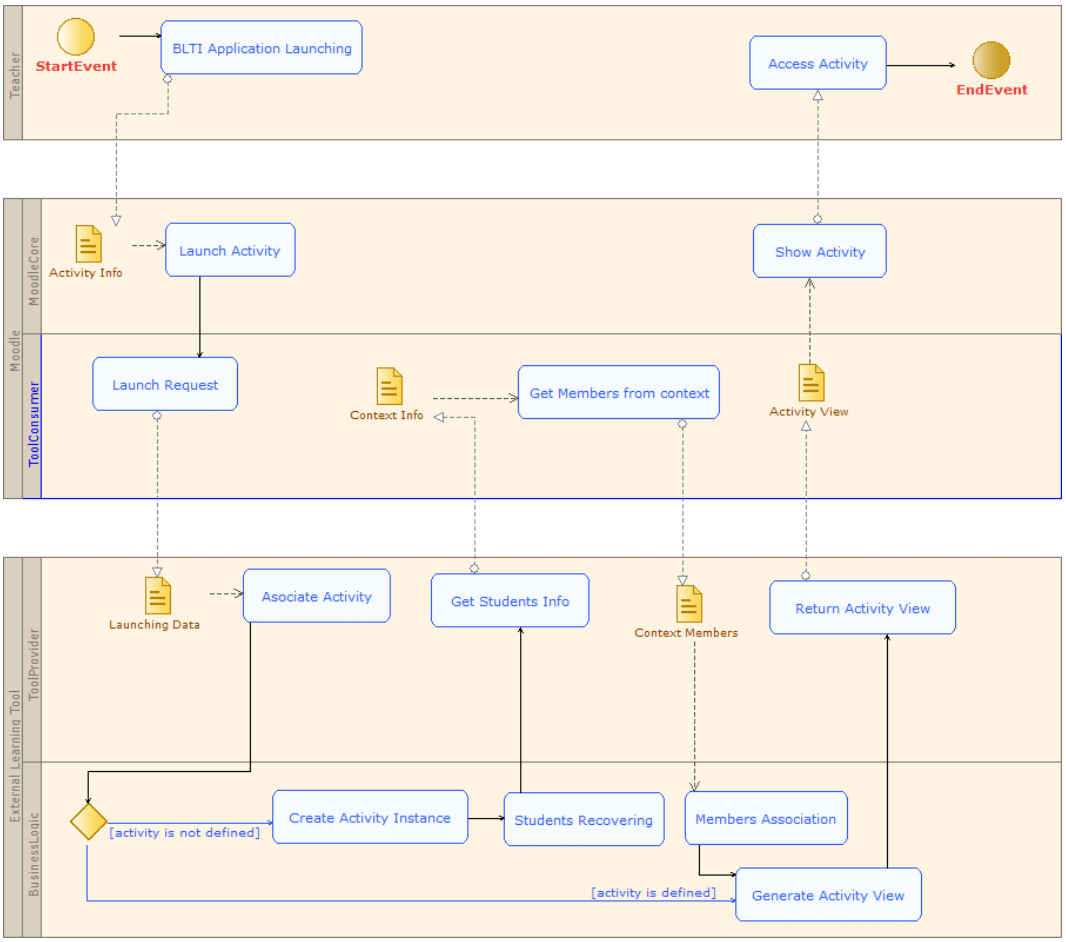


Figure 2 BPMN diagram to define the launching activity of an external tool from a LMS. The figure show the different participants involved in this action and the messages exchanged between them.

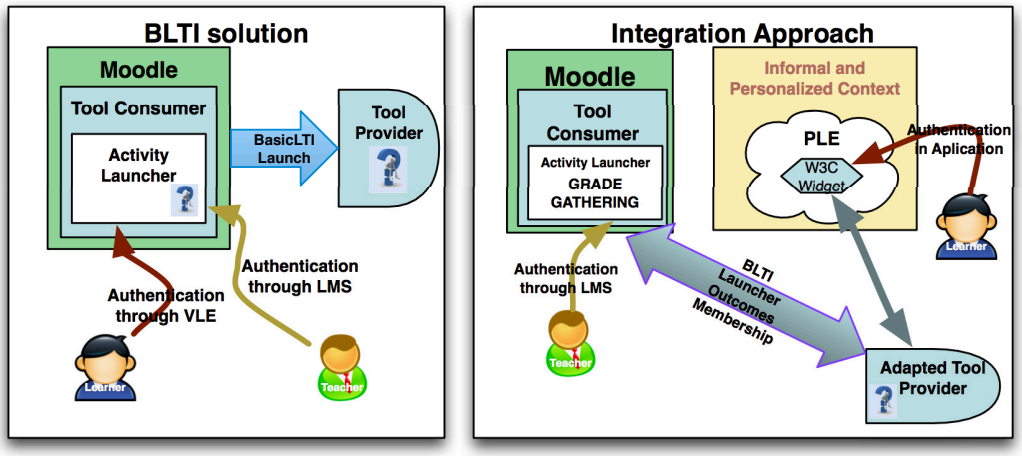


Figure 3 BLTI approach Vs. Integration approach. On the left side, quiz tool integration launched inside the LMS. On the right side, quiz tool integrated in the PLE and accessed by the learner from it and the BLTI extensions used.

with teachers. In order to analyze the data a mixed methodology is used [26], which means that quantitative and qualitative techniques are employed. In this paper the results of the quantitative methodology application in scenario 3 are shown as an example of the evaluation carried out. Firstly the methodology is presented and after that the results and discussion.

4.4.1 Methodology

During the experiment are involved 50 students of the subject Project Management. With them students it has been applied a quantitative methodology and with the teachers a qualitative one endorsed with quantitative results.

The methodology used with the students is a quasi-experimental design [13]. It is used because in this experiment pre-established groups of students (class-groups) are used, so it is not possible to have a complete randomized group of people [30] and therefore neither is possible a control study approach. Quasi-experimental design implies the definition of a scientific hypothesis, from which a dependent variable is derived. Such variable is operationalized through several assertions that are proposed to the students of the experimental and control group (independent variable). These assert are graded by the students using a five-value levels scale (1=strongly disagree, 2=disagree, 3=indifferent, 4=agree, 5=strongly agree). In both groups the same tests are applied, a pre-test at the beginning of the experiment and a post-test after it, but the students of the experimental group test the quiz widget in the PLE, while the people in the other group do not. After running the experiment, data is analysed by using probabilistic techniques to validate the initial hypothesis.

The scientific hypothesis is going to be accepted if the results of the pre-test are similar in both groups (which proves that both groups are similar and have a common knowledge and background) and the results of the post-test between the people involved in the experimental group and the control group are different (those who have test the tool should answer in a different way). This has been checked using two statistic tests Student's T test and Mann Whitney's U test. This last one is applied because with a sample of 50 students the test is near to the limit in which it can be applied in a robust way and also because the data to consider is ordinal. With this statistic test is proposed the null hypothesis for the Student's T is $H_0 : \overline{\mu_E} = \overline{\mu_C}$ (where X refers to the average range, E refers to the experimental group and C refers to the control group), which compare the average grade of each item between the control and the experimental group. In Mann Whiney's U what is checked is the difference of ranges through the following null hypothesis $H_0 : \overline{R_E} = \overline{R_C}$ (where R refers to the average range, E refers to the experimental group and C refers to the control group).

4.4.2 Data Analysis and Discussion

During the quantitative study of students' opinion the scientific hypothesis was "The inclusion of the activity carried out by the student in external educational tools into the LMS, improves her learning, the knowledge the institution have about her and facilitates her evaluation". To test this hypothesis, some assertions have been proposed to the students.

- In the pre-test:
  - I1. Moodle provides a great variety of tools to use in the subjects and no more are needed.
  - I2. I use other online educational tools than those provided by Moodle to learn (such as simulators, resources libraries, external quizzes, etc.).

- In the post-test:
  - I3. The fact that Moodle does not facilitate the introduction of my activity in other external tools (such as simulators, resources libraries, self-evaluation tests, etc.) supposes that it does not satisfy properly my learning needs.
  - I4. The fact that Moodle does not facilitate the introduction of my activity in other external tools (such as simulators, resources libraries, self-evaluation tests, etc.) supposes that I was just partially evaluated.

The results of the Student's T test can be seen in the Table 1, with a signification of a 0.05. If the signification of the item is under 0.05 the null hypothesis is accepted, if not it is rejected.

Table 1 Results of the Student's T-test. The table shows the medium and variance for each item of the pre-test and post-test, the result and the bilateral significance.

Pre-test results for Student's T test						
VD	$\overline{X_E}$	S_E	$\overline{X_C}$	S_C	t	ρ
I.1	2.40	0.968	2.15	0.587	1.033	0.307
I.2	3.90	1.348	3.55	1.234	0.930	0.357
Post-test results for Student's T test						
I.3	3.57	0.817	2.40	0.821	4.937	0.000
I.4	3.47	0.973	2.55	0.823	3.461	0.001

In the table is shown that in both pre-test items the null hypothesis is retain (that is to say the experimental and control group answer more or less the same) and in the post-test the null hypothesis is rejected (so the results between the experimental and control group are different). From the pretest data it can be also concluded that, in the student perception, Moodle needs more tools than those included and students use more tools that those provided by Moodle to learn. From the pretest it can be seen that the students who has experimented other tools integrated into Moodle consider that this platform does not satisfy their learning needs and that in their opinion they are not properly evaluated. The difference in the pre-test and pos-test between the experimental and control groups in every assertion means that the scientific hypothesis should be accepted, but in order to check it, Mann Whitney U test is also applied (Table 2). In this table is shown that for the pre-test assertions (I1 and I2) the differences in ranges are minimum, so null hytothesis is retained for each of them; whilst after the experiment (in the postest) the difference in the assertions (I3 and I4) is significative so the null hypothesis is rejected. This endorse the results of the Student's T test.

Table 2 Results of the Mann-Whitney U test. It shows the average range for experimental and control groups, the result of contrast statistic and the significance per each item.

Pre-test results for Mann-Whitney U test					
VD	$\overline{R_E}$	$\overline{R_C}$	U	Significance	Result
I.1	26.77	23.60	262.0	0.412	Retain null hypothesis
I.2	27,63	22,30	236.0	0.186	Retain null hypothesis
Post-test results for Mann-Whitney U test					
I.3	31.97	15.80	106.0	0.000	Reject null hypothesis
I.4	30.28	18.33	156.5	0.003	Reject null hypothesis

To support these conclusions an assertion about the experience was posed to the students of the experimental group. This assertion is: "The activity I carried out in external online tools should be integrated into the LMS because it would enrich my learning". The 80% of the experimental group students agrees or strongly agrees with the assertion, which means that in their opinion the inclusion of external activity into the institution open new ways to enrich learning with additional tools.

In order to consider also the teachers' opinion several semi-structured interviews have been carried out. On them, the system is presented to the teachers, they used it, and their opinion is recovered. The results are: 1) The 100% of the teachers agree or strongly agree that by including students' activity on external tools it is possible to assess them in more comprehensive way; and 2) The 90% consider that this evaluation can be easier for them if the students' activity outcomes are directly integrated into the LMS so they do not need to check these results in external environments.

5 Conclusions

Along this paper is shown that interoperability is one of the key factors to define systems based on different tools, architectures and technologies. This kind of interoperability is much more necessary today because the users require specific services and are not too much interested in downloading, install and use a set of different systems that are not customized to their necessities. In order to achieve this, it is necessary to join different applications to provide a service that satisfy user's specific needs. In the eLearning landscape the situation is similar to this. There are several LMS, different portfolio systems, repositories, contents and so on. These elements are developed in different programming languages, use different contexts and most of the times are not able to talk with other systems to compose learning services more adapted to real student necessities.

On the other hand there is a shift in the "locus of control" in learning environments, the student needs spaces more adapted to her necessities, because she does not learn only in the institutional context but also along her life and by using different tools from different systems in different context related or not with different institutions. PLEs are defined to address this problem, as a concept that provides support to all those tools. However a PLE is another element that is not communicated with the existing tools such as the LMS so interoperability specifications and standards need to be applied to open real interaction channels valid for a changeable context. This interoperability application is difficult because of the variety of interoperability specifications; the difficulty to implement them and that it requires changing the LMS; and also the tools that need to interoperate.

Given this context a service-based framework approach has been posed. Such framework takes into account the LMS, the PLE and a set of interfaces that facilitates the interoperation between them. In addition it includes the most common interoperability scenarios that can be employed. In order to validate the framework it has been implemented as a proof of concept and the scenarios have been evaluated. In this paper one of these scenarios and its evaluation has been described, in this case the integration in the LMS of students' results carried out in an educational external tool included in the PLE. From that experience, and in the opinion of the students and teachers involved in the experience, it can be seen that the students use other tools than those included by Moodle to learn and that it is necessary to take what they do into them into account, because in this way the students will be more motivated to learn and teachers will have more knowledge about students' skills. An evaluation similar to this has been done for each of the other scenarios and also

qualitative techniques has been applied to exploit some of the information gathered with the semi-structured interviews with teachers. With the previous framework and the mentioned experiences it is possible to conclude that interoperability between LMS and PLE is possible, and that informal learning can be taken into account from the formal environments, while informal learning can be enriched with functionalities of the institutional contexts as well. In addition, this interoperability facilitates students the definition of their own PLE in a seamless way, so that they only need to access the LMS for a minimum set of indispensable activities. Moreover, interoperability also gives teachers more information about what the students do in the external environments and give them a more broad set of tools for the proposal of learning activities. All these tools may heavily contribute to the evolution of the LMS.

As a future work it would be possible to integrate other scenarios to our proposal, other contexts, to collaborate in the definition of the new IMS LTI (taking into account the experiences carried out) and define pilot experiences with other students.

References

1 Jordi. Adell and Linda. Castañeda. Los entornos personales de aprendizaje (ples): una nueva manera de entender el aprendizaje. In R. Roig Vila and M Fiorucci, editors, *Claves para la investigación en innovación y calidad educativas. La integración de las Tecnologías de la Información y la Comunicación y la Interculturalidad en las aulas. Stumenti di ricerca per l'innovazioni e la qualità in ámbito educativo. La Tecnologie dell'informazione e della Comunicaziones e l'interculturalità nella scuola.* Marfil – Roma TRE Universita degli studi, Alcoy, Spain, 2010.

2 Mohammad Al-Smadi and Christian Gütl. Soa-based architecture for a generic and flexible e-assessment system. In Manuel A. Castro, Edmundo Tovar, Michael E. Auer, and Manuel P. Blázquez, editors, *IEEE EDUCON 2010 - Education Engineering - The Futute of Global Learning Engineering Education*, Series SOA-based Architecture for a Generic and Flexible E-assessment System, pages 493–500, Madrid, Spain, 2010.

3 Carlos Alario-Hoyos, Juan I. Asensio-Pérez, Miguel L. Bote-Lorenzo, Eduardo Gómez-Sánchez, Guillermo Vega-Gorgojo, and Adolfo Ruiz-Calleja. Integration of external tools in virtual learning environments: Main design issues and alternatives. In M. Jemni, Kinshuk, D. Sampson, and JM Spector, editors, *10th IEEE International Conference on Advanced Learning Technologies, ICALT2010*, Series Integration of External Tools in Virtual Learning Environments: Main Design Issues and Alternatives, pages 384–388, Sousse, Tunisia, 2010.

4 Carlos Alario-Hoyos and Scott Wilson. Comparison of the main alternatives to the integration of external tools in different platforms. In *International Conference of Education, Research and Innovation, ICERI 2010*, Series Comparison of the main Alternatives to the Integration of External Tools in different Platforms, pages 3466–3476, Madrid, Spain, November, 2010.

5 Julio Alba. ¿qué es soa - arquitectura orientada al servicio. *Bit*, 167:52–53, 2008.

6 José Alfonso. Ágora virtual: una propuesta de entorno colaborativo y de enseñanza sobre interfaces osid. *RedIRIS: boletín de la Red Nacional de I+D RedIRIS,*, (76):21–32, 2006.

7 Marc Alier, Maria J. Casañ, Jordi Piguillem, Nikolas Galanis, Enric Mayol, Miguel A. Conde, and Franciso Garcia. Integration of google docs as a collaborative activity within the lms using ims basiclti. In *Knowledge Management, Information Systems, E-Learning, and Sustainability Research. Fourth World Summit on the Knowledge Society, WSKS 2011*, volume CCIS 0278 of *Communications in Computer and Information Science series*, Mykonos, Greece, 2011. Springer-Verlag.

8 Juan I. Asensio-Pérez, Miguel L. Bote-Lorenzo, Guillermo Vega-Gorgojo, Yannis A. Dimitriadis, Eduardo Gómez-Sánchez, and Eloy D. Villasclaras-Fernández. Adding mash-up based tailorability to vles for scripted collaborative learning. In F. Wild, M. Kalz, and M. Palmér, editors, *Mash-Up Personal Learning Environments - 1st Workshop MUPPLE'08*, volume 388 of *Series Adding mash-up based tailorability to VLEs for scripted Collaborative Learning.*, pages 14–18, Maastricht, The Netherlands, 2008. CEUR-Workshop Proceedings.

9 S. Bennett, K. Maton, and L. Kervin. The 'digital natives' debate: A critical review of the evidence. *British Journal of Educational Technology*, 39(5):775–786, 2008.

10 Blackboard. Blackboard powerlinks kit for software development. Technical report, Blackboard Learning System CE and Vista Enterprise License, 25/08/2011 2008.

11 Andrew G. Booth and Brian P. Clark. The waffle bus: A model for a service oriented learning architecture. In D. Whitelock and S. Wheeler, editors, *13th Association for Learning Technology Conference (ALT-C 2006).*, Series The WAFFLE Bus: A model for a service oriented learning architecture, pages 172–182, Scotland, UK., 2006.

12 Andrew G. Booth and Brian P. Clark. A service-oriented virtual learning environment. *On the Horizon.*, 17(3):232–244, 2009.

13 D.T. Campbell and J.C. Stanley. *Experimental and quasi-experimental designs for research.* Rand McNally, 1963.

14 Oskar Casquero, Javier Portillo, Ramón Ovelar, Manuel Benito, and Jesús Romo. iple network: an integrated elearning 2.0 architecture from university's perspective. *Interactive Learning Environments*, 18(3):293–308, 2010.

15 Oskar Casquero, Javier Portillo, Ramón Ovelar, Jesús Romo, and Manuel Benito. igoogle and gadgets as a platform for integrating institutional and external services. In F. Wild, M. Kalz, and M. Palmér, editors, *Mash-Up Personal Learning Environments - 1st Workshop MUPPLE'08*, volume 388 of *Series iGoogle and gadgets as a platform for integrating institutional and external services*, pages 37–42, Maastricht, The Netherlands, 2008. CEUR-Workshop Proceedings.

16 Jui-Hung Chen, Timothy K. Shih, Chun-Chia Wang, Shu-Wei Yeh, and Chen-Yu Lee. Combine personal blog functionalities with lms using tools interoperability architecture. In *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications*, Series Combine Personal Blog Functionalities with LMS Using Tools Interoperability Architecture, pages 146–151, Okinawa, Japan, 2008. IEEE Computer Society.

17 W. Clark, K. Logan, R. Luckin, A. Mee, and M. Oliver. Beyond web 2.0: mapping the technology landscapes of young learners. *Journal of Computer Assisted Learning*, 25(1):56–69, 2009.

18 Miguel Á. Conde, A. Pozo, and F. J. García-Peñalvo. e-learning services in moodle 2.0. *CEPIS Upgrade.*, 12(2), 2011.

19 Luis de-la Fuente-Valentín, Derick Leony, Abelardo Pardo, and Carlos Delgado Kloos. Mashups in learning design: pushing the flexibility envelope. In *Mash-Up Personal Learning Environments - 1st Workshop MUPPLE'08*, Series Mashups in Learning Design: pushing the flexibility envelope, pages 18–24, Maastricht, The Netherlands, 2008.

20 Stephen Downes. E-learning 2.0. *Elearn magazine*, 2005(10):1, 2005.

21 Stephen Downes. New technology supporting informal learning. *Journal of Emerging Technologies in Web Intelligence*, 2(1):27–33, 2010.

22 Jorge Fontenla, Manuel Caeiro, and Martín Llamas. Una arquitectura soa para sistemas de e-learning a través de la integración de web services. In *Congreso Iberoamericano de Telemática. CITA 2009*, Series Una Arquitectura SOA para sistemas de e-Learning a través de la integración de Web Services, pages 22–29, Gijón, Spain, 2009.

