7th Symposium on Languages, Applications and Technologies

SLATE 2018, June 21–22, 2018, Guimarães, Portugal

Edited by

Pedro Rangel Henriques
José Paulo Leal
António Leitão
Xavier Gómez Guinovart

 OASICS

Editors

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

José Paulo Leal
Faculdade de Ciências
Universidade do Porto
`zp@dcc.fc.up.pt`

António Leitão
Instituto Superior Técnico
Universidade Técnica de Lisboa
`antonio.menezes.leitao@ist.utl.pt`

Xavier Gómez Guinovart
Galician Language Technology and Applications
Universidade de Vigo
`xgg@uvigo.gal`

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.

Contents

Preface

This book compiles the 19 papers —10 full and 9 short— for the 7th edition of the Symposium on Languages, Applications and Technologies (SLATE'2018), held at Minho University, Guimarães, Portugal, from 21st to 22nd of June.

The Symposium receives submissions covering theoretical and practical (technologies and applications) topics on the large area of computer-based automatic language processing. They focus the different problems that arise when dealing with programming languages, annotation or serializing languages and natural languages, presenting the approaches, methods and techniques that shall be used to cope with them. While the approaches are (usually) different for each subarea, they clearly have clear similarities. So, by tradition, this symposium is organized in three tracks chaired by different researchers and reviewed by distinct program committees.

These tracks are:

HHL Track: *Processing Human–Human Languages* is dedicated to the discussion of research projects and ideas involving natural language processing and their industrial application. In 2018 we have 8 papers in this subarea, being dominant the topics like automatic translation, corpora processing; or sentiment analysis;

HCL Track: In *Processing Human–Computer Languages*, researchers, developers, and educators exchange ideas and information on the latest academic or industrial work on language design, processing, assessment, and applications. In 2018, we have 6 papers under the HCL title on language (and domain specific language) design, grammars, and parsing.

CCL Track: The main goal of *Processing Computer–Computer Languages* is to provide a broad platform for discussion on the XML markup language: examples of usage and associated technologies. In 2018 this track has 5 papers focusing learning environments and automatic program assessment.

I am sure we succeed to gather in this book a selection of valuable articles that will provide an enjoyable reading and that will contribute for the progress of the research on language processing.

As General Chair of SLATE 2018, I want to thank the many people without whom this event would never have been possible. The three track Chairs, *António Menezes Leitão*, *José Paulo Leal*, *Xavier Gómez Guinovart*, Publication Chair, *Alberto Simões* and Advertising Chair, *Maria João Varanda*.

I extend this acknowledgment to all the Members of the Scientific Program Committee for their valuable effort reviewing the submissions and deciding the final list of accepted paper; all the Members of the Organizing Committee for looking carefully after all the details concerned with the tremendous logistics necessary to put up the event; to the invited Speakers, *Kent Pitman*, *Luísa Coheur*, *Nuno Carvalho*, and *José Pereira* that let us learn with their research and experiences; to the Authors that communicate their fully implemented ideas or projects, or their fresh proposals that are intended to be realize in the near future; and finally, to the Participants that made actually the conference happen and be a fruitful forum for the exchange of experiences and know-how.

At last but not least, my acknowledgements go to the Institutional Partners and Sponsors, namely the *University of Minho Engineering School* (EEUM), and in particular the *Computer Science Department* (DIUM); the *UNU-EGOV Operating Unit*, for serving as the venue of the symposium; and *Checkmarx Portugal* for sponsoring the keynotes speakers.

We are also indebted to OASIcs —the OpenAccess Series in Informatics for the online publication of this peer-reviewed proceedings— and to MAKE —Machine Learning and Knowledge Extraction, from MDPI—, COMLAN —Computer Languages, Systems and Structures from Elsevier—, and COMSIS —Computer Science and Information Systems journals— for accepting submissions of additionally revised and extended journal-oriented versions of the best papers presented at the symposium; and to the EasyChair conference management system authors and maintainers, whose system was crucial to manage all the Program Committee work.

Pedro Rangel Henriques
Conference Main Chair

This book compiles the 19 papers —10 full and 9 short— for the 7th edition of the Symposium on Languages, Applications and Technologies (SLATE'2018), held at Minho University, Guimarães, Portugal, from 21st to 22nd of June.

The Symposium receives submissions covering theoretical and practical (technologies and applications) topics on the large area of computer-based automatic language processing. They focus the different problems that arise when dealing with programming languages, annotation or serializing languages and natural languages, presenting the approaches, methods and techniques that shall be used to cope with them. While the approaches are (usually) different for each subarea, they clearly have clear similarities. So, by tradition, this symposium is organized in three tracks chaired by different researchers and reviewed by distinct program committees.

These tracks are:

HHL Track: *Processing Human–Human Languages* is dedicated to the discussion of research projects and ideas involving natural language processing and their industrial application. In 2018 we have 8 papers in this subarea, being dominant the topics like automatic translation, corpora processing; or sentiment analysis;

HCL Track: In *Processing Human–Computer Languages*, researchers, developers, and educators exchange ideas and information on the latest academic or industrial work on language design, processing, assessment, and applications. In 2018, we have 6 papers under the HCL title on language (and domain specific language) design, grammars, and parsing.

CCL Track: The main goal of *Processing Computer–Computer Languages* is to provide a broad platform for discussion on the XML markup language: examples of usage and associated technologies. In 2018 this track has 5 papers focusing learning environments and automatic program assessment.

I am sure we succeed to gather in this book a selection of valuable articles that will provide an enjoyable reading and that will contribute for the progress of the research on language processing.

As General Chair of SLATE 2018, I want to thank the many people without whom this event would never have been possible. The three track Chairs, *António Menezes Leitão*, *José Paulo Leal*, *Xavier Gómez Guinovart*, Publication Chair, *Alberto Simões* and Advertising Chair, *Maria João Varanda*.