23 Jorge Fontenla, R. Perez, and M. Caeiro. Using ims basic lti to integrate games in lmss — lessons from game×tel. In *IEEE EDUCON 2011 - Education Engineering - The Futute of*

Global Learning Engineering Education, Series Using IMS basic LTI to integrate games in LMSs — Lessons from Game×Tel, pages 299–306, Amman, Jordan, 2011. IEEE.

24 Adam Franc. Outside-in: Application interoperability using an osid-based framework. In *OpeniWorld:Europe2008 - Federating resources through open interoperability*, Series Outside-in: Application interoperability using an OSID-based framework, Lyon, France, 2008.

25 Francisco José García-Peñalvo. *Preface of Advances in E-Learning: Experiences and Methodologies*. Information Science Reference (formerly Idea Group Reference). Information Science Reference, Hershey, PA, USA, 2008.

26 Judith L. Green, Gregory Camilli, and Patricia B. Elmore. *Handbook of Complementary Methods in Education Research*. American Educational Research Association by Lawrence Erlbaum Associates, Inc, 2006.

27 IMS-GLC. Common cartridge and basic learning tools interoperability progress and conformance status, 2011.

28 José P. Leal and Ricardo Queirós. Integrating the lms in service oriented elearning systems. *International Journal of Knowledge Society Research (IJKSR)*, 2(2):1–14, 2011.

29 Jon Mott and David Wiley. Open for learning: The cms and the open learning network. *In Education - Exploring our connective educational landscape*, 15(2), 2009.

30 Santiago Nieto and Adriana Necamán. Investigación y conocimiento científico en educación. In Santiago Nieto and María J. Rodriguez-Conde, editors, *Investigación y Evaluación Educativa en la sociedad del conocimiento*. Ediciones Universidad de Salamanca, Salamanca, 2010.

31 Bill Olivier and Oleg Liber. Lifelong learning: The need for portable personal learning environments and supporting interoperability standards. Technical report, The JISC Centre for Educational Technology Interoperability Standards, Bolton Institute., 2001.

32 S. Pätzold, S Rathmayer, and S Graf. Proposal for the design and implementation of a modern system architecture and integration infrastructure in context of e-learning and exchange of relevant data, 2008.

33 Yvan Peret, Sabine Leroy, and Eric Leprêtre. First steps in the integration of institutional and personal learning environments. In *Workshop Future Learning Landscape - EC-TEL 2010*, Series First steps in the integration of institutional and personal learning environments, pages 1–5, Barcelona, Spain, 2010.

34 Alejandro Piscitelli, Iván Adaime, and Inés Binder. *El proyecto facebook y la posuniversidad. Sistemas operativos sociales y entornos abiertos de aprendizaje*. Fundación Telefónica. Editorial Ariel, S.A., Barcelona, 2010.

35 Hans Põldoja and Mart Laanpere. Conceptual design of edufeedr — an educationally enhanced mash-up tool for agora courses. In Fridolin Wild, Marco Kalz, Matthias Palmér, and D. Müller, editors, *Mash-Up Personal Learning Environments - 2nd Workshop MUPPLE'09*, volume 506 of *Series Conceptual Design of EduFeedr — an Educationally Enhanced Mash-up Tool for Agora Courses*, pages 98–102, Nize France, 2009. CEUR Proceedings.

36 M Prensky. Digital natives, digital immigrants. *On the Horizon*, 9(5), 2001.

37 Ricardo Queirós, Lino Oliveira, José Paulo Leal, and Fernando Moreira. Integration of eportfolios in learning management systems. In *International Conference Computational Science and Its Applications - ICCSA 2011*, volume 6786/2011 of *Series Integration of ePortfolios in Learning Management Systems*, pages 500–510, Santander, Spain, 2011. Springer Verlag.

38 Michael Rosen, Boris Lublinsky, Kevin T. Smith, and Marc J. Balcer. *Applied SOA: service-oriented architecture and design strategies*. Wiley Pub., 2008.

39 Francesc Santanach, Jordi Casamajó, Pablo Casado, and Marc Alier. Proyecto campus. una plataforma de integración. In *IV Simposio Pluridisciplinar sobre Diseño, Evaluación y*

Desarrollo de Contenidos Educativos Reutilizables. SPDECE 07, Series Proyecto CAMPUS. Una plataforma de integración, Bilbao, Spain, 2007.

40 Ra Schaffert and Wolf Hilzensauer. On the way towards personal learning environments: Seven crucial aspects. *eLearning papers*, 2(9):1–11, 2008.

41 Niall Sclater. Web 2.0, personal learning environments, and the future of learning management systems. *Research Bulletin*, (13), 2008.

42 Charles Severance, Ted Hanss, and Josepth Hardin. Ims learning tools interoperability: Enabling a mash-up approach to teaching and learning tools. *Technology, Instruction, Cognition and Learning*, 7(3-4):245–262, 2010.

43 Charles Severance, Joseph Hardin, and Anthony Whyte. The coming functionality mash-up in personal learning environments. *Interactive Learning Environments*, 16(1):47–62, 2008.

44 Ricardo Torres, Palitha Edirisingha, and Richard Mobbs. Building web 2.0-based personal learning environments: A conceptual framework. In *EDEN Research Workshop 2008*, Series Building Web 2.0-Based Personal Learning Environments: A Conceptual Framework, Paris, France, 2008.

45 UE. Decision 2004/387/ec of the european parliament and of the council of 21 april 2004 on the interoperable delivery of pan-european egovernment services to public administrations, businesses and citizens (idabc), 2004.

46 UOC. Uoc at ims learning impact 2010. Technical report, Inversitat Oberta de Catalunya, 2010.

47 M. van Harmelen. Personal learning environments. In *Proceedings of the Sixth IEEE International Conference on Advanced Learning Technologies*, Series Personal Learning Environments, pages 815–816, Kerkrade, The Netherlands, 2006. IEEE Computer Society.

48 Dominique Verpoorten, Christian Glahn, Milos Kravcik, Stefaan Ternier, and Marcus Specht. Personalisation of learning in virtual learning environments. In *Proceedings of the 4th European Conference on Technology Enhanced Learning: Learning in the Synergy of Multiple Disciplines*, Series Personalisation of Learning in Virtual Learning Environments, pages 52–66, Nice, France, 2009. Springer-Verlag.

49 Chun-Chia Wang. The development of collaborative learning environment with learning blogs. *Journal of Software*, 4(2), 2009.

50 F. Wild, F. Mödritscher, and S.E. Sigurdarson. Designing for change: Mash-up personal learning environments. *eLearning Papers*, 9:1–15, 2008.

51 S. Wilson, O. Liber, M. Johnson, P. Beauvoir, P. Sharples, and C. Milligan. Personal learning environments: Challenging the dominant design of educational systems. *Journal of e-Learning and Knowledge Society*, 3(3):27–38, 2007.

52 Scott Wilson, Paul Sharples, and Dai Griffiths. Distributing education services to personal and institutional systems using widgets. In Fridolin Wild, Marco Kalz, and Matthias Palmér, editors, *Mash-Up Personal Learning Environments - 1st Workshop MUPPLE'08*, volume 388 of *Series Distributing education services to personal and institutional systems using Widgets*, pages 25–33, Maastricht, The Netherlands„ 2008. CEUR Proceedings.

53 Scott Wilson, Paul Sharples, Dai Griffiths, and Kris Popat. Moodle wave: Reinventing the vle using widget technologies. In Fridolin Wild, Marco Kalz, Matthias Palmér, and D. Müller, editors, *Mash-Up Personal Learning Environments - 2nd Workshop MUPPLE'09*, volume 506 of *Series Moodle Wave: Reinventing the VLE using Widget technologies*, pages 47–58, Nize France, 2009. CEUR Proceedings.

54 Xiaobo Yang, Xiao Dong Wang, Rob Allan, Matthew Dovey, Mark Baker, Rob Crouchley, Adrian Fish, Miguel Gonzalez, and Ties van Ark. Integration of existing grid tools in sakai vre. In *Proceedings of the Fifth International Conference on Grid and Cooperative Computing Workshops*, Series Integration of Existing Grid Tools in Sakai VRE, pages 576–582, Changsha, China, 2006. IEEE Computer Society.

Enhancing Coherency of Specification Documents from Automotive Industry

Jean-Noël Martin[1] and Damien Martin-Guillerez[2]

1 **All4Tec**
6 Rue Léonard de Vinci – BP 0119 – F-53001 LAVAL Cedex, France
jnm@all4tec.net

2 **Inria Bordeaux Sud-Ouest**
351 cours de la Libération, 33405 Talence Cedex
damien.martin-guillerez@inria.fr

Abstract

A specification describes how a system should behave. If a specification is incorrect or wrongly implemented, then the resulting system will contain errors that can lead to catastrophic states especially in sensitive systems like the one embedded in cars.

This paper presents a method to construct a formal model from a specification written in natural language. This implies that the specification is *sufficiently* accurate to be incorporated in a model so as to find the inconsistencies in this specification. *Sufficiently* means that the error rate is down 2%. The error counting method is discussed in the paper. A definition of specification consistency is thus given in this paper.

The method used to construct the model is automatic and points out to the user the inconsistencies of the specification. Moreover once the model is constructed, the general test plan reflecting the specification is produced. This test plan will ensure that the system that implements the specification meets the requirements.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases coherency, specification, model generation, automatic text processing

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.225

1 Introduction

Specification documents define the way a system should behave and not how it is structured. They list the system's properties and limits that we will call qualifiers. Those documents are used to communicate between stakeholders of a project and are the reference documents to verify if an engineered system meets the requirements. To ensure the correctness of the system, modeling its specification with a formal paradigm such as Markov chains is of great importance. It indeed enables automatic tests [10].

Of course, specification documents contain errors, leading to defects in the design and thus in the product itself. However, determining whether a specification is correct or not is a hard job especially due to lack of criteria to judge whether a specification is correct or not. The IEEE has issued standards on the subject such as the IEEE 830 [1] and the IEEE1233 [2], but these standards provide only part of the criteria and no objective criteria to evaluate a specification. In fact, they define general rules but do not give support to a quantitative qualification nor for a complete list of qualities.

To determine objective criteria to judge the correctness of a specification, one must define what the means for a specification are to be correct. Of course, consistency is a key aspect of

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 225–237

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

correctness. Correctness includes both a good choice of words and the consistency itself. If a specification is consistent, then the construction of a formal model of the specification becomes possible In this paper, consistency is taken into account in our work and we automatically extract a Markov model for test generation software from a specification document from the automotive industry that is written in English. However a wrong choice of words will be reported in the model will show that there will not be any negative consequences. This paper is organized as follows: Section 2 presents the work related to this study and Section 3 presents the key aspects of specification documents that enable text analysis. After an overview of the method used in Section 4, we go through the several steps of the treatment of the specification in Section 5. We then examine how to improve the consistency of a specification document thanks to our system in Section 6. Finally, we conclude in Section 7.

2 Related Work

Since the work done by Chomsky in 1965 [7] on formal grammars, automatic language processing has evolved. Of course grammar parsing has also evolved. We can notably mention the works of Nique [18] and Winograd [23] on context-free grammar. More recently a new book had summarized the state of the art that was produced by Jurafsky and Martin [11]. Nique and Winograd explain in a clear way what the Chomsky approach is. Jurafsky opens on the use of Markov models. These analyses, from Winograd to Juravsky, provide a large base of information on how to process documents analysis in natural language and extract relevant subparts of a document. Other works of interest are based on the concept of translation as explained by Lutkens and Fermont [15], Planas [19]; especially those to compare text [20] using alignment techniques. However, no work has really succeeded in generating general purpose parser of natural language. Nevertheless, specification documents are well structured and we will show in this paper that it is possible to parse them, to provide accurate detection on non consistency, using some adaptation of the Chomsky grammar mixed with alignment techniques.

Extracting the model from this parsing requires having a consistent specification at the entry of the system. However, it is not clear what a consistent document is.

The consistency domain has been explored by Michel Charolles [4, 5] and by Fabien Wolf [24]. The sentential calculus is addressed initially by Michel Charolles [6] as the consistency theory. The world of uncertainty is discussed by R. Martin [17] and Lehrer [14] and deals with fuzzy logic that give a track to the alignment of texts. The concept of 'discourse framing', i.e., generating a tree structure giving a summary of the speech, comes from the work of Robert Martin [17] which relied on G. Fauconnier's work [8]. All those works give key points of what makes a text consistent but we found no work that precisely defines the component of consistency of a text. So we did it.

Actually, some research has addressed the topic of making a specification consistent with various results. Out of those, we can quote V. Gal [9], P. Serre [21] and M. W. Trojet [22]. All those works rely on the description of the specification in specific language like the language Z [15, 16] that is far from natural language.

This paper leverages the existing works on grammar and translation systems to parse the well-structured specification documents.

As an example of what we call a well structured document, Listing 1 exhibits a paragraph of a final writing of a specification. This start with a title which defines the universe and any of the above requirements include one or two preconditions linked to a predicate.

This paper also presents a definition of the six qualities that compose consistency and

Listing 1 An extract of a specification.

```
Speed
[DRL004-A-v1] the boot lid release feature shall be enabled by any
 boot lid release button, if the ignition is run or start and the
 vehicle speed is no more than 7 km/h.
[DRL004-A-v1] the boot lid release feature shall be enabled by any
 boot lid release button, if the ignition is off, independent of the
 actual vehicle speed information.
[DRL004-A-v1] the boot lid release feature shall be enabled by any
 boot lid release button, if the vehicle speed fault bit is set,
 independent of the actual vehicle speed information.
```

how the parsing uncovers consistency errors in specification documents. Indeed, Indeed, the previous contributions consisted mainly in giving example of what was an inconsistent text. We never found any definition of consistency except the one included in the dictionary.

Finally, this parsing results in a Markov model that can be used to prepare the test plan for the system.

3 Specification Structure

We based our work on four specification documents written in English from the automotive industry. Two of them come from Lear, the next comes from Audi and the last comes from BMW. From those documents, we extracted the overall form of a specification document. It is composed of four different semantic elements:

- an action (or predicate), denoted P, describes what is done (for example, *"the user presses the button to open the device"*, *"the user presses the button xxx that starts the motor that opens the boot lid"*), this defines the post-condition;
- a condition, denoted C, defines the initial conditions under which the predicate may apply (for example, *"when the vehicle is traveling slower than 7 km / h"*, *"when the Can message is not missing"*);
- a property, denoted Q, is a group of terms that gives quantitative data on the action (for example, *"during T_EnableBootLidRelease = 23 seconds"*, *"PJB_RearWindowsStatus = 0 within the Can message 433"*);
- and some background information, denoted A, treated as comments.

Thus, we consider that a specification is a sequence $(C+\ P\ Q*\ A*)+$[1]; in a specification one requirement is a set of non empty preconditions that applies to an action; this action is qualified by one, several or no property and is subject to potential comments. A specification is a list of requirements.

4 System Overview

We built software that convert a consistent specification written in English into a Markov model for test generation software. This system is based on the fact that specification documents are well-structured as presented in the previous section. This software is decomposed in 7 steps as shown in Figure 1:

[1] For clarity the + sign means one or more and the ∗ sign means zero, one or more.

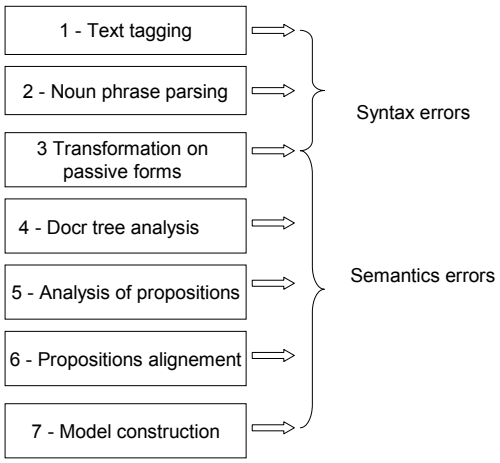


Figure 1 System overview.

1. A text tagging phase introduces the process, including the choice of stating the value of words in conjunction with a statistical tagger [3].
2. A text parsing step, based on the Chomsky grammar [7, 12, 13, 18], decomposes the text into word phrases, that is into elementary groups of words that have a grammatical function.
3. The sentences that are in passive forms are then transformed into active form.
4. The doc tree elements are added to the propositions which are categorized in $PQCA$ propositions.
5. The sentences are decomposed into three term propositions (subject / verb / complement) (see the work of F. Wolf [24]).
6. The propositions are aligned using classical translation methods. This enables the identification of similar terms (i.e., the same action or the same object) written differently in the text.
7. Finally, the propositions verbs are converted into states of a Markov Chain and the logic, extracted thanks to steps 2 and 5, is converted into transition between states. Subjects are the inputs and complements are the expected result.

This process succeeds if and only if most consistency errors have been removed from the specifications. From the experiment explained further we had an automatic diagnostic of 97.6% of errors found. This condition enables automatic detections of many consistency errors especially ambiguous statements in the specification.

5 System Description

This section presents each step done to parse a textual specification into a Markov Model.

5.1 Text Tagging

The text tagging tags the words of the documents with a set of 23 parts of speech. This is done thanks to Brill parser [3]. At this stage we initialize the value of words: the value of words is an extension of the excluded attribute given to weak words (weak words are the

Listing 2 Definition of the grammar of a proposition.

```
<C> => <triggers word><SN>
        | <triggers+> <Text><SV><(<pivot><Text>)
<P> => <SN> + <SV>
        | <GV> <SN><SV>
<Q> => <CONJ><SN>
        | <SN>
<Text>  => <SN><SV >
        | <SN>
        | <SV>
<SN> => <DET><N+><O>
        | <PREP> <SN>
        | <Past participle>
        | <DET><A><N+>
        | O
        | <DET><NP>
<SV> => <Aux><GV>
        | <GV>
<GV> => <V><Q>
```

words that have a poor semantic value: the articles, the auxiliary, etc...). The value of words consists of allocating a value to any words, mainly:

- 0 for weak words;
- 1 for normal words;
- 3 for numeric values;
- 5 for named entities;
- 8 for negative words.

The allocation of the value of word at the tagging stage prepare for a further accurate alignment of propositions. The value of word is chosen to provide a first idea of how to quote the word depending on the semantic discrimination of any word.

5.2 Text Parsing

As we explained before, a specification is composed of conditions C, predicates P, properties Q and comments A. As a consequence, our system leverages the regular form of specification to analyze it. The parsing is thus the extraction, for each specification line, of the C, P, Q and A elements.

This extraction is a classical grammar parsing using the grammar detailed in Listing 2. To illustrate this schema we express the first rule in words: a condition C is made of a trigger word that introduces a nominal noun phrase or a trigger that introduces a text followed by a verb phrase, etc.

This grammar is a sub part of the one proposed by Chomsky in [7] to include only part relative to the English used in the specification document. It specifies how a condition C, a predicate P, and a property Q are decomposed. Figure 2 shows the results of the parse tree of the sentence *"If the PJB is configured for S389 and receives a valid boot lid release request from the CRM"*. Of course the root of the parse tree in that figure is a condition C with the *"if"* word as the trigger. *"the PJB"* is the subject to the *"is configured for S389"* verbal group. *"and"* is a pivot introducing the text *"receives a valid boot lid release request from the CRM"*. Using this grammar, we thus have a basic decomposition of the text.

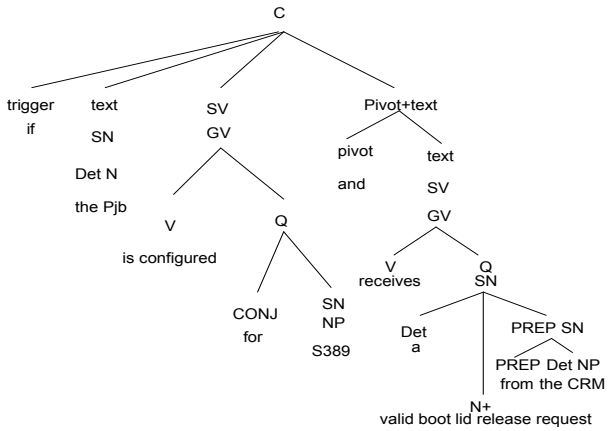


Figure 2 Grammar parse tree of a condition.

Listing 3 TPassive rule from the transformation defined by Chomsky.

```
TPassive:
SN1 + Aux + "to" + VPP + SN2 <=> SN2 + Aux + "be" + VPP + SN1
```

5.3 Active Form

Once the sentences are extracted using the previously described grammar, sentences in passive form are transformed into active form. This step is performed so that logic of the specification is direct and thus transferable to a Markov model.

First of all we stated that the transformations were dual. Assuming VPP stands for verb participle, which is changed to infinitive thanks to the rule exhibited in Listing 3. This rule comes from the Chomsky defined transformation but it is limited to the only sentence with exactly two verbs. We extended it with two new rules shown in Listing 4[2] to include all the possible cases especially sentences with three verb phrases and simpler sentences with one only verb phrase.

5.4 Doc Tree Adding

Thanks to the plan of the document we extract the "Universe" from the document and we construct a couple of propositions, adding a "Universe" to any proposition. The "Universe" concept comes from the works of R. Martin[17] and G. Fauconnier [8]. In our case we extract the universe from the plan of the documents.

The "Universe" step is the one where the major traps for incorrect statement are set.

[2] In Listing 4, noun phrases are labelled $GN1$ to $GN4$ and verb phrases VPP, $VP1$ and $VP2$

Listing 4 Grammar for passive forms with three verbs.

```
<VPP> <GN1> <VP1> <GN2> <GN3> <VP2> <GN4>
   <=>
<GN3> <GN4> <VP2> <"to"> <VPP> <GN1> <"to"> <VP1> <GN2>
```

5.5 Proposition Normalization

As shown in Figure 1, sentences are then decomposed in propositions by analyzing the parse tree obtained by the second step. Each proposition is a group of three groups: a subject, a verb and a complement. Each proposition is thus stored in that form and the logic between propositions is kept (for example, the *"if"* trigger of the sentence analyzed in Figure 2 is kept as a causal link between the sentence analyzed and the following sentence). The two propositions are *"the PJB is configured for S389"* and *"the PJB receives a valid boot lid release request from the CRM"*. The proposition analyzed in Figure 2 shows after that step:

1. *"the PJB"* (subject) *"is configured"* (verb) *"for S389"* (complement)
2. *"the PJB"* (subject is to be found) *"receives"* (verb) *"a valid boot lid release request from the CRM"* (complement)

The two propositions are expressed in the causal links *"if 1 and 2"*. Note that the subject of the second proposition needs to be extracted from the first proposition. After this step, we have a list of propositions decomposed into subject / verb / complement and the causal link between each proposition.

5.6 Proposition Alignment

The subsequent step is to align the propositions using translation techniques. This step helps to identify propositions or nominal groups that have the same meaning but that are written differently.

To do so, we use alignment techniques described by Lutkens et al. [15] and Planas [19]. To measure the similarity between sentences, the Jaccard measure is used [20] because it is simple and meets our needs. Used in conjunction with the value of words, it allows for an accurate comparison between propositions: for example two propositions with a different numerical value will be paired and if a negative word is in one of the propositions the alignment will fail. Thanks to the same propriety we can correct a wrong comma in a sentence. Calculating the Jaccard value with and without the value of word will disclose formal contradiction.

5.7 Markov Model Generation

Using the previous steps, we now have a list of propositions with their causal link in a database.

Using this, we can construct the Markov model for test generation software. The model for this software is described in [10] and was chosen for our process because it has proven its efficiency for test plan generation.

A Markov process is defined by a process in which the probability of going to a given state only depends on the current state. The direct consequence of this property is that a Markov process can be described as a series of states and a transition function. This function $\delta(x, y)$ gives the probability of transiting from state x to state y. The model used by the test generation software is a variation of Markov processes in which the transition functions are also labelled by an extra condition that should be met for the transition to be triggered (Figure 3).

This model thus represents perfectly the elements extracted from the specification. Each proposition is a state and the causal link a transition. The extracted model (Figure 4) can then be inserted into the test generation software that will construct the test plan for the specified system.

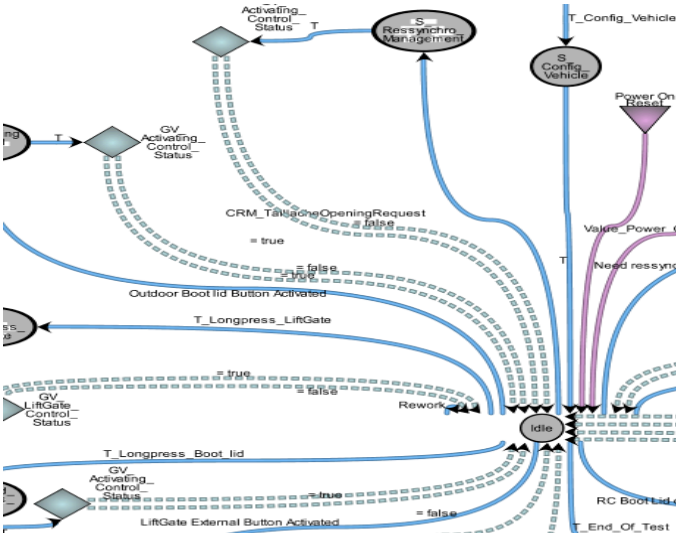


Figure 3 Extract of a manually constructed model.

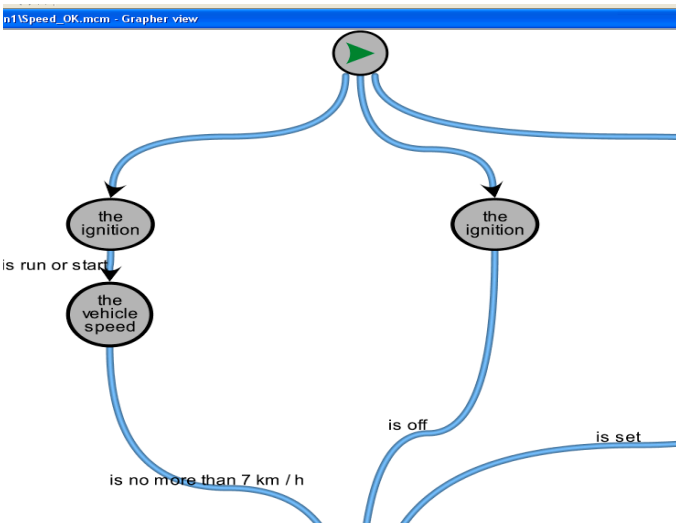


Figure 4 Extract of a manually constructed model.

6 Enhancing Consistency helped by the Model Generator

As presented in Figure 1, several steps in the model generation enable the detection of errors in the specification documents. In this section, we present how our system can be used to remove several consistency errors in specifications after presenting our definition of the consistency of a specification.

6.1 Consistency Approach

The word consistency distinguishes parts of an ensemble that focuses on a logical connection including a lack of contradiction in the devoted set; these parts are closely linked. They include a logical link and are organized to provide a stepwise link. This consistency initially dedicated to the discourse, proved suitable for the specification documents.

The first quality that we require from a text, related to consistency, is the property of a text to make sense. It is easy to infer informational relation between elements from one part of a document to the next. We will make use of the example provided by a personal access system equipment to demonstrate this property:

- The *"Pase"* is the smartcard that allows a vehicle user to open the vehicle doors and to click on the switch instead of the ignition key;
- The *"Pase"* user can approach his vehicle without taking the smartcard in his hands;
- He can see the flashers welcome him, and he can hear the door unlock;
- After entering the vehicle, he uses the *"Pase"* which he introduces into a smartcard reader, and he can then press the *"Start"* button to start the vehicle.

The inferences here are common, respecting the principle of relevance in the transmission of information. The inferences to be done to understand this text are progressive. The *"Pase"* replaces the ignition key, and there is a button that replaces the *"Neimann"*. Yet it must characterize these inferences, "the *"Pase"* is the smartcard (...) instead of the ignition key" and thus explicitly states the first inference, and "the *"Start"* button (...) starts the vehicle" explicitly states the second inference. Further progress is respected: we discussed the device, then we talk about the doors opening and finally about the car starting.

Remove the first sentence from this text and we can no longer justify its consistency. We must then make a less explicit inference. "The *"Pase"* is a remote command."

Note that the principle of relevance is more demanding in a written language when the preset is low than in the spoken language where the preset is important because of the knowledge of the listener. It must be recognized that many texts do not satisfy this property. We have many examples that give situations of ambiguity in the texts:

- In the text *"The phone rings. I drive my car."*, there are some connections to be established to achieve consistency.
- The text *"The vehicle was parked by Mark on a very busy place. The noise was terrible. Paul spent the evening on a bench beside the ocean. The wind was blowing. It was raining."* is given as an example of ambiguity: submitted to a panel of readers, interpretations range from *"there is a vehicle parked in a noisy place and strangely a man named Paul spends an evening at the seaside"* to *"Paul, who was in the camper is sad to stay there the next day because of bad weather"*.

6.2 Definition of the Consistency

We identified a series of documents qualities that make the documents understandable. Some of them are components of the documents consistency while others simply make the

documents less ambiguous and more understandable.

The qualities of documents that do not qualify as components of the consistency are:

- The undefined or ambiguous words: words that are not defined in the dictionary are to be defined or changed, a word in a text that combines a single definition in a given context is not ambiguous.
- The generic documents deserve to be written in the generic present tense, standardizing as much as possible the negative forms[3].

The qualities of the documents that qualify as components of the consistency:

- Cohesion and progressiveness: cohesion and progressiveness are the properties of a text that establish the continuity of the progress of the text, the property reflects the ability of text to be consistent in terms of chronological steps;
- Logical consistency: we define logical consistency as the absence of logical contradictions raised by the text;
- Clarity: we will define clarity as the property of a group of proposals to mean something in a clear way;
- Plausibility: the plausibility of an act is its ability to seem possible. In the field of natural language, we will consider as a plausible sentence a phrase that we are not surprised to hear [8]. Operationally we are considering the Dempster and Shaffer's theory, which can allocate two trust values to a predicate such as credibility of P defined by $Credibility(P) = 1 - Confidence(\neg P)$;
- Explicit knowledge: knowledge is explicit if it helps to understand a text without knowing the local context; it is based on the principle of relevance applied to items overlooked by the author in the specification;
- Accuracy of the text consists in lack of over information: we found in some texts two parts of the text which have exactly the same meaning. In the specification domains, this is named over specification.

Consistency of specification documents will be defined in part by our ability to generate a test model that can produce a test plan. The deficiencies in this criterion will be considered as inconsistencies in the text and we will work to detect and fill them with the cooperation of the author of the specification document.

If consistency requires a precise definition, it is not difficult to define the inconsistency: in fact the inconsistency is the property of a text whose consistency cannot be established.

6.3 Removing Inconsistencies in Specifications

We believe that our automatic translation of specification documents into the Markov model helps to remove most consistency errors in specification documents (we measured 97,6% consistency errors removed in our specification documents compared to manual consistency error removal).

The parsing of the text detects grammar errors and some ambiguous words. The user is prompted to define ambiguous words or to change them. We rewrites the document in the generic present tense. Defects related to cohesions and progressiveness are found at the universe step. At the model step, the absence of links (non-attainable states) could also disclose lack of logical consistency. The proposition alignment uncovers ambiguous and undefined propositions by duplicating links in the model. Finally, the relevance of the

[3] This quality is relevant for specification documents but seems to not apply to novels

Table 1 Table of defects automatically found. The defect rate is stated as the total weight of the defects rated to the number of words of the document.

	First specification			Second specification		
diagnosis	**occurrence**	**weight**	**total**	**occurrence**	**weight**	**total**
lexical conformity	148	1	148	18	1	18
missing words	14	1	14	2	1	2
text parts	4	3	12	0	3	0
syntax not conform	6	9	54	20	9	180
progressiveness	1	96	96	0	96	0
logical inconsistency	2	18	36	1	18	18
lack of clarity	0	36	0	1	36	36
explicit knowledge	3	9	27	7	9	63
accuracy	1	9	9	2	9	18
defect total weight			396			335
size of the document			1172			4440
defect rate			33,79%			7,55%

specification is hard to assess but the obtained model will be intricate if the text is irrelevant. Indeed, irrelevant phrases induce extra states and links in the model.

We applied those principles to two specifications from the automotive industry and it leads us to a specification that we believe is non ambiguous and permits the extraction of the model shown in Figure 4. In the automatic analysis we have an automatic diagnosis of non-correct writing and from that we issued the Table 1. This table shows the number of occurrences of a class of error, the estimated weight of a error (i.e., the number of modification the error requires to fix it) and the total weight of errors.

The defects extracted are the result of the whole process. The main parts are detected at the analysis step, mainly at the "Universe" step, but some errors are detected at the model step due to the model form that make the errors evident.

Finally we can say that the three major classes of error are balanced: a third is semantic (41%), a third is syntaxic (33%), and the remaining part is lexical. Among the semantic errors the main part is progressiveness then explicit knowledge, then logical consistency then clarity, and finally accuracy.

7 Conclusion

In this paper, we presented our method for automatically extracting a Markov model for a test generation software from a specification written in english using a natural language. We proved this method works by applying it to a real specification from the automotive industry. This method is now to be applied to large specifications up to 500 pages.

The principle that has been applied to automotive documents should soon be applied to other disciplines like defense or transportation.

This work is made possible thanks to the large literature in automatic language processing and to the structure of specifications. It is now being integrated into the MaTeLo software and we are extending it to handle other languages: French is soon to be used, Italian, Spanish and Portuguese are natural but German introduces specific problems.

This model extractor also help to make a specification consistent. We defined what we call the consistency of a document that is to say a document has to adhere to six properties. Each

step in the automatic analysis can disclose consistency problems and errors in the obtained model. This discloses the most dramatic consistency problems of the original specifications. Once the specification is consistent enough, the obtained model will generate the test plan for the final system. This system will increase the quality (less defects) of the resulting products.

A consistent specification might still need repetition of explicit knowledge to ensure that the reader who will implement the specification does not overlook important information. Ensuring such a thing in a current work will increase the overall consistency of specification documents. A last improvement in our roadmap is to address non-functional properties specified in the document. For instance, a specification can contain safety rules to avoid a certain state and those rules should be translated into the model. Our current implementation needs to be improved to handle those general rules.

Acknowledgements Jean-Noël Martin would like to thank his Thesis advisors very much: Bernard Levrat and Amghar Tassadit due to the fact they convinced him to do this project as it will be of benefit to his company.

References

1 IEEE 830 – recommended pactice for software requirement specifications, 1993.
2 IEEE 1233 – guide for developing system requirement specifications, 1996.
3 Eric Brill. A Simple Rule-Based Part Of Speech Tagger. In *Third Conference on Applied Computational Linguistic*, 1992.
4 Michel Charolles. Note sur la cohérence des textes. *Pratiques*, 10:105 – 111, 1976.
5 Michel Charolles. Text connexity, text coherence and text interpretation processings. In E. Sozer, editor, *Text Connexity, Text Coherence, Aspects, Methods, Results*, pages 1 – 16, 1985.
6 Michel Charolles. Cohérence, pertinence et intégration conceptuelle. *Intégration Conceptuelle*, 2002.
7 Noam Chomsky. *Aspects of the theory of syntax*. MIT Press, 1965.
8 G. Fauconnier. Domains and connections. *Cognitive Linguistics*, 1:151–174, 1990.
9 Viviane Gal. *Spécification à l'aide du langage LOTOS d'un algorithme de gestion d'une mémoire répartie à cohérence causale*. CNAM, 1995.
10 H. Le Guen and T. Thelin. Practical experiences with statistical usage testing. In *Eleventh Annual International Workshop on Software Technology and Engineering Practice*, 2003.
11 Daniel Jurafsky and James H. Martin. *Speech and Language processing*. Pearson International Edition, second edition, 2009.
12 J. J. Katz. *Sur la compréhension des phrases agrammaticales.*, chapter Semi-sentences, pages 400–416. Fodor and Katz, 1964.
13 J. J. Katz and P. Postal. *An Integrated Theory of Linguistic Descriptions*. MIT Press, 1964.
14 Keith Lehrer. Coherence, consensus and language. *Linguistics and Philosophy*, 7(1):43 – 55, 1984.
15 E. Lutkens and Ph. Fermont. A prototype machine translation based on extracts from data. Manuals, Université libre de Bruxelles, Bruxelles, 1985.
16 Clara Mancini, Donia Scott, and Simon Buckingam Shum. Visualing discourse coherence in non linear documents. *TAL*, 47(2), 2006.
17 Robert Martin. *Pour une logique du sens*. Presses Universitaires de France, 1983.
18 C. Nique. *L'initiation Méthodique à la grammaire générative*. Colin, 1974.
19 Emmanuel Planas. *Structure et algorithmes pour la traduction fondée sur la mémoire*. PhD thesis, Université Joseph Fourier, Grenoble, 1998.

20 Reinhard Rapp. Identifying word translations in non-parallel texts. In *the 33rd annual meeting on Association for Computational Linguistics*, 1995.

21 Philippe Serré. *Consistency of the geometrical specification for objects in n-dimensional Euclidian space*. PhD thesis, École Centrale des Arts et Manufactures, 2000.

22 Mohamed Wassim Trojet. *Approche de vérification formelle des modèles DEVS à base du langage Z*. 2010.

23 T. Winograd. *Language as a Cognitive Process*, volume 1: Syntax. Addison-Wesley, 1983.

24 Fabien Wolf and Edward Gibson. *Coherence in Natural Language*. Massachusetts Institute of Technology, 2006.

Probabilistic *SynSet* Based Concept Location

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

1 **Departamento de Informática, Universidade do Minho**
Braga, Portugal
{narcarvalho,jj,prh}@di.uminho.pt

2 **Escola Superior de Tecnologia e Gestão, Instituto Politécnico de Bragança**
Bragança, Portugal
mjoao@ipb.pt

Abstract

Concept location is a common task in program comprehension techniques, essential in many approaches used for software care and software evolution. An important goal of this process is to discover a mapping between source code and human oriented concepts.

Although programs are written in a strict and formal language, natural language terms and sentences like identifiers (variables or functions names), constant strings or comments, can still be found embedded in programs. Using terminology concepts and natural language processing techniques these terms can be exploited to discover clues about which real world concepts source code is addressing.

This work extends symbol tables build by compilers with ontology driven constructs, extends synonym sets defined by linguistics, with automatically created Probabilistic *SynSets* from software domain parallel corpora. And using a relational algebra, creates semantic bridges between program elements and human oriented concepts, to enhance concept location tasks.

1998 ACM Subject Classification D.2.5 Testing and Debugging: code inspections and walkthroughs

Keywords and phrases program comprehension, program visualization, concept location, code inspection, synonym sets, probabilistic synonym sets, translation dictionary

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.239

1 Introduction

Program comprehension provides valuable insight in many software evolution and maintenance tasks: bug hunting, fixing, feature improvements, etc. Reverse engineering techniques often rely on a mapping between human oriented concepts and program elements [15]. This mapping is required mainly because there is a big gap between the natural languages used to discuss and describe concepts in the problem domain, and the formal programming languages used to actually implement them [5]. To address this issue a clear definition of the program elements is required and also a way to relate these elements with human oriented concepts.

Although programming languages grammars are very formal and strict, and strongly limit the expressions and statements that can be composed to write programs, some small degree of freedom is still given to the programmer to use some more natural terms, when writing comments or naming functions and variables for example. These terms can give clues to which concepts the implementation is addressing, and the meaningfulness of these terms can have a direct impact in future program comprehension approaches [14].

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 239–253

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

In order to close this gap between languages a new structure needs to be devised to represent program elements with information derived and extracted from program identifiers. This work introduces the use of a Ontology Oriented Symbol Table (*OntOSymbolTable*), to represent elements in the program. A symbol table enriched with relations gathered from static analysis performed on source code. This table is used to represent some of the elements in the program, which gives a more structured and formal way to reason about source code. Program identifiers are present in this table and directly related with them is a data structure called Probabilistic Synonym Set (*ProbSynSet*), an extended version of synonymous sets used by linguistics to gather terms that are conceptually equivalent. These key structures are used as clamps to sustain conceptual bridges between program and other elements and are built from a terminology translation memory (in simple terms a dictionary) called *PTD*. This particular dictionary is built from parallel corpora using Natural Language Processing techniques [19], and can be used to expand the terms used to represent a concept. An important detail is that the text used to build the parallel corpora constrains the domain of terms that appear in the final dictionaries, this helps in keeping the calculated related terms in the same context. A complete definition and more details about these structures is presented in section 3.

The next section discusses other work in this area, and uses some of this work to motivate and substantiate the approach introduced in this paper. Section 3 introduces some concepts and definitions. Section 4 describes the relational algebra devised to relate program elements with other concepts, and Section 5 discusses how this algebra can be used in the context of concept location. Section 6 illustrates some case studies and experimental validation done so far to support the initial claims. And the last section concludes with some final notes and future work.

2 Related Work

Previous work shows the relevance of program identifiers when reverse engineering programs. *Lawrie et al* have shown that terms used as program identifiers have a direct impact on future comprehension tasks quality and accuracy [14]. A study by *Takang et al*, also shows that programs that use full terms for program identifiers, instead of abbreviations, are easier to understand [20].

Caprile et al state that program identifiers are one of the most relevant source of information about programs. This was so important that their work was about restructuring named identifiers to improve other program comprehension activities [6].