I extend this acknowledgment to all the Members of the Scientific Program Committee for their valuable effort reviewing the submissions and deciding the final list of accepted paper; all the Members of the Organizing Committee for looking carefully after all the details concerned with the tremendous logistics necessary to put up the event; to the invited Speakers, *Kent Pitman*, *Luísa Coheur*, *Nuno Carvalho*, and *José Pereira* that let us learn

with their research and experiences; to the Authors that communicate their fully implemented ideas or projects, or their fresh proposals that are intended to be realize in the near future; and finally, to the Participants that made actually the conference happen and be a fruitful forum for the exchange of experiences and know-how.

At last but not least, my acknowledgements go to the Institutional Partners and Sponsors, namely the *University of Minho Engineering School* (EEUM), and in particular the *Computer Science Department* (DIUM); the *UNU-EGOV Operating Unit*, for serving as the venue of the symposium; and *Checkmarx Portugal* for sponsoring the keynotes speakers.

We are also indebted to OASIcs —the OpenAccess Series in Informatics for the online publication of this peer-reviewed proceedings— and to MAKE —Machine Learning and Knowledge Extraction, from MDPI—, COMLAN —Computer Languages, Systems and Structures from Elsevier—, and COMSIS —Computer Science and Information Systems journals— for accepting submissions of additionally revised and extended journal-oriented versions of the best papers presented at the symposium; and to the EasyChair conference management system authors and maintainers, whose system was crucial to manage all the Program Committee work.

Pedro Rangel Henriques
Conference Main Chair

List of Authors

Afonso Xavier Canosa
University of Santiago de Compostela
Galiza, Spain
`canosarodrigues@gmail.com`

Alberto Simões
2Ai Lab, Escola Superior de Tecnologia
Instituto Politécnico do Cávado e do Ave
Barcelos, Portugal
`asimoes@ipca.pt`

Ana Alves
CISUC / ISEC
Polytechnic Institute of Coimbra
Coimbra, Portugal
`ana@dei.uc.pt`

André Silva
Faculty of Sciences
University of Porto, Portugal
`up201007410@fc.up.pt`

António Menezes Leitão
Instituto Superior Técnico
INESC-ID, Lisbon, Portugal
`antonio.menezes.leitao@tecnico.pt`

Bruno Ferreira
Instituto Superior Técnico
INESC-ID, Lisbon, Portugal
`bruno.b.ferreira@tecnico.ulisboa.pt`

Helder Correia
CRACS & INESC-Porto LA
Faculty of Sciences
University of Porto, Portugal
`up201108850@fc.up.pt`

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

Jan Janoušek
Department of Theoretical Computer Science
Czech Technical University in Prague
aculty of Information Technology
`Jan.Janousek@fit.cvut.cz`

Jan Trávníček
Department of Theoretical Computer Science
Czech Technical University in Prague
Faculty of Information Technology
`Jan.Travnicek@fit.cvut.cz`

José Carlos Paiva
CRACS & INESC-Porto LA
Faculty of Sciences
University of Porto, Portugal
`up201200272@fc.up.pt`

José João Almeida
Centro Algoritmi / Dpt. de Informática
Universidade do Minho, Campus de Gualtar
Braga, Portugal
`jj@di.uminho.pt`

José-Luis Sierra
Fac. Informática
Universidad Complutense de Madrid
28040 Madrid, Spain
`jlsierra@ucm.es`

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

Pablo Gamallo
Centro Inv. en Tecnoloxías da Información
University of Santiago de Compostela
Galiza, Spain
`pablo.gamallo@usc.es`

Paulo Gomes
CISUC, University of Coimbra
Coimbra, Portugal
`pgomes@dei.uc.pt`

Paulo Novais
Centro Algoritmi / Dpt. de Informática
Universidade do Minho
Braga, Portugal
`pjon@di.uminho.pt`

Pedro Rangel Henriques
Centro Algoritmi / Dpt. de Informática
Universidade do Minho
Braga, Portugal
prh@di.uminho.pt

Radomír Polách
Department of Theoretical Computer Science
Czech Technical University in Prague
Faculty of Information Technology
Radomir.Polach@fit.cvut.cz

Ricardo Giuliani Martini
Centro Algoritmi / Dpt. de Informática
Universidade do Minho
Braga, Portugal
rgm@algoritmi.uminho.pt

Ricardo Martins
Centro Algoritmi / Dpt. de Informática
Universidade do Minho
Braga, Portugal
ricardo.martins@algoritmi.uminho.pt

Ricardo Queirós
CRACS & INESC-Porto LA
DI/ESMAD/Politécnico do Porto
Porto, Portugal
ricardoqueiros@esmad.ipp.pt

Ricardo Ribeiro
INESC-ID Lisboa & ISCTE
Instituto Universitário de Lisboa
Lisbon, Portugal
ricardo.ribeiro@iscte-iul.pt

Ricardo Rodrigues
CISUC / ESEC
Polytechnic Institute of Coimbra
Coimbra, Portugal
rmanuel@dei.uc.pt

Rubén Pereira
ISCTE / Instituto Universitário de Lisboa
Lisbon, Portugal
Ruben.Filipe.Pereira@iscte-iul.pt

Rui Encarnação
CISUC, University of Coimbra
Coimbra, Portugal
race@dei.uc.pt

Rui Mendes
Centro Algoritmi / Dpt. de Informática
Universidade do Minho, Campus de Gualtar
Braga, Portugal
azuki@di.uminho.pt

Sara Silva
ISCTE, Instituto Universitário de Lisboa
Lisbon, Portugal
satsa@iscte-iul.pt

Tomáš Pecka
Department of Theoretical Computer Science
Czech Technical University in Prague
Faculty of Information Technology
Tomas.Pecka@fit.cvut.cz

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

Committees

Conference Chairs

Main Program Chair:

Pedro Rangel Henriques
Universidade do Minho, PT

Track Chairs:

Human-Human Languages:
Xavier Gómez Guinovart
Universidade de Vigo, ES

Human-Computer Languages:
António Leitão
Instituto Superior Técnico, PT

Computer-Computer Languages:
José Paulo Leal
Universidade do Porto, PT

Publication Chair:

Alberto Simões
Instituto Politécnico do Cávado e do Ave, PT

Organization Committee

Pedro Rangel Henriques
Universidade do Minho, PT

Alberto Simões
Instituto Politécnico do Cávado e do Ave, PT

Maria João Varanda Pereira
Instituto Politécnico de Bragança, PT

José Carlos Ramalho
Universidade do Minho, PT

José João Dias de Almeida
Universidade do Minho, PT

Sara Santos Fernandes
United Nations University, PT

Goreti Pereira
Universidade do Minho, PT

Scientific Committee

Alberto Simões
Instituto Politécnico do Cávado e do Ave, PT

Alda Gancarski
Telecom SudParis, FR

Alexander Paar
TWT GmbH Science and Innovation, DE,

Alexandre Rademaker
IBM Research Brazil, BR

Antoni Oliver
Universitat Oberta de Catalunya, ES

Antonio Leitão
Instituto Superior Técnico, PT

António Teixeira
University of Aveiro, PT

Arantza Diaz De Ilarraza
University of the Basque Country, ES

Arkaitz Zubiaga
The University of Warwick, UK

Brett Drury
SciCrop, BR

Cristina Ribeiro
University of Porto, PT

Daniel Zeman
Univerzita Karlova, CZ

Daniela da Cruz
Checkmarx, PT

Dietmar Seipel
University of Wuerzburg, DE

Fernando Batista
INESC-ID & ISCTE-IUL, PT

Filipe Portela
University of Minho, PT

Francis M. Tyers
Higher School of Economics, RU

Gabriel David
Universidade do Porto, PT

Giovani Librelotto
Universidade Federal de Santa Maria, BR

Horacio Saggion
Universitat Pompeu Fabra, ES

Kaang: A RESTful API Generator for the Modern Web

Ricardo Queirós

CRACS & INESC-Porto LA & DI/ESMAD/P.PORTO, Porto, Portugal
ricardoqueiros@esmad.ipp.pt
 https://orcid.org/0000-0002-1985-6285

Abstract

Technology is constantly evolving, as a result, users have become more demanding and the applications more complex. In the realm of Web development, JavaScript is growing in a surprising way, already leaving the boundaries of the browser, mainly due to the advent of Node.js. In fact, JavaScript is constantly being reinvented and, from the ES2015 version, began to include the OO concepts typically found in other programming languages.

With Web access being mostly made by mobile devices, developers face now performance challenges and need to perform a plethora of tasks that weren't necessary a decade ago, such as managing dependencies, bundling files, minifying code, optimizing images and others. Many of these tasks can be achieved by using the right tools for the job. However, developers not only have to know those tools, but they also must know how to access and operate them. This process can be tedious, confusing, time-consuming and error-prone.

In this paper, we present Kaang, an automatic generator of RESTFul Web applications. The ultimate goal of Kaang is to minimize the impact of creating a RESTFul service by automating all its workflow (e.g., files structuring, boilerplate code generation, dependencies management, and task building). This kind of generators will benefit two types of users: will help novice developers to decrease their learning curve while facing the new frameworks and libraries commonly found in the modern Web and speed up the work of expert developers avoiding all the repetitive and bureaucratic work. At the same time, Kaang promotes the good development principles by adding automatic testing and documentation generation.

For this accomplishment, Kaang generates the main API content based on the user's input and a set of templates which will help developers to manage and test routes, define resources, store data models and others. In order to provide an addition level of confidence to the generator's end-users, the generator will be integrated on Travis CI and published on both the npmjs and Yeoman registries.

1   Introduction

Nowadays we are witnessing a remarkable revolution in the Web frontend development. A decade ago we just needed to master the 3 pillars of the Web: the HyperText Markup Language (HTML) for structuring content, the Cascading Style Sheets (CSS) to style it and the JavaScript language to attach some behavior. With the evolution of the Web, users became more demanding and the applications became more complex. At the same time, browsers became more powerful and, consequently, the growing access to the Web through mobile devices increased. This made the frontend work more bureaucratic from the ad hoc management of few HTML/CSS/JS files to the automation of complex workflows. These new workflows are characterized by several tools and libraries organized into several categories which can be boil down to three main categories: scaffolding tools, dependency managers and build tools [3].

While all these tools help developers in their day-to-day lives, they also create a big learning curve for those that are starting in the Web development realm. Even for experts, this kind of tools and repetitive tasks become time-consuming, boring and their mechanical nature can produce undesirable errors either working alone or within a team. A typical scenario is the creation of RESTful APIs. In this context, we need to gather several libraries and frameworks to handle route management, resource definition, data persistence and unit tests. Also, we need to create the client-side, typically composed by a responsive and friendly GUI based on forms to easily communicate with the API endpoints and feed the respective models. As you can imagine, this process is time-consuming and error prone.

This paper presents **Kaang** as a RESTful API generator. In order to use the generator, the developer only has to open the command line and invoke the generator. During the application generation process, the API is customized based on the developers' input data. At the end, a tested and documented RESTful API is generated and can be consumed by a responsive GUI or using your favorite API tester (e.g., Postman). Obviously, the programmer may have to do some refinement, namely adding routes or new features. But the important thing is that the entire software development workflow had been set up and all the initial hard work mitigated, freeing the developer for more important coding aspects of the application.

The remainder of this paper is organized as follows: Section 2 reviews the current tools that nowadays gravitate in the Web development workflow and enumerates and compares the main existing generators based on three criteria: maturity, coverage and performance. In Section 3, we present Kaang and describe its architecture and main components, the input system, the routes management and the generation of tests and documentation. In Section 4, we validate Kaang enumerating the steps for the generation of a Movies API. Finally, we conclude with a summary of our main contributions and a perspective of future research.

2   Web development workflow

The Web development is going through an unprecedented phase of profound changes. Nowadays, a frontend Web developer should not only master the HTML/CSS/JavaScript triad, but also have a (basic) knowledge of what are preprocessors, module bundlers, task runners, scaffolding tools and other automation tools. In fact, these tools are mandatory in the so-called "new web development workflows" mitigating developers work and saving their time for others tasks.