The relevance of program identifiers used clearly affects future program comprehension tasks, but how much can we rely on this source of information? The work of *Anquetil et al* try do define what it means to have a "reliable naming convention" [3] to later improve program readability, *Deissenboeck et al* propose a formal model that provides rules for concise and consistent naming [7].

Abebe et al presented their use of Natural Language Processing techniques for parsing program to extract concepts [1]. In this work they build an ontology from domain concepts extracted from source code, that can be later used to suggest which files can be more relevant to a specific software change. Although, there are many similar facts between this approach and the one described in this paper, one major difference is that natural language resources used by *Abebe et al* are found in the program, and some algorithms presented in this paper take advantage of resources built outside the program scope. Also the suggested elements where the concept can be found is more accurate than a file.

Falleri et al also used Natural Language Processing techniques to enhance the extraction of concepts from program identifiers analysis [10]. Their extracted artifact is similar to a WordNet [11] and can be used later to browse concepts found in the program in a hierarchical structure. Although, their work shares many objectives with this work, they are actually quite different, mainly because *Falleri et al* are solely based their lexical structure in terms used as identifiers, while in this work identifiers terms are expanded, and bags of conceptually equivalent words are used.

Lawrie et al also discussed how to expand possible abbreviations found in term used as program identifiers [13]. This work is directly related with the one described in this paper, and furthermore an inclusion on this expanding approach into this workflow would surely benefit the final results. Since the final terms used to map concepts would be using more meaningfully vocabulary.

In other work, *Enslen et al* introduced an algorithm to automatically split program identifiers in sequences of words. This is required mainly because unlike natural languages, identifiers do not use spaces or punctuation to join lists of terms (due to programming language syntax constrains), so other techniques are used [9]. Introducing this algorithm in the work described in this paper would probably help produce better results, since instead of processing an identifier that contains more than one term, for example using *CamelCase*, the list of terms could be processed for more accurate results.

The approaches described in this paper could also be used to complement other existent work. For example, *Bacchelli et al* discussed how to link e-mails free text with software artifacts [4]. The approach described in this paper could be used to help mapping concepts from both contexts.

These are some examples of previous work that help and motivate for improving techniques and approaches for exploring program identifiers found in source code [8]. And also show that there is a direct link between the meaningfulness of terms used as program identifiers and the degree of confidence and accuracy of the mapping between source code and human concepts build based on those identifiers. The accuracy of this mapping can improve future reverse engineering tasks, and improve program comprehension techniques and results.

3 Definitions and Concepts

To make it easier to discuss our contributions, this section presents some concepts and definitions.

3.1 Ontology Oriented Symbol Table

A symbol table is a data structure used to hold information about source code constructs, usually created by a compiler or interpreter [2]. Entries in the symbol table contain information about program identifiers, such as type or scope (it can vary depending on the language transformation being done). In its' simple form a symbol table can be defined as (note that some important data is being omitted, memory addresses for variables for example, this definition emphasizes the information required in the context of this work), that is why this structure is called $PseudoSymbolTable$. Another advantage of this simplification is that most probably it won't be necessary a full-featured compiler to build this table, more practical details on this subject in Section 4. In summary this table is a list, one triple for

each identifier found in the program:

$$PseudoSymbolTable = (id \times type \times scope)*$$

where,
id is the string that represents the term used as identifier;
type is the type of identifier (variable, function, etc.);
scope the scope where the identifier is declared, global or local, or a specific code block, sometimes this could be simply the line where the identifier appears.

This section introduces an Ontology Oriented Symbol Table ($OntOSymbolTable$) definition that will be used later to create the resources required to implement concept location. This table is defined as an hash-map [1] where the keys are program constructs identifiers ($PCid$) and the values are defined by the $Entry$ datatype:

$$OntOSymbolTable = PCid \rightharpoonup Entry$$

$$\begin{array}{rcrcl} Entry & = & id & : & String \times \\ & & type & : & PCType \times \\ & & prt & : & PCid \times \\ & & src & : & (File \times Line) \times \\ & & pss & : & ProbSynSet \\ & & & & (...) \end{array}$$

For each program identifier found a new entry is created in the table, where:
PCid is a string identifying a program construct, this is unique for an entire program (even for programs written across multiple files);
id is a string that represents the actual term used as identifier;
type the identifier type, not the type of variable the identifier is declaring (defined types are described later), when in a ontology context this represents the IS_A relation;
prt represents the parent for this program element, for example a function, code block, object or method, the element is identified by its' corresponding $PCid$, in an ontology context this represents the IN_CTX relation;
src specifies where the identifier can be found in the persistent storage medium, typically a filename and a line number;
pss the $ProbSynSet$ calculated from the identifier id, more details on this data structure in the next section;
(...) states that this definition is not complete, more information is to be added like call graphs edges and dependencies, but these elements are the most relevant in this article's scope.

The following types are defined to use has $PCType$: m

$$PCType = P + V + K + M$$

where,
P represents a procedure, can be used to represent functions or methods;
V represents a variable, used to represent local and global variables;

[1] An association between keys and values, where the keys used are the program unique identifiers, and the values the corresponding calculated structure for the identifier.

```
1 P(authenticate)@F(auth.c:14)
2 V(password)@F(auth.c:3)
3 V(username)@F(auth.c:2)
4 V(result)@P(authenticate)@F(auth.c:15)
5 P(main)@F(auth.c:20)
```

Figure 1 Example of a *OntOSymbolTable* textual representation.

K represents a constant, used to represent local and global constants;
M represents a module, used to represent aggregations of functions or methods (objects or libraries for example).

A function called *tNormalize* was defined that is used to normalize program constructs' types:

$$tNormalize : term \longrightarrow PCType$$

that given a *term* (normally obtained from the *PseudoSymbolTable*), returns the *term PCType*. For example:

$$tNormalize(\text{"method"}) = P$$

$$tNormalize(\text{"function"}) = P$$

$$tNormalize(\text{"variable"}) = V$$

A *OntOSymbolTable* allows a more structured and systematic reasoning about program elements, also more information can be added to build more complex artifacts, more on this later. This definition is language agnostic, so far it was only used in the imperative programming paradigm [21], like C/C++ or Java, but minor tweaks can be required if other details characteristic to other paradigms need to be represented. This means that a front-end can be easily built for a specific language, or a compiler refactored, to create this table, and take advantage of all the features described in the next sections. Algorithm 1 summarizes how to build this table.

Algorithm 1 Create a *OntOSymbolTable* as a hash-map.

Require: $T : PseudoSymbolTable$
  $oost \leftarrow \{\}$ *// start with empty hash-map*
  for all $(id_t, type_t, scope_t) \in T$ **do**
    $type \leftarrow tNormalize(type_t)$
    $prt \leftarrow$ *{* parent PCid determined based on* $scope_t$ **}*
    $src \leftarrow$ *{* file and line number of identifier *}*
    $pss \leftarrow ProbSynSet(id_t)$
    $PCid \leftarrow id_t + +prt$
    $oost[\,PCid\,] \leftarrow (id_t, type, prt, src, pss)$
  end for
  return *oost*

A textual representation of the content of this table was also devised. Figure 1 is a small snippet of the *OntOSymbTab* calculated from a C source file `auth.c`. For example in Figure 1, line 1 is stated that a function name `authenticate` exists in file `auth.c` line `14`, or from Figure 1, line 4 that a local variable named `result` is defined in function `authenticate` in file `auth.c` line `15`.

The next section defines probabilistic synonymous sets, used in a $OntOSymbolTable$ and how they are calculated.

3.2 Probabilistic Synonym Set

In linguistics, and in the terminology discipline for a given term a list of synonyms can be built, this list is commonly called a *SynSet*, for more details about these sets please refer to [11] [12] [19]. For this work an extended version of this data structure called $ProbSynSet$ is defined. A list of triples composed by a term t that represents the same concept as the original term, a relation set r that represents the kind of relation between this term and the original one (synonym or translation for example), and a probability p which defines the degree of confidence that this term is actually conceptually equivalent to the original term.

$$ProbSynSet = Term \rightharpoonup Triple$$

$$\begin{array}{rcrcl} Triple & = & t & : & Term & \times \\ & & r & : & RelSet \in \{\{S\},\{T\}\} & \times \\ & & p & : & Confidence \in [0,1] & \end{array}$$

where,

t is a term (word) conceptually equivalent to the given term;

r is the relation that exists between t and the given term;

p is the degree of confidence that t is conceptually equivalent to the given term, $p \in [0,1]$.

The $ProbSynSet$ is calculated using a Probabilist Translation Dictionary (PTD), this can be seen as a common translation dictionary but with some important subtleties. First, PTDs are usually build from parallel corpora [19] which means that the language domain can be restricted to a specific domain. This is important because if translations are being used to find synonyms, in a software development context we wouldn't want for example *fork*, which can have many different meanings depending on context, representing a piece of cutlery, but a process (or similar concept). The second thing is the certainty level implied in PTD, that will be used to calculate the degree of confidence in the $ProbSynSet$.

A PTD can de defined as a finite function that given a $term$ returns a list of possible translations for $term$ and the degree of confidence that this translation is correct. More formally:

$$PTD = term \rightharpoonup PTDEntry$$

$$PTDEntry = t \rightharpoonup t \times p$$

where,

t is a possible translation of $term$;

p is the degree of confidence that t is a correct translation, $p \in [0,1]$;

This set ends up being a list of quasi-synonym, a list of terms that can represent the same concept. Besides, probability p gives a fine tuning capability to broad or short the scope of term gathering in this list. Typically a *cut line*, a way of saying that below this degree of confidence terms are to be ignored.

Figure 2 illustrates the $ProbSynSet$ calculated for term $username$. And Algorithm 2 describes how they can be calculated. In this algorithm a $WebService$ is being used to

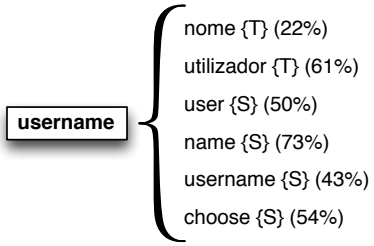


Figure 2 *ProbSynSet* for term *username*.

provide the required *PTDEntry*s. These resources were calculated and made available in the Per-Fide Project [2] environment.

Algorithm 2 Create a *ProbSynSet* for a given *term*.

Require: $term : String$
  $pss \leftarrow \emptyset$ // *start with empty hash*
  $ptd_{entry} \leftarrow WebService.PTD(term)$
  for all $k_i \in keys(ptd_{entry})$ **do**
    $(t_1, p_1) \leftarrow ptd_{entry}[\,k_i\,]$
    $pss[\,t_1\,] \leftarrow (t_1, \{T\}, p_1)$ // *add element to hash*
    $ptd^{-1}_{entry} \leftarrow WebService.PTD(t_1)$
    for all $k_j \in keys(ptd^{-1}_{entry})$ **do**
      $(t_2, p_2) \leftarrow ptd^{-1}_{entry}[\,k_j\,]$
      $pss[\,t_2\,] \leftarrow (t_2, \{S\}, min(p_1, p_2))$ // *add element*
    end for
  end for
  return pss

A more complete description of how PTDs are calculated and made available is out of scope for this article, please refer to references [18] [17] [16] for more details.

4 Relational Algebra

After calculating an *OntOSymbolTable* many information is available, including *ProbSynSet*s which are the base construct for creating semantic bridges between concepts. This section describes a set of algorithms that implement relational functions between *ProbSynSet*s. These functions are the minimal operations required to implement algorithms for relating terms and concepts. Figure 3 illustrates an example of comparing two terms: *username* and *user*, to determine if their *ProbSynSet*s are related, this could imply a semantic relation between these terms.

The first operation, that defines this relation, is the intersection ($\cap$) between *ProbSynSet*s. Although a *ProbSynSet* is a list, that can be seen as a set as defined by mathematics, intersection has to take in account probabilities and relation sets. This function is defined as:

$$\cap : ProbSynSet_A \times ProbSynSet_B \longrightarrow ProbSynSet_\cap$$

[2] http://per-fide.di.uminho.pt/

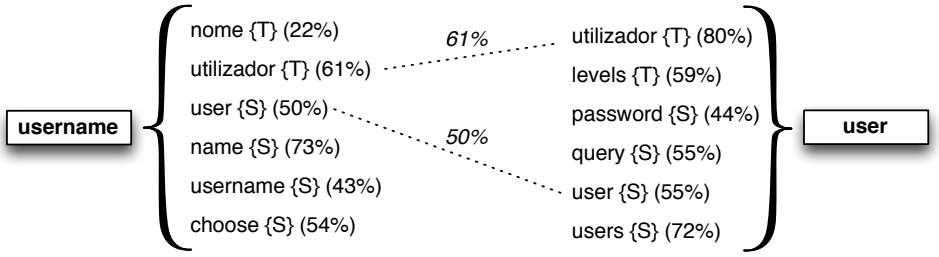


Figure 3 Relating *ProbSynSet*s for different terms.

Algorithm 3 describes how to calculate $ProbSynSetn_{\cap}$.

Algorithm 3 *ProbSynSet* intersection ($\cap$).

Require: $pss_A : ProbSynSet$
Require: $pss_B : ProbSynSet$
  $pss_{\cap} \leftarrow \emptyset$ *// start with empty hash*
  for all $k \in keys(pss_A)$ **do**
    $(t_A, r_A, p_A) \leftarrow pss_A[\,k\,]$
    if $t_A \in terms(PSS_B)$ **then**
      $(t_B, r_B, p_B) \leftarrow pss_B[\,t_A\,])$
      $r \leftarrow r_A \cup r_B$
      $p \leftarrow min(p_A, p_B)$
      $pss_{\cap}[\,t_A\,] \leftarrow (t_A, r, p)$ *// add element to hash*
    end if
  end for
  return $pss_{\cap}$

This operation can be used to claim that a non-empty intersection of *ProbSynSet*s implies that the terms are related.

$$ProbSynSet(t_A) \cap ProbSynSet(t_B) \neq \emptyset$$

$$\Rightarrow t_A \text{ is related to } t_B$$

This relations can mean that t_A and t_B are conceptually equivalent, if the degree of confidence in this relation is high enough. In order to be able to build ranks a similarity level function between two *ProbSynSet*s was defined (*simil*):

$$simil : ProbSynSet_A \times ProbSynSet_B \longrightarrow Float$$

Algorithm 4 shows how this function is implemented. This degree of similarity will be used in the next section to create ranks of conceptually equivalent suggestions.

Another operation required is the union of *ProbSynSet*s ($\cup$), defined as:

$$\cup : ProbSynSet_A \times ProbSynSet_B \longrightarrow ProbSynSet_{\cup}$$

This operation is used to join different *ProbSynSet*s in a single *ProbSynSet*. Algorithm 5 defines how this operation is implemented.

The next section shows how these operations can be used to implement algorithms for locating concepts in programs.

Algorithm 4 Similarity between two *ProbSynSets* (*simil*).

Require: $pss_A : ProbSynSet$
Require: $pss_B : ProbSynSet$
  $pss_\cap \leftarrow pss_A \cap pss_B$
  $similarity \leftarrow 0$
  for all $k \in keys(pss_\cap)$ **do**
    $(t, r, p) \leftarrow pss_\cap[\,k\,]$
    $similarity \leftarrow similatiry + p$
  end for
  return *similarity*

Algorithm 5 *ProbSynSet* union ($\cup$).

Require: $pss_A : ProbSynSet$
Require: $pss_B : ProbSynSet$
  $pss_\cup \leftarrow pss_A$ // *start with* pss_A *hash*
  for all $k \in keys(pss_B)$ **do**
    $(t_B, r_B, p_B) \leftarrow pss_B[\,k\,]$
    if $t_B \in terms(pss_U)$ **then**
      $(t_A, r_A, p_A) \leftarrow pss_A[\,t_B\,]$
      $p_B \leftarrow max(p_A, p_B)$
      $r_B \leftarrow r_A \cup r_B$
    end if
    $PSS_\cup[\,t_B\,], \leftarrow (t_B, r_B, p_B)$ // *set element in hash*
  end for
  return $pss_\cup$

5 Concept Location

Having defined basic operations to relate *ProbSynSet*s it is now possible to implement functions that make use of these basic operations to implement concept location oriented techniques. The most simple simple interesting function that can be implemented consists in a plain search of a concept in source code. Most of the times this is already possible, especially in modern development environments (Eclipse or Visual Studio for example). Commonly what these environments do is a previous collection of program constructs, and later when the programmer is trying to find a variable or function they do a simple pattern match between the term the programmer provides and the list of collected identifiers. This of course will not work if the programmer is using a term to look up a concept completely different from the one used by the original developer of the code. The advantage of this approach is that it tries to look up for conceptually equivalent terms, even if they were wrote using completely different words.

$$locate : OntOSymbolTable \times term \longrightarrow Rank$$

The *Rank* data structure is very simple, its' a list of pairs containing the program construct unique identifier *PCid*, and a positive real number representing the degree of similarity between the *ProbSynSet*s compared.

$$Rank = (PCid \times simil)*$$

Where,
PCid is the unique identifier for the program construct that matched as possible conceptually equivalent;
simil is the degree of similarity.
Algorithm 6 illustrates how the *locate* function can be defined.

Algorithm 6 Locate a *concept*.

Require: $oost : OntOSymbolTable$
Require: $term : String$
  $pss_C \leftarrow ProbSynSet(concept)$
  $rank \leftarrow \emptyset$ *// start with empty set*
  for all $pcid \in keys(oost)$ **do**
    $(id, type, prt, src, pss) \leftarrow oost[pcid]$
    $pss_\cap \leftarrow pss_C \cap pss$
    if $pss_\cap \neq \emptyset$ **then**
      $push(rank, (pcid, simil(pss_\cap))$ *// add element to set*
    end if
  end for
  return *rank*

This *locate* function is very simple, but more complex operations can be devised. For example, a *ProbSynSet* can be calculated by joining all the *ProbSynSet*s for all the program constructs found in a function, or code block. A real case scenario is for example having a function named f, that contains a variable named *username* and another named *password*. Although the identifier f by itself does not say much about what the function is dealing with, the variables *username* and *password* inside the function give a clue that is possible that this function is related with authentication or authorization. This means that joining the

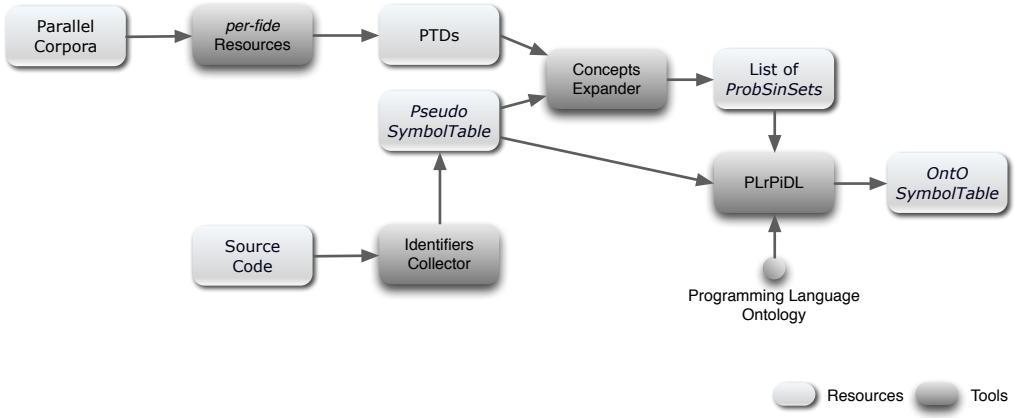


Figure 4 Web application architectural overview.

ProbSynSets for the identifiers found inside the function could probably suggest this, even that the name of the function by itself does not provide any significant semantic information. The required functions for *ProbSynSets* required to implement this algorithm were described in the previous sections: $\cup$ and *simil*. Similar reasoning functions could be implemented for code blocks, objects or even files.

6 Experimental Validation

In order to validate the approach described in this paper a set of tools were written to implement all the algorithms and functions described in the previous sections. This set of tools where then used to implement a web application prototype that using a browser would provide programmers an environment for locating concepts in programs.

The prototype application architectural overview is illustrated in Figure 4. The dark gray boxes represent tools that consume and create resources (which are represented in the light gray boxes), and the arrows represent the data flow of data between tools to produce the final *OntOSymbolTable*.

The *Identifiers Collector* is a tool that analyses source code and builds a *PseudoSymbol Table* that includes all the required information to create a *OntOSymbolTable*. This tool was implemented using *Exuberant Ctags* [3], that is normally used to create tag files that are used by editors or integrated development environments to implement features like auto-complete. This tool already provides support for a broad range of programming languages, this way it would be easier for our prototype to support all of those programming languages.