Based on a typical software development workflow, you can easily translate it for the Web realm identifying three phases developers go through when coding (Figure 1).

The **setup phase** is the starting point where developers commonly set up project structure, apply reusable patterns/boilerplate code and install third-party libraries.
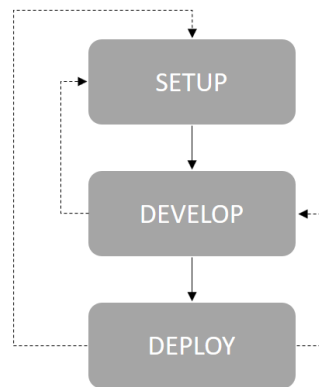
Figure 1 Phases of the Web development workflow.

The **development phase** is where developers write code using a programming language. If necessary, developers can go back to the setup phase, if they need to add a new module or refactor some code. Also here is the right place to perform quality control based on tests.

The **deployment phase** is where developers create an executable bundle from the code written in the previous phase and deploy it. In the Web context, this boils down to deploy the HTML/JavaScript/CSS to an web server. Obviously, developers can return to the development phase (fixing bugs or adding new features to existing code) or to the setup phase (creating new modules).

In order to meet all those expectations proper tooling is needed. In the next two sections Web tooling is depicted. In Section 2.1 the Web tooling is organized into three categories: scaffolding tools, dependency managers and build tools. In Section 2.2, we take a closer look for scaffolding tools and we compare those that generate RESTfull Web apps based on predefined criteria.

2.1 Frontend tooling

Nowadays, a typical Web development workflow [1] comprises three types of tooling which can be organized as follows:

- **Scaffolding tools**: generate the Web project structure, inject boilerplate code and add new modules;
- **Dependency managers**: take care of the self-contained modules, or packages and potential conflicts between them (e.g., dependencies versions) avoiding redundancy;
- **Build tools**: perform all the file-processing tasks, transforming your source code into something deployable. Typical examples are bundle and minify JavaScript and CSS files (e.g., MinifyCSS, UglifyJS), remove dead code (e.g., PurifyCSS), lint code (e.g., ESlint), optimize images (e.g., ImageMagick, ImageOptim), preprocess source code (e.g., CoffeeScript, LiveScript, Less, Sass, PostCSS), run tests (e.g., Jasmine, Mocha, Jest) and many others [2]. The configuration of those tools is stored in build files.

As depicted in Figure 2, the modern Web developer, in a typical scenario, developers start a scaffolding tool (e.g., Brunch, Yeoman) to set up the project, search and install components with a dependency manager (e.g., Bower, WebPack), and process files periodically through a build tool (e.g., Grunt, Gulp) [4]. The first two types of tools work mainly in the initialization phase, while the last one gravitates in the development and deployment phases.
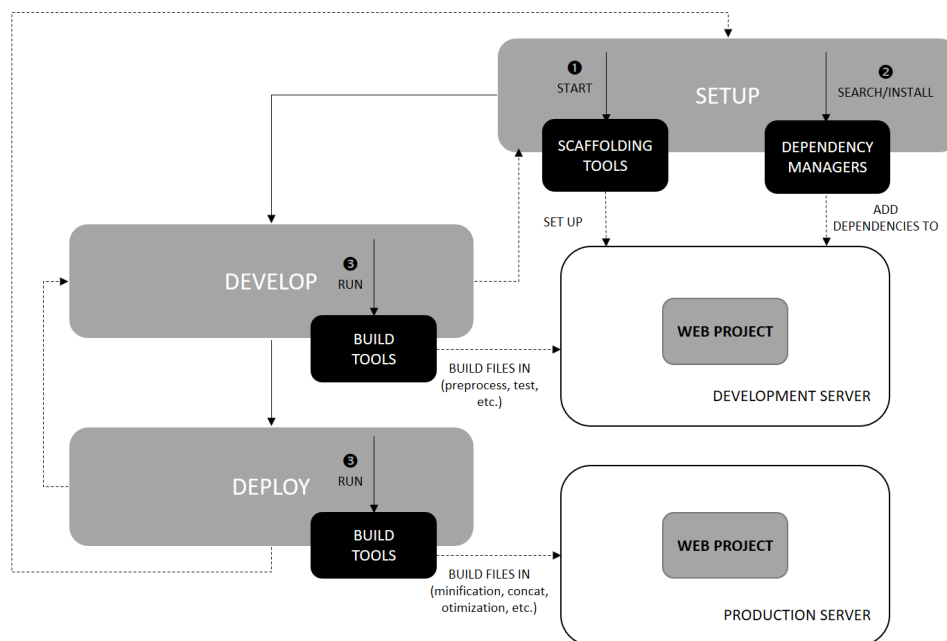
■ **Figure 2** Web tooling.

Thus, the idea is straightforward: instead of using a huge number of tools individually for all the development tasks, developers should invest on automating the execution of tasks through these three different tool types. With these three easy-to-use interfaces all the complexity is hidden and developers can thereafter concentrate their efforts in the business logic.

2.2   REST API generators

One of the most important type of tools in the modern Web workflow is the scaffolding tools. As said before, these type of tools help developers to quickly build web applications by creating the necessary folders, copying initial files (like build scripts), applying boilerplate code, and triggering the installation of dependencies. Throughout development, these tools are also responsible for the creation of the base structure of new modules inside the Web project.

In this context, one of the most dominating tool is **Yeoman**, powered by Google. Despite, the earlier success of Loom and Brunch tools, Yeoman is currently the best way to kick-start new projects, prescribing best practices and tools. The success of yeoman is mainly due to its generator ecosystem. A generator can be defined as a plugin that can be run with the 'yo' command to scaffold complete projects. Currently, Yeoman supports more than 7000 plugins from basic web apps to complex generators for the popular frameworks of Angular, React, Polymer and others.