The *Concepts Expander* is a tool that can create *ProbSynSets*. It makes use of the *PTDs* created in the Per-Fide Project environment. These are built from parallel corpora automatically, and software related texts were used to try to constrain the vocabulary domain to the context of software development. Around *700 MB* of software related texts, for example documentation and user interface messages, in several languages were used to build the parallel corpora. This would help ensuring that the translation memories calculated terms were inside the software domain. The languages used were mainly portuguese and english, this means that the bag of conceptually equivalent terms in a *ProbSynSet* includes

[3] http://ctags.sourceforge.net/

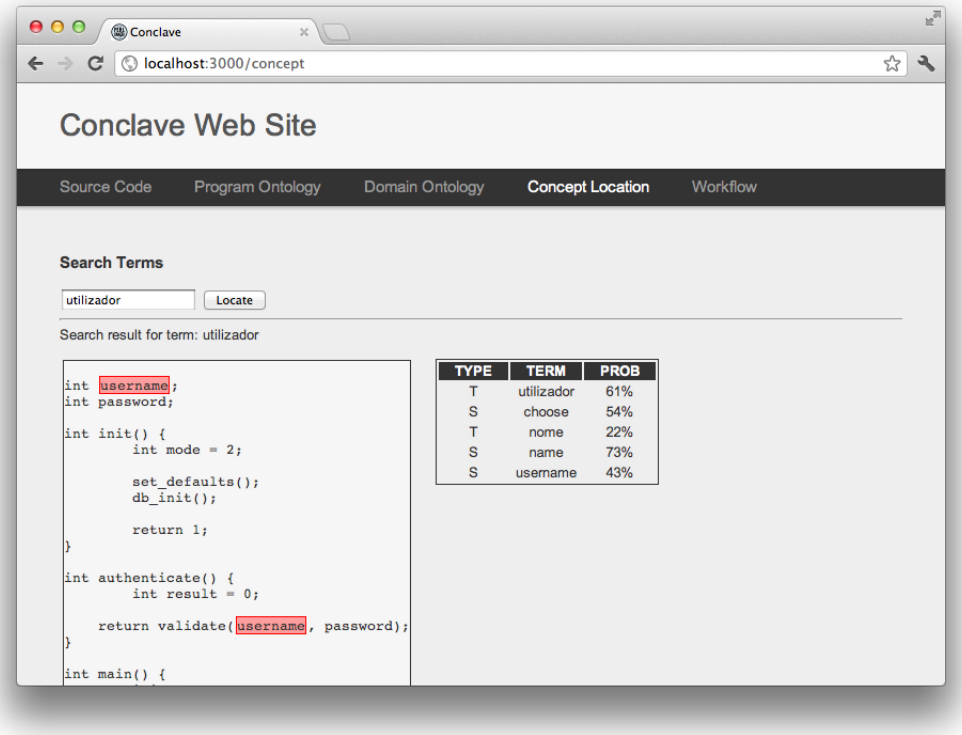


Figure 5 Web application screenshot.

portuguese and english terms. This is helpful if identifiers were not written in english, and the program maintainer is an english native speaker. A multilingual aware concept search is another advantage of this implementation. The more languages used to build the parallel corpora, the more languages can be supported.

Finally *PLrPidL* is a tool that creates a *OntOSymbolTable* for a given program, using the resources created by other tools. Figure 5 illustrates the application when searching source code for a term that was not in the original list of identifiers, but was found in a *ProbSynSet*, and therefor the application can highlight in the source code the terms that have high probability of representing the same concept the user was searching. The searched term *utilizador*, is a possible portuguese translation for *username*, which in addition to not being in the identifiers list is not even written in the same language.

These tools can also be used independently or composed in other workflows. Table 1 illustrates an initial study done with identifiers using these tools. The initial question that motivated this study was if the list of program identifiers is actually composed by words that a program maintainer would try to search for. The object of this study was TTH [4] a program that creates a HTML file from a LaTeX file.

Typically the first job of a programmer when acting as a maintainer for a given program is to actually find the source code responsible for implementing a part of the specification.

[4] http://hutchinson.belmont.ma.us/tth/

Table 1 Comparing Terms Found.

	# Identifiers Found	# *ProbSynSet* Created
TTH	925	925

	Strings	Words	%
Identifiers	476	55	12%
ProbSynSet	1659	1039	63%

Normally the maintainer needs to verify several areas of the code manually until the correct zone (or zones) that need update are found. The search usually starts with words that are related with the concept that needs to change, for example terms like *date*, *username* or *create* are common search keywords if for example trying to fix a bug in the creation date function. Most of the times this means that the programmer is using keywords in his natural language that in his domain are suitable candidates to represent the concepts that require changes. But, most of the times program identifiers are not so explicitly and many different names and versions of the same name can be used. For example looking at Table 1 the identifiers collector tool found 476 unique strings that identify program elements (variables, functions, etc). If this list of strings is checked in a typical dictionary, since the author of the code used english to write comments and the documentation the english dictionary of words was used, and we verified that only 55 of these strings were actually found. Well, if a programmer looking for source code zones to edit normally uses words in his domain language it will have a very low chance of success locating program elements that are related with the search key words. The *ProbSynSet* list of related terms was then used. A *ProbSynSet* is calculated for every identifier found in the program, this means that a total of 925 were calculated. Every *ProbSynSet* has a bag of terms that are conceptually equivalent to the original term found as identifier. This means that the scope of terms, linguistic speaking, was broaden. A total of 1659 strings are now available for the programmer to search for. And since 1039 of these strings are present in the dictionary the chances that the maintainer finds a word was greatly increased. This does not necessarily implies that the programmer will find the important zones of code to change, but will greatly improve the chances that a zone of code will be suggested based on key words search. In an information retrieval context we could say that the recall was greatly increased. An experiment that would verify if the precision is also increased is currently being devised.

7 Conclusion and Future Work

Several program comprehension techniques often rely on a mapping between program elements and more natural language terms that represent human oriented concepts. Program identifiers are a possible source of information about which specific source code areas are responsible for implementing these concepts. The meaningfulness of program identifiers directly influence the quality of the required mapping, and also future accuracy of concept locating tools.

The adoption of an ontology oriented table to represent identifiers, which describes not only the identifiers but also a set of more discovered information about them, provides an artifact that can be object of a systematic reasoning. Using this reasoning approach views of programs can be built that provide useful insight for locating code responsible for implementing concepts. Probabilistic *SynSet*s are an important element of this table that

provides anchors that can increase the recall concept location based on keywords search and similar approaches.

The implemented tools and current prototype help showing that concept location tasks recall and precision can be greatly increased by taking advantage of *OntOSymbolTable*. This ontology oriented constructs enriched table, can also be object of other processing functions using very simple and elegant basic operations to implement different analysis tasks.

Interesting tasks for further development of this work:

- combination of techniques described in the related work section, for example expand known abbreviations before calculating *ProbSynSet*s:
  - one example is using the work described in Section 2 to expand terms often written using abbreviations, or other shortening styles like camel case, this would allow the use of more accurate terms to build *ProbSynSet*s;
  - another example is adapting the algorithms described in this paper to allow the use of multi word terms, this would allow using identifiers that contain several words (sometimes joined with _ , for example *get_data*);
- experimental verification of the precision of recall of suggested areas of source code, similar studies like the one introduced in Section 6.

References

1 S.L. Abebe and P. Tonella. Natural language parsing of program element names for concept extraction. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 156–159. IEEE, 2010.

2 A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007.

3 N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, page 4. IBM Press, 1998.

4 A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 205–214. IEEE, 2009.

5 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, 1993.

6 B. Caprile and P. Tonella. Restructuring program identifier names. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 97–107. IEEE, 2000.

7 F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

8 B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

9 E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. Mining source code to automatically split identifiers for software analysis. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 71–80. IEEE, 2009.

10 J.R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic extraction of a wordnet-like identifier network from software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 4–13. IEEE, 2010.

11 C. Fellbaum. Wordnet. *Theory and Applications of Ontology: Computer Applications*, pages 231–243, 2010.

12 J.J. Jiang and D.W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *Arxiv preprint cmp-lg/9709008*, 1997.

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

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

15 V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

16 A. Simoes, X.G. Guinovart, J.J. Almeida, and P. Natura. Distributed translation memories implementation using webservices. *Procesamiento del lenguaje natural*, 33:89–94, 2004.

17 Alberto Simões and José João Almeida. Parallel corpora based translation resources extraction. *Procesamiento del Lenguaje Natural*, (39):265–272, September 2007.

18 Alberto M. Simões and J. João Almeida. NATools – a statistical word aligner workbench. *Procesamiento del Lenguaje Natural*, 31:217–224, September 2003.

19 Alberto Manuel Brandão Simões. Parallel corpora word alignment and applications. Master's thesis, Escola de Engenharia - Universidade do Minho, 2004.

20 Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.*, 4(3):143–167, 1996.

21 D.A. Watt, W. Findlay, and J. Hughes. *Programming language concepts and paradigms*, volume 234. Prentice Hall, 1990.

A Multimedia Parallel Corpus of English-Galician Film Subtitling

Patricia Sotelo Dios[1] **and Xavier Gómez Guinovart**[2]

1 **University of Vigo, Galicia, Spain**
psotelod@uvigo.es
2 **University of Vigo, Galicia, Spain**
xgg@uvigo.es

Abstract

In this paper, we present an ongoing research project focused on the building, processing and exploitation of a multimedia parallel corpus of English-Galician film subtitling, showing the TMX-based XML specification designed to encode both audiovisual features and translation alignments in the corpus, and the solutions adopted for making the data available over the web in multimedia format.

1998 ACM Subject Classification H.3.1 Content Analysis and Indexing

Keywords and phrases corpora, multimedia, translation, subtitling, XML

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.255

1 Introduction

The CLUVI Corpus is an open collection of parallel text corpora that covers specific areas of the contemporary Galician language. With over 23 million words, the CLUVI Corpus comprises six main parallel corpora belonging to five specialised registers or domains (fiction, computing, popular science, law and administration) and involving five different language combinations (Galician-Spanish bilingual translation, English-Galician bilingual translation, French-Galician bilingual translation, English-Galician-French-Spanish tetralingual translation and Spanish-Galician-Catalan-Basque tetralingual translation). Among the various applications of this corpus, it has been used mainly for lexical extraction in terminology and translation [8, 14, 10, 11, 7].

The format chosen for storing the aligned parallel texts is an adaptation of the TMX format [9], as this is the XML encoding standard for translation memories and parallel corpora, regardless of the application used. A translation memory is a database that collects and records source text segments and their corresponding translated versions with the purpose of being reused for further translations via a computer-aided translation system. Albeit with some differences, an aligned parallel corpus is equivalent to a translation memory. In fact, the last few years have seen an increasing number of TMX-encoded aligned parallel corpora, which offer the additional advantage that they can be used as translation memories for feeding computer-aided translation programs (as proposed in [15]).

In this paper, we present the methodology developed by the SLI (Computational Linguistics Group of the University of Vigo) for building and processing the Veiga Corpus, a multimedia extension of the CLUVI featuring English-Galician cinematographic parallel texts, which is currently underway. In the following section we describe the data and briefly discuss the nature of the corpus. Section 3 deals with the actual construction of the corpus, including annotation and the two-layered segmentation and alignment processes. Section 4

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 255–266

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

arises some discussion about the potential use of such a corpus as a tool for researchers, teachers and subtitling practitioners. And in the last section, we draw some conclusions, point to certain challenges and outline possible future directions in terms of corpus development and research.

2 The Veiga Corpus

The Veiga Corpus is a small (approximately 300,000 words) but ever-growing English-Galician corpus consisting of 24 American, British, and Australian English-language films subtitled in both English (intralingual subtitling) and Galician (interlingual subtitling) for DVD, cinema and Internet distribution. Developed under the broader framework of the CLUVI Corpus, the Veiga was born as a text-only corpus of subtitles. It was not until very recently that we decided to make it multimedia, as soon as we found the appropriate tools to process the data and to make it accessible to the public in what we considered to be an appropriate way. The Veiga multimedia corpus of subtitles is already available for public consultation at `http://sli.uvigo.es/CLUVI/vmm_en.html`. However, it should be noted that only 13 of the 24 films are available in multimedia format at this writing.

Unlike the other CLUVI corpora, strictly speaking the Veiga Corpus cannot hold the title of parallel, nor can it be given the label of comparable, in accordance with how these concepts are traditionally understood and defined in the literature. The term 'parallel corpora' usually refers to 'original, source-language texts in language A and their translated version in language B' [1]. In contrast, a comparable corpus can be defined as a document collection composed of two or more disjoint subsets of documents, each written in a different language but dealing with the same topic.

Thus, the Veiga Corpus inhabits a certain intermediate land between a parallel and a comparable corpus. On the one hand, the Galician subtitles cannot be deemed to stand for translations of the English subtitles, although it could also be the case that subtitlers used the English file (if available) when translating into Galician. On the other hand, these two subsets share more than their semantic content: they both could be considered versions of the same original audiovisual text. Hence, we could say that the relationship among the original English subtitles and the Galician subtitles is triangular shaped.

The real, strict parallelism would be that occurring between the original text and each of the two subsets of English and Galician subtitles. The English set would correspond to a very particular type of transcription, which is known as intralingual subtitling, and the Galician set would embody an also very special modality of translation, which is given the name of interlingual subtitling. And yet, a parallel relation is very likely to come into play between the two sets of subtitles as well –a peculiar kind of 'cotranslation'–, inasmuch as they both are 'sub-products' of the same original text. In sum, a double unidirectional parallel may be established between the original audiovisual text and the subtitles, and a bidirectional correlation is also expected to exist between the subtitles themselves, as shown in Figure 1.

Obviously, a text-only corpus of subtitles would not allow for any kind of parallel observance in terms of source vis-a-vis translated text. By giving users access to the original audiovisual product some comparisons and parallelisms can be made, providing them with the opportunity to explore the manifold dimensions of subtitles, as for example phenomena related to the semiotics of interlingual and intralingual subtitling.

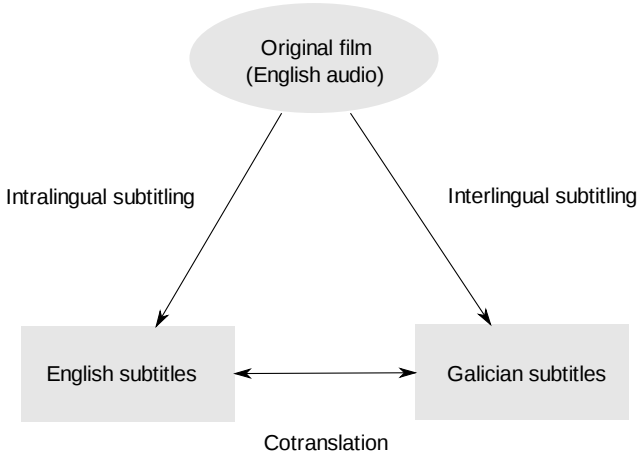


Figure 1 The subtitling triangle.

3 Tagging the Subtitling Triangle

As mentioned before, the Veiga Corpus is hosted at the CLUVI Corpus collection. The CLUVI Corpus functions as a repository of parallel subcorpora of different sizes and thematic fields, all of which undergo identical compiling and processing routines, and can be similarly accessed from one single search interface. Nonetheless, the Veiga Corpus requires further processing in comparison to the other CLUVI subcorpora. Besides annotating stylistic aspects of translation such as omissions, additions and reordering of translation units, all the subtitles include both the in-cue and out-cue time and the line break indicator, allowing users to examine aspects which are inherent to the subtitling practice, e.g. time and space constraints, segmentation, and condensation, among other particularities. In addition to this, the multimedia version of the Veiga incorporates a bonus feature: it enables users to stream the video clips corresponding to the bilingual pairs found in the search results, thus giving them access to the (co-)text in its original, multi-semiotic form. This means that wherever there is a result that matches the query in text format, the search interface shows a link to the corresponding video clips subtitled in each of the two languages involved (English and Galician). All the above mentioned aspects of the Veiga Corpus are annotated according to the TMX-based XML CLUVI specification for parallel corpora, which is summarized in Listing 1.

3.1 Tagging Subtitles at Textual Level

The basic segmentation unit for the alignment of the CLUVI bitexts is the orthographic sentence of the source text. Therefore, the correspondence between source and target text will always be of the 1:n type. The most frequent case is to have one sentence of the source text that corresponds with one sentence of the translation (1:1). Nevertheless, there are instances in which a source sentence is not translated (1:0), or in which a source sentence corresponds with half a sentence (1:1/2) or with two sentences of the translation (1:2), or even in which a sentence of the translation does not correspond with any source sentence (0:1). Moreover, translating frequently implies rearranging and relocating sentences and parts of sentences, in such a manner that these sentences or segments get moved to a different position in the translated text. These elements are reordered in the target section of the CLUVI parallel corpora in order to match the 1:n alignment criterion that preserves the

Listing 1 TMX-based CLUVI specification.

```
<!-- CLUVI_TMX DTD -->
<!ELEMENT cluvi_tmx (header, body) >
<!ATTLIST cluvi_tmx
       version CDATA #REQUIRED >
<!ELEMENT header (#PCDATA)>
<!ELEMENT body (tu*) >
<!ELEMENT tu (tuv+) >
<!ELEMENT tuv (seg) >
<!ATTLIST tuv
       xml:lang CDATA #REQUIRED>
<!ELEMENT seg (#PCDATA | s | l | hi | ph)*>
<!ELEMENT hi (#PCDATA | l)*>
<!ATTLIST hi
       type CDATA #IMPLIED
       x CDATA #IMPLIED>
<!ELEMENT ph EMPTY>
<!ATTLIST ph
       x CDATA #IMPLIED>
<!ELEMENT s EMPTY>
<!ATTLIST s
       n CDATA #IMPLIED
       d CDATA #IMPLIED
       a CDATA #IMPLIED>
<!ELEMENT l EMPTY>
```

integrity and the disposition of the translation units of the source text. This criterion is crucial when applied to the processing of multilingual corpora, where source sentences must provide for the establishment of correspondences among equivalent sentences in various languages.

The TMX specification does not consider the encoding of these translation phenomena, it has been designed for storing and exchanging translation memories and not for representing equivalent segments in parallel corpora. The TMX-based CLUVI encoding system uses an adapted version of some of the tags which are part of the TMX 1.4 specification [13] in order to represent the not-1:1 correspondences and reorderings encoded in the CLUVI parallel corpora. The aspects of translation encoded in the CLUVI corpora can be described as either omission, addition or reordering, and will be tagged using an adapted version of TMX 1.4 content elements <hi> and <ph>.

An omission occurs when an item of the source text does not correspond with any item of the target text, that is, when a sentence or part of a sentence is not translated. Omissions in the CLUVI parallel corpora are encoded by means of the <hi> element. According to the TMX 1.4 specification, the <hi> (or highlight) element 'delimits a section of text that has special meaning, such as a terminological unit, a proper name, an item that should not be modified, etc.' [13]. In the TMX-based CLUVI encoding, the <hi> element marks the piece in the source text that is omitted in the target text. This use of the <hi> tag is noted by means of the type attribute with the "supr" value. For instance, the non-translation of the English source sentence 'Yeah, okay, I'll be there' in the alignment of Wim Wenders' film

Listing 2 Example of omission in the Veiga Corpus.

```
<tu>
  <tuv xml:lang="en"><seg><s n="19" d="00:08:30,411" a="00:08:31,924">Oh
      okay, all right.</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="24" d="00:08:31,371" a="00:08:32,599">
      ¡Vale!</seg></tuv>
</tu>
<tu>
  <tuv xml:lang="en"><seg><s n="20" d="00:08:32,011" a="00:08:36,084"><hi
      type="supr">Yeah, okay, I'll be there.</hi></seg></tuv>
  <tuv xml:lang="gl"><seg>[[---]]</seg></tuv>
</tu>
<tu>
  <tuv xml:lang="en"><seg><l/>I'll get there as fast as I can.</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="25" d="00:08:34,411" a="00:08:37,130">Irei
       o máis axiña que poida.</seg></tuv>
</tu>
<tu>
```

Paris-Texas –included in the Veiga Corpus– would be encoded as shown in Listing 2.

On the other hand, the translation technique known as addition involves the insertion of elements in the target text that have no correspondence in the source text. Addition is also encoded in the CLUVI by means of the <hi> element, which highlights the inserted unit in the target text. This use of the <hi> tag is indicated by means of the type attribute with the "incl" value. The added text joins the translation unit into which it is inserted. If the new element is a sentence (or a sequence of sentences), it joins either the preceding or the following translation unit, depending on its context, thus respecting the 1:n alignment criterion, as shown in the example of Listing 3 excerpted from this same film.