A generator comes with a folder full of templates that have to be transferred and updated by the generator script based on a simple prompting API that allows different parameters to change the output generated by the generator. Yeoman's only task is to run the generator.

■ **Table 1** Generators maturity.

GitHub data	generator-rest	generator-sails-rest-api	generator-api
First release date	Sep/2016	Jan/2015	Sep/2016
Last release date	0.12.0 (Mar/2018)	1.3.13 (Dec/2017)	1.5.3 (Mar/2018)
# Open issues	11	12	3
# Pull requests	1	7	2
# Commits	130	1486	151
# Releases	22	63	9
# Contributors	13	10	11
# Stars	602	325	188
# Forks	117	63	33

In this paper we take a closer look for RESTful API generators. The selection of the generators was very simple. We access the yeoman generators repository[1] and search the generators using the keyword "rest". Then we sorted the results by popularity (stars of GitHub) and selected the top three, namely: generator-rest[2], generator-sails-rest-api[3] and generator-api[4].

In the next subsections the three generators are described and compared based on three facets: maturity, coverage and performance.

2.2.1 Maturity

It is difficult to determine which generator is most widely used or have more impact. Various methods of measuring tools popularity have been proposed, each subject to a different bias over what is measured. In this context, we will focus on comparing the activity in GitHub where all the selected generators are stored. In fact, the three generators chosen are pretty active on GitHub, as you can see in Table 1.

Although relatively recent, the generator-rest is the most popular, with a larger number of stars and forks. The number of forks is relevant. A fork is a copy of a repository. Forking a repository allows you to freely experiment a repo (with changes) without affecting the original project. Thus, this means that most people are using the Yeoman generator-rest base code to start their own projects. Regarding the generator-api, it presents the lower values of the three in almost all the indicators.

2.2.2 Coverage

For the coverage criterion we will make a comparison of the three Yeoman generators regarding the support for tasks typically found in build tools and REST specific features.

In Table 2 we present the comparison on generators build tools features.

As you can conclude, we have disappointing results, but there is an obvious reason. Web RESTful applications depend mostly in thin Web clients while most of the hard work is

---

[1] http://yeoman.io/generators
[2] https://github.com/diegohaz/rest
[3] https://github.com/ghaiklor/generator-sails-rest-api
[4] https://github.com/ndelvalle/generator-api

Table 2 Build tools features comparison.

Features	generator-rest	generator-sails-rest-api	generator-api
Bundler	-	-	-
Linter	ESLint	-	ESLint
Minifier	-	-	-
Optimizer	-	-	-
PreProcessor	-	Sass/Coffee	-
Reloader	-	-	-
Tester	jest	mocha	mocha

Table 3 REST features comparison.

Features	generator-rest	generator-sails-rest-api	generator-api
CRUD	yes	yes	yes
User login	yes	yes	no
Pagination	yes	no	no
Tester	yes	yes	yes
Documenter	yes	no	no
New models	yes	no	yes
Predefined services	no	yes	no
MVC	no	yes	no

server-side. Thus, you do not need to handle large and complex HTML/CSS files on the client. For that reason, is not crucial the presence of most of the tools included in this table.

In Table 3 we present the comparison on generators REST features.

One can conclude that all the generators have the basic ability to support CRUD functions in one (or more) endpoints. Also all the generators have an authentication layer based on JWT, Facebook, Twitter, GitHub, Instagram, Google Plus and other social networks. The pagination feature is only supported by the generator-rest through querymen, a Querystring parser middleware for MongoDB, Express and Nodejs (MEN). All the generators support tests. The generator-rest uses jest[5] and the others two use mocha[6]. Regarding documentation, only the generator-rest supports the creation of documentation from API annotations in the source code. The generation of documentation is based on apiDoc[7] – an inline documenter for RESTful web APIs. Also, all the generators foster the creation of new models and endpoints. However, the generator-api provides a quasi-automatic approach based on the concept of subgenerators. Once you have the generated project, if you want to add a new model you can simply run yo api:model. This will generate a new folder under model, and, then, you just need to import the route. Finally, the generator-sails-rest-api is the only generator that comprises several predefined services (cipher, image, payment, sms, etc.) and implements MVC through Sails.js[8] – a realtime MVC framework for Node.js.

---

[5] `https://facebook.github.io/jest/`
[6] `https://mochajs.org/`
[7] `http://apidocjs.com/`
[8] `https://sailsjs.com/`

Table 4 Performance benchmark.

Generators	generator-rest	generator-sails-rest-api	generator-api
Installation (npm)	10.73s	10.85s	10.91s
Generation (yo)	25.33s	4.31s	8.42s

2.2.3   Performance

In this subsection, the three generators are compared in terms of performance. This performance benchmark will be achieved by measuring the installation time of the generators through npm and the generation time of new projects through yeoman.

Obviously, these times are not very important in the production phase, but they are crucial in the development phase, since they allow to measure the impact in terms of time during the development process of the generator.

For the experiment, we started by installing Node.js v8.11.1 (includes npm 5.8.0) in a machine running Windows 10 (64 bits), Intel Core i7-6700K at 4.00GHz, 16 GB RAM and SSD. The results are presented in Table 4.

The first line of the table reflects the time spent by npm to install the generator. The npm tool is a package manager for JavaScript and is the world's largest software registry.

The npm tool is distributed with Node.js – which means that when you download Node.js, you automatically get npm installed on your computer. The command used for install the generators was the following:

```
npm install %%GENERATOR_NAME%% -g
```

The second line reflects the time spent by Yeoman to generate a new Web project based on the respective generator. For these you should firstly install yeoman, then create a folder for the new project and, finally, you can use yo to generate your project.

```
npm install -g yo
mkdir %%PROJECT_FOLDER%%
cd %%PROJECT_FOLDER%%
yo %%GENERATOR_NAME%%
```

Based on these results, the main conclusion is that all the generators take almost the same time to be installed through npm. However, when we look for the projects generation time, we see that generator-rest takes too much time when compared with the remainder generators. The reason for this discrepancy is due to the high number of libraries and tools provided by this generator.

3   Kaang

In this section we present Kaang, a RESTful API generator. The ultimate goal of Kaang is to help developers to quickly create basic REST API applications using the modern Web tooling. Kaang is based on Yeoman.

Yeoman is an open source client-side development stack, consisting of tools and frameworks intended to help developers build web applications. The most important part of Yeoman is the concept of generators. Yeoman generators are, at their core, Node.js modules and can be defined as building blocks on the Yeoman ecosystem. For the creation of Kaang, we didn't create it from scratch, instead we use a Yeoman generator (called generator-generator) to
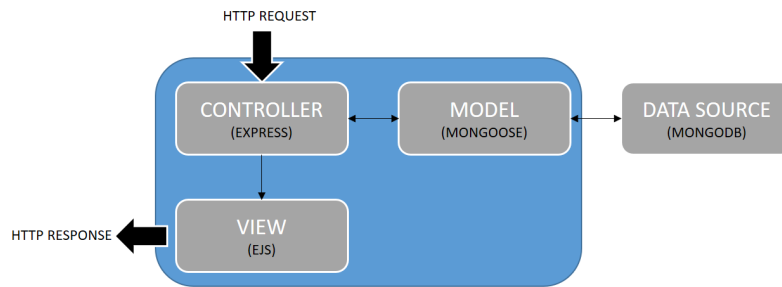
■ **Figure 3** Kaang architecture.

automatically create a generator skeleton. Based on this skeleton we applied several changes which are depicted in the next subsections.

3.1 Architecture

Kaang's generated application architecture is based on a typical server-side API, running in a Node.js server and implementing MVC as depicted in Figure 3.

Kaang is composed by the following four main components:

1. Web framework
2. Database
3. Object document Mapper
4. Template engine

Web framework.   For the Web framework we selected Express.js[9] as the REST framework which includes basic routing to determine how the generated application will respond to a client request to a particular endpoint. Each route can have one or more handler functions, which are executed when the route is matched.

Database.   The database chosen was MongoDB[10]. Currently, MongoDB is the most popular NoSQL database since it integrates seamlessly in the Node.js realm.

Object document mapper.   In order to map the model with the database, we selected a mediator responsible to define objects with a strongly-typed schema which is mapped to a MongoDB document. In this case, the selection was easy due to the popularity of Mongoose[11], an Object Document Mapper (ODM) for MongoDB documents.

Template engine.   A template engine enables the use of static template files in a Web application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page. Express uses Pug (by default), but we can use others such as Mustache or EJS. For the Kaang generator, we opted for EJS[12] as the official template engine due to its simplicity and increasing popularity.

---

[9] https://expressjs.com/
[10] https://www.mongodb.com/
[11] http://mongoosejs.com/
[12] http://ejs.co/

3.2 Structure and Input

By default, Kaang presents a basic structure influenced by the structure generated by the express-generator tool, responsible to quickly create an application skeleton in Express. The following list shows the main folders and files generated by Kaang:

```
-- config/
    -- bower/
    -- gulp/
-- public/
    -- images/
    -- js/
    -- css/
    -- libs/
        -- bootstrap
        -- jquery
-- routes/
    -- all.js
-- models/
    -- movies.js
-- views/
    -- error.ejs
    -- index.ejs
    -- layout.ejs
-- tests/
    -- api.test.js
-- node_modules/
-- server.js
-- package.json
```

Once you have this structure in place, it's time to write the actual generator. Yeoman offers a base generator which you can extend to implement your own behavior.

The first task is receive input data from the user while generating the Web application. To accomplish this task, we use the Prompting API from Yeoman. The prompt module is provided by Inquirer.js and you should refer to its API for a list of available prompt options. The prompt method is asynchronous and returns a promise. You'll need to return the promise from your task in order to wait for its completion before running the next one. Listing 1 shows an excerpt from the Kaang generation main file.

The generator starts by asking some questions to the user. For our generator, the questions are:

- **Name of the application**: the name of the folder for the generated application. Also this name will be injected in several configuration files (bower, gulp, etc.) and in the title element of the main HTML file;
- **Use of a Web framework (Bootstrap)**: a yes|no question that will give the user the chance to install Bootstrap and use it to add responsiveness to the main HTML file and for more sophisticated GUI components to be included in the Web application;
- **Use of a JavaScript library (jQuery)**: a yes|no question that will give the user the chance to install jQuery and use it in the event management and AJAX calls in the main JavaScript file of the application;

■ **Listing 1** Main generator file.

```
'use strict';
const Generator = require('yeoman-generator');
const chalk = require('chalk');
const yosay = require('yosay');

module.exports = class extends Generator {
  constructor(args, opts) {
    super(args, opts);
    this.log('Initializing KAANG generator!');
  }
  prompting() {
    this.log(yosay('Welcome ${chalk.red('generator-kaang')}!'));
    const prompts = [
      {
        type: 'input',
        name: 'name',
        message: 'Your project name?',
        default: this.appname
      },
      ...
    ];
    return this.prompt(prompts).then(props => {
      // To access props later use this.props.name;
      this.props = props;
      ...
    }); }
};
```

- **Use of a test framework (jest)**: a yes|no question that will give the user the chance to use jest to test the API endpoints.
- **Use an API documenter (apiDocs)**: a yes|no question that will give the user the chance to have inline Documentation for the RESTful web API.

Based on the users' answers, Yeoman takes the predefined templates (based on the previous structure) and execute several tasks: injects the user data, made conditional copies, etc. Listing 2 shows the use of the copyTpl method in order to copy a template file with automatic data injection.

The template language used is EJS which aims to help render JavaScript code in the client side. In this case, the value of the name variable is injected in the `_index.ejs` template as shown in Listing 3.

Beyond data injection, the user's data can influence the inclusion of references in the dependency management file (`_bower.json`). That is the case of Bootstrap and jQuery installation which will depend on the users acceptance.