Reordering in translation implies moving sentences or parts of sentences from their original position in the source text to a new location in the translated text. These displaced elements are reordered in the target section of the CLUVI parallel corpora to fit with the 1:n alignment criterion that preserves the integrity and the order of the translation units of the source text. Reordering in the CLUVI is encoded by means of a combination of the <hi> element and the <ph> element. The phrase or sentence that is being moved is tagged with the <hi> element, whose type attribute has the value "reord", and an x attribute with a numeric value that acts as an unambiguous index. In addition, a <ph> element in the translated text indicates the original location of the item being moved. According to the TMX 1.4 specification, the <ph> (or placeholder) element is used 'to delimit a sequence of native standalone codes in the segment. Standalone codes are codes that are not opening or closing of a pair, for example empty elements in XML' [13]. In the TMX-based CLUVI encoding, the adapted <ph> element marks the departure point of the moved text block, and the relationship between this piece of text and its place of origin is encoded in the <ph> element by means of an x attribute that has the same value as the index encoded in the corresponding <hi> tag of the segment being moved. Given the need for synchrony between the subtitles and the audiovisual narrative that is distinctive to the practice of subtitling, reorderings are very rare in the Veiga Corpus. The example in Listing 4 –gathered also from the above mentioned film– shows how reorderings are encoded.

Listing 3 Example of insertion in the Veiga Corpus.

```
<tu>
  <tuv xml:lang="en"><seg><s n="89" d="00:18:25,251" a="00:18:27,242">O.K.,
       I'll be right back.</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="100" d="00:18:26,011" a="00:18:27,808">
      Volvo de contado.</seg></tuv>
</tu>
<tu>
  <tuv xml:lang="en"><seg>[[---]]</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="101" d="00:19:01,051" a="00:19:05,010"><hi
       type="incl">CALZADOS<l/>FERRAXERÍA</hi></seg></tuv>
</tu>
<tu>
  <tuv xml:lang="en"><seg><s n="90" d="00:20:13,571" a="00:20:14,845">Damn
       it.</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="102" d="00:20:14,451" a="00:20:15,486">
      ¡Merda!</seg></tuv>
</tu>
<tu>
```

Furthermore, the tagging of the Veiga Corpus includes annotation of line breaks in subtitles. Line breaks within subtitles are encoded in the Veiga with the <l/> tag, an element added to the TMX 1.4 specification to allow for the examination of aspects which are relevant to subtitling, such as typographical conventions and space constraints.

The Veiga texts were aligned using the free alignment software Trans Suite 2000 Align, which performs automatic segmentation and alignment of both source and target texts. This tool operates at the sentence level, meaning that whenever the system detects a punctuation mark in either language, a new segment is identified and created. Considering that suspension dots in subtitling are also used to indicate that the sentence is not finished and will continue in the next subtitle, and that each subtitle timecode is enclosed in square brackets –which are often erroneously recognized by the software as sentence boundaries–, some manual checking and editing of the automated segmentation and alignment needs to be done. Besides, as mentioned earlier, we must comply with the 1:n criterion, which involves segment merging and splitting, mainly in the target side.

3.2 Tagging at the Textual/Audiovisual Interface Level

Tagging the Veiga Corpus at the textual/audiovisual interface level implies, on one hand, tagging the correspondences between the English subtitles stored as XML textual data in the TMX-based CLUVI encoding and the equivalent segment of the original English-language film with English subtitles, and, on the other hand, tagging the correspondences between the Galician subtitles stored as XML textual data and the equivalent segment of the original English-language film with Galician subtitles. In order to be able to establish these textual/audiovisual correspondences, all of the Veiga English-language films have been cut into video clips, each one corresponding to a subtitle. A first step is to check if the subtitles are in sync with the movie. In some cases, mostly when the subtitle file and the movie come from different sources, we need to edit the subtitles (using the freeware tool

Listing 4 Example of reordering in the Veiga Corpus.

```
<tu>
  <tuv xml:lang="en"><seg><l/>Everybody, everybody'll see me.</seg></tuv>
  <tuv xml:lang="gl"><seg><hi type="reord" x="2">-Vanme ver todos.</hi></
      seg></tuv>
</tu>
<tu>
  <tuv xml:lang="en"><seg><s n="354" d="00:47:21,131" a="00:47:24,328"><hi
       type="supr">No, Travis, I insist.</hi></seg></tuv>
  <tuv xml:lang="gl"><seg>[[---]] </seg></tuv>
</tu>
<tu>
  <tuv xml:lang="en"><seg><l/>He'll wait for you out in front.</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="381" d="00:47:22,051" a="00:47:25,009">-
      Esperarate na porta.<ph x="2"/></seg></tuv>
</tu>
```

Subtitle Workshop) and add a time delay (forward or backward) so that their speed matches that of the video. Secondly, we embed the subtitles in the two languages in the original film with a free, open-source video editing tool called VirtualDubMod. And finally, we edit each film subtitled both in English and in Galician and segment it into subtitles.

Therefore, we come up with two subsets of subtitled video clips, one in English and the other in Galician, each made up of as many videos as subtitles has the corresponding film. Moreover, given that a high number of subtitles are not long enough to be played and watched properly (they are only one or two seconds long), each individual clip/subtitle is allotted ten extra seconds –five seconds before the subtitle shows up, and five seconds after it fades out–, thus providing the viewer with some context. Needless to say, this segmentation process is very monotonous and time consuming, which is a major hurdle that could be overcome if we found a freeware video editing tool offering customary, automatic batch splitting features. Now, once we get two sets of subtitled clips for each film, we link them to their corresponding text in the bitextual TMX-based CLUVI representation by means of their video clip identification tag, encoded both in the TMX file and in the video clip (in its file name).

These two sets of subtitled clips are stored as FLV files (because of their compression rate and small file size) in the server file system, where they are named –with a unique file name– according to their film title, their subtitle language (English or Galician), and their sequential number. Thus, whenever users search the Veiga they get both the bilingual text pair and the clips where this text/subtitle appears. On the other hand, the bitextual TMX files are stored with a file name according to their film title, and include the tags of both the in-cue and out-cue time of each subtitle and their sequential number. This information is encoded in the Veiga Corpus with a second element added to the TMX-based CLUVI tagging: the <s> element, which contains three attributes –s for the sequential number, d for the in-cue time, and a for out-cue time– for each tagged subtitle. To illustrate this, Listing 5 shows the code included in the TMX file named peixe.tmx (from the film entitled *Shooting fish*, by Stefan Schwartz) that would correspond to the video clips stored in the file system as peixe_en-848.flv, peixe_en-849.flv, peixe_gl-808.flv; and peixe_en-850.flv, and peixe_gl-809.flv.

Listing 5 Example of textual/audiovisual interface tagging in the Veiga Corpus.

```
<tu>
  <tuv xml:lang="en"><seg><s n="848" d="01:07:32,351" a="01:07:34,342"/>We
       play our cards right,<l/>we could end up with... <s n="849" d="01
      :07:34,431" a="01:07:36,023"/>two million pounds of tobacco<l/>to
      spend it for us.</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="808" d="01:07:33,271" a="01:07:36,946"/>
      Podemos gastar 2 millóns en tabaco<l/>e pósters de Pamela Anderson.</
      seg></tuv>
</tu>
<tu>
  <tuv xml:lang="en"><seg><s n="850" d="01:07:36,511" a="01:07:39,025"/>I
      meant to get someone<l/>to spend it for us.</seg></tuv>
  <tuv xml:lang="gl"><seg><s n="809" d="01:07:37,071" a="01:07:39,539"/>
      Buscaremos alguén<l/>que o gaste por nós.</seg></tuv>
</tu>
```

4 Results

Since 2003, the SLI at the University of Vigo offers the possibility of searching and browsing the CLUVI parallel corpora online `http://sli.uvigo.es/CLUVI/`. The parallel corpora managed by the web application are stored in the XML CLUVI specification, whereas the searching and browsing tool designed in PHP was specifically created to carry out bilingual searches in tagged texts that are conformant to this specification. This search application allows for very complex searches of isolated words or sequences of words, and shows the bilingual equivalences of the terms in context, as they appear in real and referenced translations. Due to copyright issues, it returns a maximum of 1,500 hits only. Users can search terms in either language of the corpus, although it is also possible to carry out true bilingual searches, that is, to simultaneously search one term in each of the languages present in the parallel corpus. Search results are displayed in a parallel fashion as a list of translation units. In addition, the multimedia version of the Veiga Corpus has an improved browsing functionality that enables users to stream the subtitled video clips (stored as FLV files) via the open source video player Flowplayer[1], which allows embedding FLV video files into the results page. This multimedia-aware interface is already available for public consultation at`http://sli.uvigo.es/CLUVI/vmm.html`.

The coverage and size of the multimedia Veiga are shown in Table 1, where Words_EN means 'number of words in English', TUs stands for 'number of translation units' (roughly, English-Galician equivalent sentences), and Sub_EN means 'number of subtitles in English' (i.e. number of video clips subtitled in English).

4.1 Drawbacks

As [3] pointed out, we need to access texts in an in vivo form that provides access to audio and video tracks and maintains their relationship intact, because a major part of the way in which a film text makes its meaning is precisely through the synchronization between visual

[1] `http://flowplayer.org`

Table 1 Coverage and size of the Multimedia Veiga Corpus.

Film title	TUs	Words_EN	Words_GL	Sub_EN	Sub_GL
Afterglow	1300	6839	5811	911	1051
Babel	1208	6158	3925	941	777
Blood and Wine	1027	4353	3266	887	887
Bride of Chucky	1016	4750	4130	858	858
City of Industry	782	3391	3023	464	631
Earthlings	621	6822	6692	861	1097
Faces	2230	11631	7403	1566	1536
Fury	1481	9498	7268	1207	1189
If...	1093	5341	4464	869	879
Napoleon	6815	6075	1470	1212	933
Paris-Texas	1486	8549	5605	1138	1182
Punishment Park	1684	10140	8035	1494	1473
Shooting Fish	1636	9176	6699	1191	1146
	22379	92723	67791	13599	13639

and audio resources. However, before pinning any hope on the potential strengths of the Veiga corpus of subtitles, we had better begin by acknowledging its most visible weaknesses.

The first drawback is the small size of the corpus. To our credit, we must say that only two people are currently working on the project, and that a further extension is envisaged to also include TV broadcast films and other languages in a near future. A larger corpus would no doubt provide evidence of a wider range of phenomena, which may positively impact the reliability of any subsequent research based on the Veiga data. However, size is not necessarily a guarantee of representativeness. Moreover, in some circumstances, small, field-specific corpora may be equally useful for the investigation of particular phenomena.

The second limitation is the heterogeneous origin and authorship of the translated subtitles, which may accordingly call for different approaches to data observation and foreseeably arise the question of translation (and corpus) quality. As previously mentioned, the Galician subtitles were produced for DVD, cinema and Internet distribution. Specifically, seven of them are DVD-catered subtitles, that is, they are likely to be made by professional translators and to have undergone a quality control check. Fourteen of them were produced for the cinema. Notably, they were screened at various film series organized by a Galician film association. In this case, the subtitlers are mostly volunteers (non-paid translators), and they would lie halfway between the previous (professional) and the next (amateur) kind of translators. The other three sets of subtitles are instances of a new genre of subtitling in Spain (and other countries) that is properly known by the name of amateur subtitling and described by [5] as a practice 'undertaken by non-professionals and governed by dramatically different constraints than professional subtitling'. Often, the end result 'is conditioned by how much the subtitle producer has heard and understood from the original language', which is 'likely to result in a multitude of mistakes and misinterpretations'. Nonetheless, quality was not a criterion that we took into consideration when compiling our corpus.

And a third limitation is the above-mentioned processing and editing tasks involved in the process of creating a multimedia parallel corpus, which are still, and in spite of the technological advances, very time consuming. Consequently, no matter what purpose corpus users are driven by when searching the Veiga, they must keep these limitations in mind at all times.

4.2 Applications

Notwithstanding the aforementioned, our multimedia corpus may still serve a number of potential uses and purposes. First, it may be exploited as a reservoir of examples, offering researchers and scholars a database to analyse the different strategies and procedures used in both interlingual and intralingual subtitling and helping them substantiate their theoretical assumptions with practical evidence. From a pedagogical perspective, the Veiga features suggest that it could be used for different purposes in various learning settings, ranging from general language courses dealing with pronunciation, register, collocations, and other features of oral and written discourse, to specialised courses in audiovisual translation (AVT) with a focus on interlingual and intralingual subtitling ([17]). As put forward by [16], it is important that AVT teachers provide trainees with authentic material for contrastive analysis of both source (original) and target (translated) texts. Concerning language learning, the use of assorted 'real' texts, and particularly intralingual subtitles for L1 learning and interlingual subtitles for L2 learning, is likely to increase students' motivation and cultural awareness, although careful selection, adaptation and designing of teaching materials and activities coupled with adequate teacher guidance need to be in place. At the same time, the Veiga multimedia corpus may also prove a useful e-learning tool, since it would provide students with the possibility of exploring textual properties while listening to and watching film clips, which can be played and stopped at will ([2]), thus promoting autonomous learning. Finally, professional practitioners could also benefit from the possibility to access a collection of ready-made subtitles, where they can look at how other colleagues solved particular subtitling challenges.

As we have just discussed, the limited size of the corpus and the hybrid nature of the translated subtitles do not allow for generalizations about the practice of intralingual and interlingual subtitling. In fact, further distinctions could be made based on the particular genre of the audiovisual texts (featured films, documentaries, children's films...) and the product distribution medium. Nevertheless, corpus users should keep in mind that our core aim is to provide a tool that may serve not only researchers, but also practitioners and teachers to illustrate particular aspects of subtitling, and no regard is given to issues of corpus quality and representation.

On one hand, technical issues such as subtitles' display on screen (number of lines, alignment, position, colour, dialogue markers) and duration (in and out times, delay, shot change, synchronization) can be easily looked at in the Veiga corpus. The subtitling practice is rather heterogeneous and it can vary substantially from one audiovisual program, company and country to another ([6]). And although some efforts have been made to come up with a set of conventions or harmonized guidelines, subtitling tradition seems to determine what current practice is in each particular language/culture.

On the other hand, both interlingual and intralingual subtitles are condensed versions of the original audiovisual text. Subtitling usually involves the selection of linguistic material, forcing subtitlers to make decisions on what is important and what is seemingly superfluous or even redundant. Redundancy indeed is a very important concept in subtitling, because the information not given by the subtitles may be supplied by other elements present in the audiovisual text: the image and/or the sound ([4]). Reduction, however, is often achieved through the omission of information or by sacrificing interpersonal meaning ([12]). The Veiga multimedia corpus of subtitles not only places subtitles and the original audiovisual text in juxtaposition with one another, but also brings the English intralingual subtitles face to face with the Galician translated subtitles, allowing users to explore phenomena such as cohesion and condensation, which are deeply rooted in the semiotics of subtitling.

5 Conclusions

We have presented the Veiga multimedia corpus of English-Galician subtitles, an ongoing project that aims at echoing the general idea put forth by some authors, who claim to transcend the traditional only-text approach to corpus design and call for the need to build multimedia corpora that better reflects the polisemiotic aspects of film discourse and subtitling. After describing its content, we have raised some issues regarding corpus data and design, particularly the 'in-betweenness' of the data sets, which are not exactly translations of each other. In section 3 we briefly explain the methodology used to build the corpus, which partly mirrors that of all the other CLUVI's subcorpora, with the main difference being that the Veiga undergoes further processes in order to account for its audiovisual nature and to better cater to the principles of usefulness and usability. Then we sketch the main features of the corpus query system, which offers the option to stream the videoclips containing the subtitles. This is followed by an account of some obvious limitations of the corpus, such as size and technology constraints, that we hope to resolve in the near future. And finally, we have pointed at various areas in subtitling practice, research and education where the Veiga multimedia corpus could be of most value.

Right now, we are conducting some research concerning the potential applications of the Veiga multimedia in various pedagogical settings. As we move forward, we expect to improve the corpus in general and to illustrate its usage in other scenarios as well.

References

1 Mona Baker. Corpora in translation studies: An overview and some suggestions for future research. *Target*, 7(2):223–243, 1995.

2 Anthony Baldry. The role of multimodal concordancers in multimodal corpus linguistics. In Terry D. Royce and Wendy Bowcher, editors, *New Directions in the Analysis of Multimodal Discourse*, pages 173–193, Mahwah, NJ, 2006. Lawrence Erlbaum Associates.

3 Anthony Baldry and Christopher Taylor. Multimodal concordancing and subtitles with MCA. In Alan Partington, John Morley, and Louann Haarman, editors, *Corpora and Discourse*, pages 57–70, Bern, 2004. Peter Lang.

4 Francesca Bartrina. Teaching subtitling in a virtual environment. In Jorge Díaz Cintas and Gunilla Anderman, editors, *Audiovisual translation: Language Transfer on Screen*, pages 229–239, London, 2009. Palgrave Macmillan.

5 Jorge Díaz Cintas and Gunilla Anderman, editors. *Audiovisual translation: Language Transfer on Screen*. Palgrave Macmillan, London, 2009.

6 Jorge Díaz Cintas and Aline Remael. *Audiovisual Translation: Subtitling*. St. Jerome, Manchester, 2007.

7 Xavier Gómez Guinovart. A hybrid corpus-based approach to bilingual terminology extraction. In Isabel Moskowich-Spiegel Fandiño and Begoña Crespo, editors, *Encoding the Past, Decoding The Future: Corpora in the 21st Century*, pages 147–175, Newcastle upon Tyne, 2012. Cambridge Scholar Publishing.

8 Xavier Gómez Guinovart, Eva Díaz Rodríguez, and Alberto Álvarez Lugrís. Aplicacións da lexicografía bilingüe baseada en córpora na elaboración do Dicionario CLUVI inglés-galego. *Viceversa: Revista Galega de Traducción*, 14:71–87, 2008.

9 Xavier Gómez Guinovart and Elena Sacau Fontenla. Parallel corpora for the Galician language: building and processing of the CLUVI (Linguistic Corpus of the University of Vigo). In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC'04)*, pages 1179–1182, 2004.

10 Xavier Gómez Guinovart and Alberto Simões. Parallel corpus-based bilingual terminology extraction. In *Proceedings of the 8th International Conference on Terminology and Artificial Intelligence*, Toulouse, 2009. Université Paul Sabatier.

11 Xavier Gómez Guinovart and Alberto Simões. Translation dictionaries triangulation. In Carmen García Mateo, Francisco Campillo Díaz, and Francisco Méndez Pazó, editors, *Proceedings of FALA2010: VI Jornadas en Tecnología del Habla - II Iberian SLTech Workshop*, pages 171–174, Vigo, 2010. Universidade de Vigo.

12 Josélia Neves. Interlingual subtitling for the deaf (and hard-of-hearing). In Jorge Díaz Cintas and Gunilla Anderman, editors, *Audiovisual translation: Language Transfer on Screen*, pages 151–169, London, 2009. Palgrave Macmillan.

13 Yves Savourel and Arle Lommel. TMX 1.4b Specification. Technical report. Localisation Industry Standards Association. <`http://www.gala-global.org/oscarStandards/tmx/tmx14b.html`>, 2005.

14 Alberto Simões and Xavier Gómez Guinovart. Terminology extraction from English-Portuguese and English-Galician parallel corpora based on probabilistic translation dictionaries and bilingual syntactic patterns. In António Teixeira, Miguel Sales Dias, and Daniela Braga, editors, *Proceedings of the Iberian SLTech 2009 - I Joint SIG-IL/Microsoft Workshop on Speech and Language Technologies for Iberian Languages*, pages 13–16, Porto Salvo, 2009. Designeed.

15 Alberto Simões, Xavier Gómez Guinovart, and José João Almeida. Distributed translation memories implementation using WebServices. *Procesamiento del Lenguaje Natural*, 33:89–94, 2004.

16 Cristina Valentini. A multimedia database for the training of audiovisual translators. *The Journal of Specialized Translation*, 6:68–84, 2006.

17 Federico Zanettin, Silvia Bernardini, and Dominic Stewart, editors. *Corpora in Translator Education*. St. Jerome, Manchester, 2003.

Investigating the Possibilities of Using SMT for Text Annotation

László J. Laki[1]

1 **MTA-PPKE Language Technology Research Group –**
Pázmány Péter Catholic University, Faculty of Information Technology,
50/a Práter street, Budapest, 1083, Hungary
laki.laszlo@itk.ppke.hu

Abstract

In this paper I examine the applicability of SMT methodology for part-of-speech disambiguation and lemmatization in Hungarian. After the baseline system was created, different methods and possibilities were used to improve the efficiency of the system. I also applied some methods to decrease the size of the target dictionary and to find a proper solution to handle out-of-vocabulary words. The results show that such a light-weight system performs comparable results to other state-of-the-art systems.