3.3   Routes and Models

Kaang generator creates, by default, an API for movies management. The API exposes the several endpoints. In Table 5, we present some of the available endpoints and a respective description.

■ **Listing 2** Template generation.

```
// Scaffolding
writing() {
    // Copy application files
    this.fs.copyTpl(
      this.templatePath('_views/_index.ejs'),
      this.destinationPath('views/index.ejs'),
      { name: this.props.name }
    );
      ...
  }
```

■ **Listing 3** Basic template file.

```
// _index.ejs
<html>
  <head>
    <title><%= name %></title>
  </head>
  ...
</html>
```

The reference API of the movie resource is composed by three functions. The GET function retrieves all the movies resources. The POST function inserts a new movie passed as a JSON string through a POST parameter. The DELETE function removes a specified movie (id included in the URI of the endpoint). Listing 4 shows the routing of these endpoints to specific function handlers

This file imports the module all.js which contains the concrete implementation of the routes, as presented in Listing 5.

These functions will interact with the Movies model represented as a Mongoose schema. This schema mediates the interaction with the respective MongoDB collection, as shown on Listing 6.

3.4   Testing

Kaang uses Jest, a testing framework written by Facebook, to test the API routes. The best way to organize the API tests is to separate each generator (and sub-generator) into its own describe block. Then, use a test block for each assertion. In code, this should end up with a structure similar to the presented in Listing 7.

Then, you can trigger the tests in the task runner workflow or invoke manually the npm command **npm run test** from the command line. The results are shown in Figure 4.

3.5   Documentation
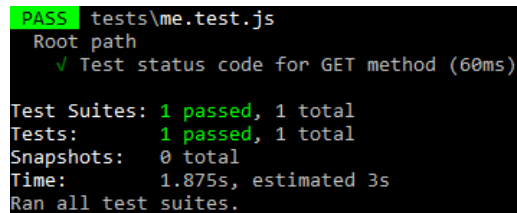
For the documentation, the Kaang generator uses the apiDoc tool that can be defined as an inline documentation generator for RESTful web APIs. This tool creates documentation based on API annotations included in the JavaScript source code. For the use of this tool we had to update the package.json file, namely the devDependencies section to include the

Table 5 Endpoints of the Movies API.

Endpoint	Description
GET /movies	Gets all the movie resources
POST /movie	Creates a new movie resource
DELETE /movie/:movieId	Remove a specified movie resource

Listing 4 Application routing.

```
...
let routes = require('./routes/all');
// Creation of the routes
app.get('/', routes.index);
app.get('/movies', routes.getMovies);
app.post('/movie', routes.postMovie);
app.delete('/movie/:id', routes.removeMovie);
...
```


Figure 4 Results of API test using Jest.

references to apiDoc and opn-cli (a cross-platform node-open which opens websites, files, executables, etc.).

```
"devDependencies": {
    "apidoc": "^0.17.6",
    "opn-cli": "^3.1.0"
}
```

After that we include a new script in the file:

```
"docs": "apidoc -i routes -o docs && opn docs/index.html"
```

Thus, all we need to do to generate the API docs and show the documentation is type the following command.

```
npm run docs
```

This command opens the respective file in the Web browser (Figure 5).

4   Validation

In order to validate Kaang we enumerate the necessary steps to generate and run a Web application. For that purpose, you just need to execute the following steps to see your Web app running:

■ **Listing 5** Routes implementation.

```
//_routes/_all.js
const Movie = require('../models/movie').movies
// Renders the main page (index.ejs)
module.exports.index = function(req, res) {
  res.render('index')
}
// Returns all movies
module.exports.getMovies = function(req, res) {
  Movie.find().exec(function(err, movie) {
    return err ? res.send(err) : res.json(movie)
  })
}
// Adds a new movie
module.exports.postMovie = function(req, res) {
  Movie.create(req.body, function(err, movie) {
    return err ? return res.send(err) : res.json(movie)
  })
}
// Remove an existent movie
module.exports.removeMovie = function(req, res) {
    let movieId = req.params.id
    Movie.find({'_id': movieId}).remove().exec(function(err, movie) {
      return err ? return res.send(err) : res.json(movie)
    })
  }
```

■ **Listing 6** SOS schema for a task.

```
//_model/_movie.js
let mongoose = require('mongoose');
let Schema = mongoose.Schema;
let movieSchema = new Schema({
    title:  String,     year: Number,
    genre: String,      votes: Number
  });
module.exports.movies = mongoose.model('Movie', movieSchema);
```