1998 ACM Subject Classification I.2.7 Natural Language Processing

Keywords and phrases SMT, POS-tagging, Lemmatization, Target language set, OOV

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.267

1 Introduction

A wide spectrum of opportunities has been opened due to the fast development of information technology in almost all disciplines. This evolution could be detected on the field of computational linguistics as well. Processing of huge text materials has become easier, even the efficiency of these systems is increasing. Marking texts with syntactic and/or semantic information, or the morphological analysis of the language are really important tasks for computational linguistics. The task of part-of-speech (POS) tagging has not yet been perfectly solved, even though several systems have been implemented to achieve better results to this complex problem. The most popular ones are based on machine learning, in which the rules recognized by the systems themselves are based on different linguistic features. Further difficulties lie in determining the features, since these could be hardly formulated. Instead statistical machine translation (SMT) systems are able to recognize essential translation rules and features without any previous linguistic knowledge [8].

Based on this assumption the application of SMT systems for text analysis could be successful. With the help of the standard frameworks and tools [11, 10, 15] used for statistical machnie translation tasks, it is straightforward to handle complex POS structures. In this work I examine the applicability of these systems to solve the task of part-of-speech disambiguation and lemmatization.

2 Basic Concepts

2.1 Statistical Machine Translation

Statistical machine translation (SMT) is a method of statistical language processing usually applied to translation between human languages [12]. It has a great advantage over rule-

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 267–283

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

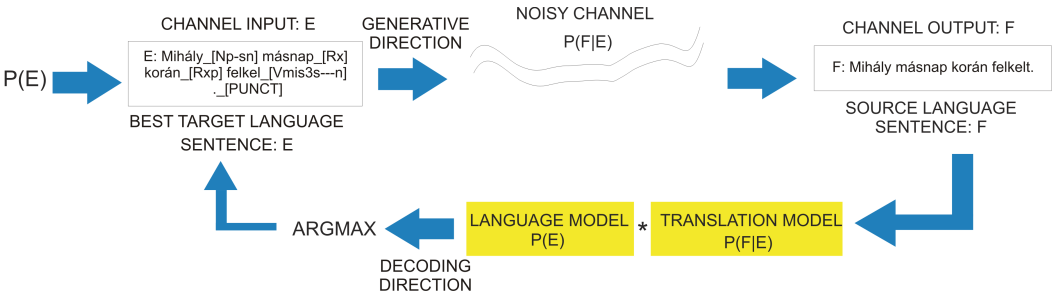


Figure 1 Representation of SMT method.

based translation: only a bilingual corpus is needed to set up the training set of the system, but knowledge about the grammar of the language is not required to create the architecture of a baseline SMT system. The system is trained on this corpus, from which statistical observations and rules are determined.

The phrase, which we want to translate – i.e. the source sentence, is the only certain thing we know prior to the translation. Therefore, the system is defined as a noisy channel [8]. A set of target sentences are passed through this channel and the output of the channel is compared with the source sentence.

This process can be formulated by Bayes' theorem as the product of two stochastic variables called language model and translation model. The result of the translation is the phrase, which provides the most appropriate match with the source sentence. In addition this match is a probability that could be determined from the language model $p(E)$ and the translation model $p(F|E)$ according to the following formula [12, 8]:

$$\hat{E} = \underset{E}{argmax}\ p(E|F) = \underset{E}{argmax}\ p(F|E) * p(E) \tag{1}$$

2.2 Part-of-Speech Disambiguation

POS-tagging is the process of assigning a part-of-speech or other lexical class marker to each word in a corpus. The input to a tagging algorithm is a string of words and a specified tagset. The output is a single best tag for each word [9].

However POS tagging is harder than just having a list of words and their part of speech, because some words can represent more than one part of speech depending on the context. A simple example is the Hungarian word "vár" that has two different meanings – "wait" and "castle" – with different part of speech, i.e. verb and noun.

Most solutions apply analysis of the text based on pre-specified rule systems. The disadvantage of these methods is the huge cost of the creation of rules. Other frequently used approaches are based on machine learning, in which there are also some kind of rules used, however these are not of the same kind as linguistics rules, but are developed by the algorithms themselves based on relevant features. Further difficulties lie in the determining of these features, since these could be hardly formulated. It is very hard to determine and create a complete rule system that covers all the linguistic features and which can be processed by a computer.

2.3 Lemmatization

In computational linguistics, lemmatization is the algorithmic process of determining the lemma for a given word. The lemma is the dictionary form of a word. Since the process may involve complex tasks such as understanding context and determining the part of speech of a word in a sentence (requiring, for example knowledge of the grammar of a language), it is a hard task to implement a lemmatizer for a language like Hungarian, because words appear in several inflected forms.

Several implementations exist to solve this problem (e.g. HUMOR [18]), but most of them are based on complex methods of preprocessing that separate this task from that of POS-tagging, though these are very strongly related.

3 POS-tagging as SMT Problem

As described above both POS-tagging and lemmatization could involve huge amomunt of resources and complexity especially when applied to more complex languages, like Hungarian. In English a word can only have a limited number of forms, however in agglitunating languages this number is several orders of magnitude higher. Each affixum of a word contains some morphological information and also might produce a change in the lemma of the word. Therefore even more sophisticated algorithms are necessary to handle such a behaviour properly. These considerations deduce the application of the methods of statistical machnie translation to POS-tagging. In such a case POS-tagging is considered as a translation between sentences (F) and their tagged versions ($\hat{E}$) [12, 8]:

$$\hat{E} = \underset{E}{argmax}\ p(E|F) = \underset{E}{argmax}\ p(F|E) * p(E) \tag{2}$$

In equation $p(E)$ is the language model of the POS tags and $p(F|E)$ is the translation/analysis model. The source sentence is a set of phrases that are to be translated to tags. POS-tagging is a simpler task for an SMT system than the translation between natural languages, since the change of word order in a sentence is not required. The number of elements in the source and target side is equal; the system does not make item insertion or deletion [14, 5]. That is why an SMT system might be applied successfully to solve the task of POS-tagging.

Even though POS-tagging would not require to use the sophisticated tools of an SMT framework, these might have an effect on handling the deeper structure of the sentence or the dependency and the context of the words that is to be annotated.

Handling and analyzing out-of-vocabulary words that are not included in the training set (OOV words) has a significant influence on the success of a POS-tagging system. The type frequency of OOV words might vary in different languages. In English an OOV word will probably be a proper noun. In some other languages – such as Hungarian – OOV words would equally be nouns or verbs as well. This is due to the practically infinite number of word forms that might appear, thus it is impossible to have a corpus containing all forms of each word.

The benefit of this method is that the system is able to find rules without defining feature sets and it could do POS-tagging and lemmatization simultaneously. Another advantage is that it is a language independent method, where the the performance of the system only depends on the quality of the bilingual corpus used to train the system. Though my purpose was only to test the system on Hungarian, in later works it might be extended to other languages easily.

3.1 Coding Systems of POS Tags

There are three types of coding systems used for morphological coding in Hungarian language; namely KR [13, 4] , HUMOR [18] and MSD [4] systems. The morphology of Hungarian language was taken into consideration, when KR-coding system was developed. It is basic syntax however is language-independent. The HUMOR morphological coding system [18] is based on unification. Different labels are used as tags, based on the capability of fusing morphemes with others. Different labels could allow or contradict each other. One word can be built up from morphemes, for which the labels do not exclude each other. The MSD-coding system [4] was used to analyze the corpus from morph syntactical point of view. The MSD-coding system is applied for coding different attributes of words – mainly morphologically –, and could be used for most European languages. The morph syntactical attributes of words – for example type, mood, tense, number, person etc. – are represented as a character set. In the place of attributes, which are missing or not interpreted in natural languages, character '-' is used. At the first position the main POS categories of words are available. In this work I use MSD coding only for the reason that the only available tagged corpus of Hungarian is provided with MSD codes.

3.2 Framework

3.2.1 Corpus

In this study the Szeged Corpus 2 [3] was used as parallel corpus, which was created by the Language Technology Group of the University of Szeged. This XML-based database contains both plain texts and their POS annotated version using the MSD-coding system. The advantage of the corpus is that it was manually corrected; therefore it is a highly accurate data set. Further benefit is that it is general and not topic-specific. In order to have such a reliability, it is a rather small corpus containing 1.2 million words, which cover 155.500 different word-forms and 250.000 inter-punctuation signs. In contrast to natural language translation, where this size is unusably small, it is not such a relevant problem for POS-tagging, since the target language has a very limited vocabulary compared to any natural languages. For testing the system, 1500 randomly selected sentences of the corpus were used.

3.2.2 Training and Decoding

Several methods of obtaining information from parallel corpora have been studied. Finally, I decided to use IBM models, which are relatively accurate, and the used algorithm was adaptable to the task. Based on these findings I decided to use the MOSES framework [11, 10], which implements the above mentioned IBM models. This system includes algorithms for the pre-processing of the parallel corpus, for the setup of translation and language models and for the decoding and the optimization to the BLEU score [17]. An improved SMT system framework is JOSHUA [15], which not only applies word- or phrase-level statistics, but takes into account the morphological characteristics of the language. Chomsky's generative grammars are used to solve this task. The languages, which could be described with grammatical rules, belong to the class of regular languages and context-free grammars (CFG). The advantage of the JOSHUA system is that it is able to translate between these CFG rules in such a way, that rules can be specified for both source and target languages, furthermore the probability of the transformations into each other.

3.2.3 Evaluation

To evaluate the efficiency of traditional SMT systems an automatic method is used. The BiLingual Evaluation Understudy [17] – BLEU score. The essence of this method is that the translations are compared with the reference sentences of the test set. BLEU score is calculated both to each n-gram lengths, and to a comulated average as well. Since POS-tagging is a one-to-one mapping between tags and words the most relevant measure gains from the case of 1-grams. Since BLEU score is not the usual method of evaluating a POS-tagger and lemmatizer, I also calculated the accuracy of the system to be able to compare the efficiency to other systems. This evaluation was used in sentence and in token level as well.

4 Baseline System

In the following sections I describe each versions of the system and their results.

In the first test the system was trained with unmodified corpus. The source language corpus was created from the tokenized sentences without annotation. The target language corpus contained the lemmatized words and their POS tags. Table 1 displays the results of the system (SMT_Zero) for each decoder.

Table 1 Performance of the system SMT_Zero.

System	BLEU score	Precision
MOSES	98.35%	90.29%
JOSHUA	97.28%	91.02%

The relatively low results revelad some drawbacks of the applied method. The most relevant problem arises from the structure of the corpus. In the annotated corpus the lemma of each word is connected to the morphological tags. In the case of multi-word phrases (for example: multi-word proper names, verb phrases) the tag either joins to the last word of each phrase or stands after the last word. The lack of the marks of related phrases makes false probability values in the translation model. Consequently the system assigns a random tag after proper names which made the results even worse.

4.1 Elimination of Single POS Tags

To solve the problem of missing or unjoined tags, all independent tags were joint to the previous word (system SMT_NoSinglePOS. The average results show a slight improvement as displayed in table 2.

Table 2 Performance of the system SMT_NoSinglePOS.

System	BLEU score	Accuracy
MOSES	98.40%	90.80%
JOSHUA	97.25%	90.72%

Though the change in the BLEU score is not significant, the accuracy of the system is increased with 0.5-0.6 percent (in the case of MOSES, which proved to be more efficient

than JOSHUA for all the later methods as well). This is due to the fact, that unnecessary elements are not included in the translation.

4.2 Handling of Multiple-word Phrases

The toughness of this task is that the system analyses only words, therefore each part of a phrase is tagged separately. The goal is to handle these multi-word phrases as one unit. Most of these phrases are recognized as named entities. Thus the system was improved by joining multi-word phrases in the source side corpus. Table 3 displays the result of this system called SMT_Baseline1.

Table 3 Performance of the system SMT_Baseline1.

System	BLEU score	Accuracy
MOSES	98.49%	91.29%
JOSHUA	97.31%	91.07%

Numerically from the 1500 sentences of the test set 506 were absolutely correct and 994 sentences had mistakes. At first sight this is a quite strange rate, but if we see the result at token level (24557 correct and 2343 incorrect) we got much better evaluation. Table 3 shows that joining related words increased the accuracy of the system; however the BLEU score was lower than the result of the previous system.

The evaluation revealed that the wrongly annotated sentences could be divided into two categories. The first is when the system does not perform the translation, but returns the original word (1697 pieces). In most cases these words are not included in the corpus, so they could not be in the translation model. If the decoder does not find an entry in the translation model, it keeps the original form of the word in the translation, in my case that is the word form instead of a POS-tag. The other type of error is the case of incorrect annotations (646 pieces). Two subcategories can be distinguished in this case. The first is when the system can find correctly the main POS tag of the word but it fails in the further analysis; secondly when even the main POS tag is incorrect. Table 4 shows an example of the output of the system SMT_Baseline1.

Table 4 An example from the output of the system SMT_Baseline1.

System	Translation
Simple text:	ezt a lobbyerőt és képességet a diplomáciai erőfeszítéseken kívül mindenekelőtt a magyarországi multinacionálisok adhatnák .
Reference annotation:	ez_[pd3-sa] a_[tf] lobbyerőt_[x] és_[ccsw] képesség_[nc-sa] a_[tf] diplomáciai_[afp-sn] erőfeszítés_[nc-pp] kívül_[st] mindenekelőtt_[rx] a_[tf] magyarországi_[afp-sn] multinacionális_[afp-pn] adhat_[vmcp3p—y] ._[punct]
SMT annotation:	ez_[pd3-sa] a_[tf] lobbyerőt és_[ccsw] képesség_[nc-sa] a_[tf] diplomáciai_[afp-sn] erőfeszítéseken kívül_[st] mindenekelőtt_[rx] a_[tf] magyarországi_[afp-sn] multinacionális_[afp-pn] adhat_[vmcp3p—y] ._[punct]

This system is considered as an SMT_Baseline1 system (the traditional baseline – i.e. rendering the most probable tag – for POS-tagging is used in later sections of the paper).

5 Decreasing the Size of Target Vocabulary

5.1 With only POS Disambiguation

An essential part of any statistical methods is the number of training instances, in this case the size of the corpus. Since I used the biggest available tagged corpus for Hungarian, there is no way to achieve better results with increasing the size of the training set. However another possibility is to decrease the vocabulary in order to have a relatively bigger training set with decreasing the complexity of the annotation task. One way to achieve this is if the simple text is translated to the "language" of the POS tags only without performing lemmatization. The size of the target dictionary was reduced from 152 694 to 1128 elements. This number of tags is much fewer than the number of Hungarian words; therefore a relatively accurate system could be built from a smaller corpus. On the other hand, if the lemmas are left out from the annotation and the translation is made only to the set of tags, the order of the morphemes in the sentence will be much more weighted in the translation and language models as well. The result of this system (called SMT_OnlyPOS later SMT_Baseline2) is displayed in table 5.

Table 5 Performance of the system SMT_OnlyPOS.

System	BLEU score	Accuracy
MOSES	96.22%	91.46%
JOSHUA	92.17%	91.09%

The system achieved worse BLEU score compared to the SMT_Baseline1 system, but the accuracy is better. Numerically there are 518 correct sentences and 982 incorrect ones that means 0.8% improvement compared to the SMT_Baseline1 system. Regarding the tokens, 24603 correct and 2297 incorrect ones were counted; that is 0.17% improvement. The number of not annotated words (1699 item) did not change; however the number of incorrect POS tags appeared only in 598 cases.

Thus the main improvement of the quality at this stage results from decreasing the number of incorrect POS tags. Deeper evaluations prove that besides this improvement (48 items), in some cases the previously correct tags failed. These failures were caused mainly by mixing adverbs with conjunctions or conjunctions with demonstrative pronouns. Output of system SMT_OnlyPOS showed in table 6.

5.2 With Simplifying POS Tags

Another method to decreace the size of the target language is to simplify the resulting POS-tags to include less morphological information. This method reduces the complexity of the system, but consequently the depth of analysis will be decreased as well. Only the main POS tags – the first characters of MSD codes – were used. This way the target dictionary consists of only 14 elements. The result of the system (called SMT_MainPOS) is displayed in table 7.

Table 6 An example from the output of the system SMT_OnlyPOS.

System	Translation
Simple text:	ezt a lobbyerőt és képességet a diplomáciai erőfeszítéseken kívül mindenekelőtt a magyarországi multinacionálisok adhatnák .
Reference annotation:	[pd3-sa] [tf] [x] [ccsw] [nc-sa] [tf] [afp-sn] [nc-pp] [st] [rx] [tf] [afp-sn] [afp-pn] [vmcp3p—y] [punct]
SMT annotation:	[pd3-sa] [tf] lobbyerőt [ccsw] [nc-sa] [tf] [afp-sn] erőfeszítéseken [st] [rx] [tf] [afp-sn] [afp-pn] [vmcp3p—y] [punct]

Table 7 Performance of the system SMT_MainPOS.

System	BLEU score	Accuracy
MOSES	90.35%	92.20%

The evaluation results fit to the previously seen tendency; i.e. there is a decrease in BLEU score, but the accuracy of the system increased. 553 sentences were correct and 947 incorrect; it means 2.3% improvement compared to the system SMT_OnlyPOS and 3.1% to the SMT_Baseline1 system. 24803 tokens were correctly tagged and 2097 incorrectly; this is 0.77% improvement to SMT_OnlyPOS and 0.84% compared to the SMT_Baseline1 system. A sample from the output is shown in table 8.

Table 8 An example from the output of the system SMT_MainPOS.

System	Translation
Simple text:	ezt a lobbyerőt és képességet a diplomáciai erőfeszítéseken kívül mindenekelőtt a magyarországi multinacionálisok adhatnák .
Reference annotation:	p t x c n t a n s r t a a v p
SMT annotation:	p t lobbyerőt c n t a erőfeszítéseken s r t a a v p

5.3 Conclusion

The above results are very promising, as the accuracy of the system is over 90% even if it was trained on a small-sized corpus. We have to note, however that the size of the dictionary of system SMT_OnlyPOS (1128 tags) is much smaller compared to that of the SMT_Baseline1 (152 694 tags), but the accuracy increased only with 0.17%. Furthermore in system SMT_MainPOS where 14 tags were used, the accuracy increased only with 0.88%. This 0.88% increase is not proportional to the significant information loss that is caused by the size minimization of the dictionary in system SMT_MainPOS. Furthermore despite the positive changes in the results, the above systems are still not able to tag OOV words (1698 cases). Consequently the next step should be to find a proper solution to handle words not included in the training set.

6 Handling OOV Words

The most obvious solution to reduce the number of OOV words is to increase the size of the corpus, so that all word forms would appear in it. Moreover it is important to have several occurences of each token in order to have a reliable statistics. Due to the agglutinative nature of Hungarian language, one stem could have many forms caused by the affixes; that is why an extremely large corpus would be needed to have all forms with the appropriate weight. This is an impossible requirement by itself, even more if a manually tagged corpus is expected. To eliminate this situation, I applied a method in which the system tries to find the appropriate tag for an unknown word based on the analyses of its context. In this capture the lemmatization is left out. All results will be compared with system SMT_OnlyPOS (from now SMT_Baseline2).

6.1 In the Original Text

To examine the characteristics of frequent OOV words, a further investigation is needed during training and decoding. My basic assumption is to infer OOV POS-tags from the context. Though this is quite a simple method, however the complexity of the problem can also be reduced by limiting the possible POS-tags of an unknown word to some of the most probable ones. Thus at decoding time, the system has to choose only from these few tags.

To eliminate this problem I applied Guillem and Joan Andreu's method [5]. To achieve good results for Hungarian I used their results for English with some changes. A dictionary is created from words whose frequency in the training set is over a certain threshold value. The word frequency is calculated from the corpus. The words not included in this dictionary are changed to an optional expression (in this case "UNK"). The basic idea of the method is to change the less frequent words to the string "UNK".

Since OOV words are included in just a few word classes, therefore I assume that the annotation of the context of each OOV word is very similar. The SMT system performs the translation based on phrases, therefore the context of words and tags is taken into consideration already. By replacing the less frequent words to symbol "UNK", the annotation of the environment of these phrases will be more significant. Consequently the system can identify the POS tag for symbol "UNK".

The key question is the appropriate threshold value selection, since it determines the number of "UNK" symbols in the corpus. On one hand if this value is too high, too many tokens will be changed to "UNK" symbol; the probability of this symbol increases, therefore we will not receive correct annotation. On the other hand if the threshold is too small, too many rare words will be included in the dictionary causing that the advantage of the method could not be exploited sufficiently.

Therefore the system was trained with more threshold values (2, 4, 6, 8 and 10) to find the most appropriate one resulting in the best improvement of accuracy. The results of this system (SMT_OOV_token) can be found in table 9.

If the threshold is 1 the table gives us the result of SMT_Baseline2 system. That means none of the words were replaced with symbol "UNK". In the last column we can see the accuracy of the systems for each threshold value. For example: threshold 2 means that all words that appear in the corpus less than two times were changed to symbol "UNK". The second column of the table shows the percentage of words in the training set (of size 1 459 288) added to the dictionary. For example: in the case of threshold 2 almost 60% of the words became OOV words. The third column of the table contains the percentage of the words left original in the corpus.

Table 9 Performance of the system SMT_OOV_token.

Threshold value	Rate of words in the dictionary	Rate of words in the corpus	System's accuracy
SMT_Baseline2	100%	100%	91.46%
2	39.16%	93.87%	93.13%
4	19.18%	89.22%	90.40%
6	12.99%	86.48%	88.41%
8	9.96%	84.51%	87.07%
10	8.09%	82.92%	85.97%

From the above results it is straightforward that in the case of threshold value 2 the system achieved significant improvement compared to any of the previous ones. Only 38 words were not annotated against the 1697 in system SMT_Baseline1. If the threshold value is raised, it leads to a decrease in the accuracy of the system.

During deeper evaluation it turned out that this accuracy decrease is due to the lower rate of original words in the corpus (only small number of words are in the dictionary). In the case of threshold 2, symbol "UNK" was used for 6.13% of the words in the training set. This rate is 92% in the case of threshold being 10. This tendency matches with the theorem of Zipf's laws [21]. We have to note the 85.96% accuracy at threshold value 10. This result is quite good despite that only 8.09% of the training set was added to the dictionary. Table 10 shows an example from the output of the system SMT_OOV_token in the case of threshold 8.

Table 10 An example from the output of the system SMT_OOV_token.

System	Translation
Simple text:	ezt a **unk** és **unk** a diplomáciai **unk** kívül mindenekelőtt a magyarországi **unk unk** .
Reference annotation:	[pd3-sa] [tf] [x] [ccsw] [nc-sa] [tf] [afp-sn] [nc-pp] [st] [rx] [tf] [afp-sn] [afp-pn] [vmcp3p—y] [punct]
SMT annotation:	[pd3-sa] [tf] [nc-sa] [ccsp] [vmis3p—y] [tf] [afp-sn] [nc-pn] [st] [rx] [tf] [afp-sn] [nc-pn] [nc-sa—s3] [punct]

6.2 In Case of Lemmas

From the results of table 10 it can be seen that if the threshold value is too high, too many of the words become "UNK". Due to the agglutinative features of Hungarian language the original text contains different forms of nouns, verbs and adjectives of the same stem. This is the reason that the number of these different forms is under the threshold. Consequently in most cases nouns, verbs and adjectives are also replaced with "UNK" in the sentences of the corpus, which makes the decreases the accuracy of the system. My goal was to reduce the number of symbol "UNK" with replacing only really rare words in the text. Therefore the threshold was determined based on the frequencies of the lemmas and not on different word forms. The results of this system (called SMT_OOV_lemma) can be found in table 11.

Table 11 Performance of the system SMT_OOV_lemma.

Threshold value	Rate of words in the dictionary	Rate of words in the corpus	System's accuracy
SMT_Baseline2	100%	100%	91.46%
2	70.00%	96.58%	92.57%
4	56.64%	94.50%	92.25%
6	50.18%	93.17%	91.81%
8	45.81%	92.13%	91.48%
10	37.08%	88.47%	91.10%

The results proved that calculating the threshold based on lemmas makes much fewer number of words to be marked as OOV (numerically only 3.42% of the words from the corpus).

Table 12 shows an example from the output of the system SMT_OOV_lemma in the case of threshold 8 similar to table 10. We can see that in this case only two words were replaced to "UNK" against the previous systems so the goal of reducing the number of OOV words was achieved.

We can observe that besides threshold 2, the best result of system SMT_OOV_lemma (92.57%) is worse than in the case of SMT_OOV_token (93.13%). The deep evaluation showed that the number of not annotated words increased (1015 cases) compared to system SMT_OOV_token, and 984 words were incorrectly analyzed.

Table 12 An example from the output of the system SMT_OOV_lemma.

System	Translation
Simple text:	ezt a **unk** és képességet a **unk** erőfeszítéseken kívül mindenekelőtt a magyarországi multinacionálisok adhatnák .
Reference annotation:	[pd3-sa] [tf] [x] [ccsw] [nc-sa] [tf] [afp-sn] [nc-pp] [st] [rx] [tf] [afp-sn] [afp-pn] [vmcp3p—y] [punct]
SMT annotation:	[pd3-sa] [tf] [nc-sa] [ccsw] [nc-sa] [tf] [afp-sn] erőfeszítéseken [st] [rx] [tf] [afp-sn] [afp-pn] [vmcp3p—y] [punct]

6.3 Multiple Thresholds

The above results have alraedy achieved high accuracy results of taggin Hungarian words, but still OOV words are included to several different POS types with quite high probability. In English such OOV words are mostly nouns. To have a more sophisticated method I applied numerical calculation of several threshold values that distinguish different POS-tags fo OOV words.

We can observe that the same thresholds in the above two systems divide the corpus in different proportions. Word forms with frequency values higher than the threshold are included in the dictionary of system SMT_OOV_token.; but if we determine the frequencies based on lemmas – such as in the case of system SMT_OOV_lemma. – the dictionary will contain more words. Thus a certain threshold divides the set of words to three parts. The first set contains the words, which are included in both dictionaries; these words are the

most relevant ones. In the second set we can find those OOV words, which are really rare and were under threshold in both cases. The words, for which the word form is not frequent enough, but their lemma is over the threshold were included into the third set.

I examined the types of OOV words in each set. The results showed that adjectives and other types of OOV words mostly belong to the second category (under both threshold level), while verbs to the third set. Nouns can be found in both category roughly in a similar measure. Based on this observation another system was trained, which is able to distinguish OOV words, if they belongs to the second or third categories. The results of this system (called SMT_OOV_multi) are shown in table 13.

Table 13 Performance of the system SMT_OOV_multi.

Threshold value	System's accuracy
SMT_Baseline2	91.46%
2	93.28%
4	90.65%
6	88.62%
8	87.40%
10	86.15%

The results reflect that the system with threshold 2 achieved the best performance (93.28%) of all the above systems. This improvement is caused by the fact that only 37 words were not analyzed. Furthermore in the case of incorrect analysis – numerically 1772 items – the error occurred mostly during the subanalysis of nouns.

According to the evaluation, the method of using multiple thresholds helped to distinguish adjectives and verbs; therefore lead to the improvement of the system.

6.4 Introducing Postfixes

Based on the results of the previous systems it is straightforward to conclude that using multiple thresholds – three classes – are not enough to separate nouns, verbs and other types of words. Due to the wide range of affixes in Hungarian, one word could have many forms. Different POS types however have characteristic prefixes and postfixes (in case of Hungarian language mainly postfixes). Therefore previous methods were extended to use information based on the last characters of an OOV word to determine the type.

The best method would be to use a morphological analyzer to separate postfixes of a word with different lengths, but one of the purpose of my method is its simplicity, therefore I applyed a simple implementation for this task as well. To continue the idea of the previous sections in this section the last 2, 3 or 4 characters of the original OOV words were joined to the "UNK" symbol. The results of this system (called SMT_OOV_postfix) are shown in table 14.

This system significantly outperforms any of the previous ones. The worst result is better than the result of the SMT_Baseline2. The best result was 95.96% which was achieved with threshold value 2 and 4-character-long postfixes of OOV words. The optimal length of the postfix might depend on the language, nevertheless in Hungarian most of the postfixes are 2 or 3 character long, that is why the system with 3-character postfixes and with the threshold value of 2 is above 95.83%. The slightly higher results in the case of four characters is due to the cumulative behaviour of suffixes.

Table 14 Performance of the system SMT_OOV_postfix.

Threshold value	System's accuracy Number of left characters 2	3	4
SMT_Baseline2	91.46%	91.46%	91.46%
2	95.17%	95.83%	**95.96**%
4	94.17%	95.32%	95.90%
6	93.48%	94.97%	95.73%
8	92.94%	94.70%	95.60%
10	92.61%	94.55%	95.55%

Table 15 shows an example from the output of the system SMT_OOV_postfix in the case of threshold 8 similar to previous ones. We can see that this system made correct annotations for all words of the sentence in contrast to the above ones.

Table 15 An example from the output of the system SMT_OOV_postfix.

System	Translation
Simple text:	ezt a **unk_erőt** és képességet a **unk_ciai** erőfeszítéseken kívül mindenekelőtt a magyarországi multinacionálisok adhatnák .
Reference annotation:	[pd3-sa] [tf] [x] [ccsw] [nc-sa] [tf] [afp-sn] [nc-pp] [st] [rx] [tf] [afp-sn] [afp-pn] [vmcp3p—y] [punct]
SMT annotation:	[pd3-sa] [tf] [nc-sa] [ccsw] [nc-sa] [tf] [afp-sn] [nc-pp] [st] [rx] [tf] [afp-sn] [afp-pn] [vmcp3p—y] [punct]

7 Evaluation and Comparison with Other Systems

There are several freely available part-of-speech taggers, that are used for Hungarian. First I am going to introduce the available tools, then comparing them with the my methods detailed above.

The most commonly known and used tool is HunPos [7, 6] which is an open source Hidden Markov model based disambiguator tool. It is a reimplementation of Brants' TnT [2] system. While TnT is only capable of generating a smoothed bi- uni- trigram contextual model and a unigram lexical model, HunPos generates a smoothed n-gram contextual model and a context sensitive lexical model. Both of them employs a trie based suffix guesser for determining the correct tags for unknown words, and a special lexical model for handling cardinals. Another enhancement of HunPos over TnT is that it is able to utilize a morphological table (MT[1]). HunPos uses the MT for reducing the search space of the decoding algorithm which may increase heavily for unknown words, enabling it to achieve a significantly better accuracy.

PurePos [16] is an open source tool based on HunPos and TnT. In its implementation it keeps the enhancements intruduced by HunPos, and mainly contributes by performing

[1] A MT is a list of words and their possible morphological labels.

a full morphological disambiguation[2]. It has an interface for employing a morphological analyzer, that is used for determining the correct lemma candidates and can also increase its part-of-speech tagging accuracy.

Another well-known system is the OpenNLP toolkit [1] which is an open source natural language processing tool, including a maximum entropy and perceptron based POS-tagger as well. The basis of this tool is the method developed by Ratnaparkhi [19] , that employs context sensitive features in the case of frequent words, and lexical features[3] for rare words (used for handling unknown words).

Besides OpenNLP there are many other tools applying the maximum entropy approach, most of them are able to perform almost state-of-the-art accuracy for several languages. One of them is the Stanford Log-linear Part-of-speech Tagger [20], that uses a dependency network representation in the log-linear framework. Magyarlanc [22] is an NLP toolkit that was developed for IR systems. It contains an adaptation of the Stanford tagger for Hungarian, a tokenizer and a lemmatizer as well. Unfortunately this system is not directly comparable with the others above since it uses its own tagset[4], that is generated by the integrated analyzer.

The previously detailed SMT based approaches are compared with a baseline and several state-of-the-art systems for Hungarian. For the comparison I used the following supervised learning based method: 1) the training algorithm registers the tags and their frequencies for each seen token 2) the tagger assigns the most frequently seen label for a previously seen token 3) in the case of unknown words it assigns the globally most frequently seen tag.

The evaluation was done on the same test set (a portion of the above described Szeged Corpus [3]) for each system.

Table 16 Comparison of part-of-speech tagging accuracy.

System	Token accuracy	Sentence accuracy
Baseline (BL)	89.66%	25.27%
SMT_Baselin2	91.46%	34.53%
SMT_OOV_postfix	95.96%	56.47%
PurePos	96.03%	55.87%
PurePos-MorphTable	97.29%	66.40%
OpenNLP Maxent (ONM)	95.28%	26.00%
OpenNLP Perceptron (ONP)	94.98%	26.67%

Table 16 shows the accuracy of the investigated Hungarian tagging methods. HunPos is not included since it produces exactly the same results as PurePos. PurePos-MorphTable and PurePos denotes PurePos with and without the morphological table while BL is for the baseline system. One can notice that it produces the highest accuracy when it employs morphological analyses, but without this extra information the method described in section 6 (SMT-OOV) is very close to the best one. Even more investigating the per sentence accuracy my method performs significantly better than the plain HMM based one. The accuracy of the OpenNLP maxent based (ONM) and perceptron based (ONP) methods are under the

[2] Full morphological disambiguation is the task of correctly identifying both the POS tag and the lemma.
[3] e,g, at most four character long prefixes and suffixes of the word.
[4] magyarlanc uses a reduced set of the MSD codes.

Table 17 Comparison of full morphological disambiguation accuracy.

System	Token accuracy	Sentence accuracy
SMT_Baseline1	91.29%	33.73%
PP	83.92%	10.00%
PP-MT	84.89%	11.60%

expectations. It is partly because they use features that are developed mainly for English and are not customized for Hungarian.

From Table 17 the advantage of the SMT based approach is clear especially in the case of per sentence accuracy: it outperforms the best-known one. It is important to notice that the high accuracy reported by Orosz and Novák [16] is mainly due to the usage of its integrated analyzer, that makes their tool language dependent. Without this PurePos performs lemmatizing only with a lemma guesser, that guesses the lemma for a word from its suffix and its part-of-speech tag. However the presented SMT based method is language independent and only needs a manually annotated and lemmatized corpus.

Table 18 Qualitative evaluation of the results produced by the SMT-based and the PurePos systems.

SMT			
Corpus frequency		Error type	
normalized logarithmic form	number of pieces	MSD tag form	textual form
-0.7764	41	[vmm]→[OOV]	imperative verb → OOV
-0.9777	12	[rv]→[OOV]	verbal adverb → OOV
-1.0294	20	[vmc]→[OOV]	conditional verb → OOV
-1.2529	40	[vmn]→[OOV]	infinitive verb → OOV
-1.3504	207	[vmi]→[OOV]	indicative verb → OOV
-1.6099	135	[afp]→[OOV]	qualificative adjective → OOV
-2.4850	18	[afp]→[vmi]	qualificative adjective → indicative verb
PurePos			
Corpus frequency		Error type	
normalized logarithmic form	number of pieces	MSD tag form	textual form
-1.2852	60	[pd3]→[tf]	demonstrative pronoun → definite article
-1.6460	19	[mc-]→[ti]	cardinal numeral → indefinite article
-1.9709	19	[rx]→[ccs]	adverb → coordinating conjunction
-2.3228	23	[vmi]→[afp]	indicative verb → qualificative adjective

Table 18 displays a qualitative evaluation of the results produced by the SMT_Baseline2 method of POS-tagging without lemmatization and that of PurePos. It describes the most frequent types of mistakes of each system. The main error types of the SMT-based method are not recognizing some words at all. However it is also clear that it performs a much better result on recognized words, since such expected errors (as in the case of PurePos) of

mistagging a pronoun to an article are not present. This result forecasts the application of a hybrid implementation, where the statistical behaviour would compensate the lackings of a more robust, however limited algorithm.

8 Conclusion

In this paper applicability of the SMT system was examined for part-of-speech disambiguation and lemmatization in Hungarian. Based on my observations these tasks can be considered as translations from plain text to analyzed one. The accuracy of such systems can achieve results of up to 96% accuracy. Although the quality of the above presented systems is behind the state of the art systems – still compareble to those available for Hungarian –, but in my work an absolutely automated system was created which finds the rules itself and we do not have to determine any features for training either. On the other hand this system is able to perform annotation and lemmatization simultaneously.

Some other observations are that we can achieve only minimal increase in the accuracy of the system with minimizing the target language dictionary, but this improvement is not proportional to the information loss. Further significant improvement was achieved by handling out-of-vocabulary words using a method based on word frequencies.

Results showed that only statistical methods are not enough to solve the task of POS-tagging; some kind of hybridization is necessary to improve the quality of the system. The achieved results were encouraging and they pointed out that this way of research contains further possibilities.

Acknowledgement This work was partially supported by TÁMOP – 4.2.1.B – 11/2/KMR-2011-0002 and the members of the Natural Language Processing Group of Pázmány Péter Catholic University.

References

1 Apache. Opennlp. `http://incubator.apache.org/opennlp/`, 2011.

2 Thorsten Brants. Tnt - a Statistical Part-of-Speech Tagger. In *Proceedings of the Sixth Applied Natural Language Processing (ANLP-2000)*, Seattle, WA, 2000.

3 D. Csendes, Cs. Hatvani, Z. Alexin, J. Csirik, T. Gyimóthy, G. Prószéky, and T. Váradi. Kézzel annotált magyar nyelvi korpusz: a Szeged Korpusz. In *I. Magyar Számítógépes Nyelvészeti Konferencia*, pages 238–247. Szegedi Egyetem, 2003.

4 Richárd Farkas, Dániel Szeredi, Dániel Varga, and Veronika Vincze. MSD-KR harmonizáció a Szeged Treebank 2.5-ben. In *VII. Magyar Számítógépes Nyelvészeti Konferencia*, pages 349–353, Szeged, 12 2010. Szegedi Egyetem.

5 Guillem Gascó I Mora and Joan Andreu Sánchez Peiró. Part-of-Speech tagging based on machine translation techniques. In *Proceedings of the 3rd Iberian conference on Pattern Recognition and Image Analysis, Part I*, IbPRIA '07, pages 257–264, Berlin, Heidelberg, 2007. Springer-Verlag.

6 Péter Halácsy, András Kornai, and Csaba Oravecz. HunPos: An open source trigram tagger. In *Proceedings of the 45th Annual Meeting of the ACL*, pages 209–212, Stroudsburg, 2007. Association for Computational Linguistics.

7 Péter Halácsy, András Kornai, Csaba Oravecz, Viktor Trón, and Dániel Varga. Using a morphological analyzer in high precision POS tagging of Hungarian. In *Proceedings of LREC 2006*, pages 2245–2248, 2006.

8 Dan 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. ed., [pearson international edition] edition, 2009.

9 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, 1 edition, 2000.

10 P. Koehn. Moses system. `http://www.statmt.org/moses/`.

11 P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst. Moses: Open Source Toolkit for Statistical Machine Translation. In *Proceedings of the ACL 2007 Demo and Poster Sessions*, pages 177–180, Prague, 2007. Association for Computational Linguistics.

12 Philipp Koehn. *Statistical Machine Translation*. Cambridge University Press, 2010.

13 A. Kornai, P. Rebrus, P. Vajda, P. Halácsy, A. Rung, and V. Trón. Általános célú morfológiai elemző kimeneti formalizmusa. In *II. Magyar Számítógépes Nyelvészeti Konferencia*, pages 172–176. Szegedi Egyetem, 2004.

14 László János Laki and Gábor Prószéky. Statisztikai és hibrid módszerek párhuzamos korpuszok feldolgozására. In *VII. Magyar Számítógépes Nyelvészeti K onferencia*, pages 69–79, Szeged, 12 2010. Szegedi Egyetem.

15 Zhifei Li, Chris Callison-Burch, Chris Dyer, Juri Ganitkevitch, Sanjeev Khudanpur, Lane Schwartz, Wren N. G. Thornton, Jonathan Weese, and Omar F. Zaidan. Joshua: an open source toolkit for parsing-based machine translation. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, StatMT '09, pages 135–139, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.

16 György Orosz and Attila Novák. Purepos – an open source morphological disambiguator. In *Proceedings of the 9th International Workshop on Natural Language Processing and Cognitive Science.*, Wroclaw, Poland, 2012.

17 Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pages 311–318, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

18 Gábor Prószéky and Balázs Kis. A unification-based approach to morpho-syntactic parsing of agglutinative and other (highly) inflectional languages. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, ACL '99, pages 261–268, Stroudsburg, PA, USA, 1999. Association for Computational Linguistics.

19 Adwait Ratnaparkhi. A Maximum Entropy Model for Part-of-Speech Tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, April 16 1996.

20 Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, NAACL '03, pages 173–180, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

21 G. Zipf. Human behaviour and the principle of least-effort. Addison-Wesley, Cambridge, MA, 1949.

22 János Zsibrita, Veronika Vincze, and Richárd Farkas. Ismeretlen kifejezések és a szófaji egyértelműsítés. In *VII. Magyar Számítógépes Nyelvészeti Konferencia*, pages 275–283, Szeged, 12 2010. Szegedi Tudományegyetem.