■ **Listing 7** SOS schema for a task.

```
//_tests/_me.test.js
const app = require('../app')
describe('Root path', () => {
    test('Test status code for GET method', () => {
        return request(app).get("/").then(response => {
            expect(response.statusCode).toBe(200)
        })
    });
})
```
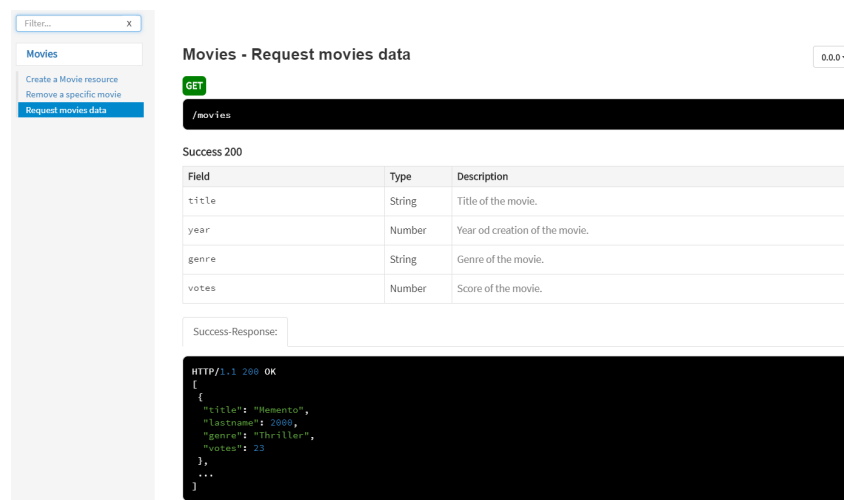
Figure 5 Documentation HTML file.
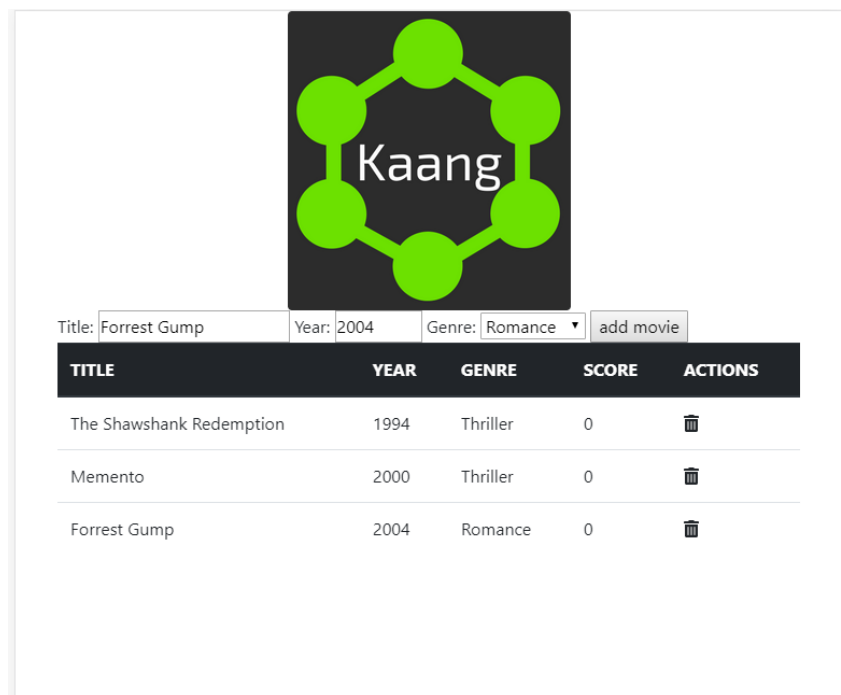
Figure 6 Kaang generator app frontend.

1. Call **mongod** to initiate MongoDB;
2. Generate Kaang's web app typing in the command line: **yo kaang**;
3. Answer the generator questions;
4. Execute **npm start** to start the server;
5. Open a browser and type **http://localhost/8080**.

The GUI of the generated Web application is depicted in Figure 6.

As you can see, it is a simple GUI, for testing purposes. Here, you can browse all the movies, add a new movie or delete an existent movie.

Kaang's source code is available at GitHub[13]. As part of being published to both the npmjs and Yeoman registries, the generator will be integrated on Travis CI. This should provide an addition level of confidence to the generator's end-users. Additionally, Travis CI feeds test results to Coveralls, which displays the generator's code coverage.

5   Conclusions

This paper describes Kaang as a RESTful API generator. The paper comprises several contributions, namely:
- It presents the state of art of REST generators using JavaScript;
- It can be used as a practical guide for those who aim to create their own generators;
- It abstracts the complexity of the creation of REST APIs;
- It fosters the use of generators decreasing the bureaucratic work and repetitive tasks;
- It decreases the learning curve for those who want to enter in the modern Web realm
- It fosters the use of good development practices such as testing and documenting.

As future work, the main idea is to extend this generator to cover other features such as the possibility to automatically generate new models/routes, the support for authentication based on JWT, the support for JavaScript and CSS preprocessors (e.g. CoffeeScript, LiveScript, Less, Sass) and the inclusion of a better module bundler and task runner with predefined runnable tasks (minification, image optimization, etc.). In this context, we expect that the new version of the Kaang generator supports Webpack[14].

──── **References** ────

1   Stefan Baumgartner. *Front-End Tooling with Gulp, Bower, and Yeoman.* Manning Publications, 2017.
2   Marijn Haverbeke. *Eloquent JavaScript.* No Starch Press, 2017.
3   Ricardo Queirós. CSS preprocessing: Tools and automation techniques. *Information*, 9(1), 2018. `doi:10.3390/info9010017`.
4   Alex Rauschmayer. *Speaking JavaScript: An In-Depth Guide for Programmers.* O'Reilly, 2017.

---

[13] `https://github.com/rqueiros/Kaang`
[14] `https://webpack.js.org/`

LearnJS - A JavaScript Learning Playground

Ricardo Queirós

CRACS & INESC-Porto LA & DI/ESMAD/P.PORTO, Porto, Portugal
ricardoqueiros@esmad.ipp.pt
https://orcid.org/0000-0002-1985-6285

Abstract

The JavaScript ecosystem is evolving dramatically. Nowadays, the language is no longer confined to the boundaries of the browser and is now running in both sides of the Web stack. At the same time, JavaScript it's starting to play also an important role in desktop and mobile applications development. These facts are leading companies to massively adopt JavaScript in their Web/mobile projects and schools to augment the language spectrum among their courses curricula.

Several platforms appeared in recent years aiming to foster the learning of the JavaScript language. Those platforms are mainly characterized with sophisticated UI which allow users to learn JavaScript in a playful and interactive way. Despite its apparent success, these environments are not suitable to be integrated in existent educational platforms. Beyond these interoperability issues, most of these platforms are rigid not allowing teachers to contribute with new exercises, organize the existent exercises in more suitable and modular activities to be deployed in their courses, neither keep track of student's progress.

This paper presents LearnJS as a simple and flexible platform to teach and learn JavaScript. In this platform, instructors can contribute with new exercises and combine them with expositive resources (e.g videos) to define specific course activities. These activities can be gamified with the injection of dynamic attributes to reward the most successful attempts. Finally, instructors can deploy activities in their educational platforms. On the other hand, learners can solve exercises and receive immediate feedback on their solutions through static and dynamic analyzers. Since we are in the early stages of implementation, the paper focus on the presentation of the LearnJS architecture, their main components and their data and integration models. Nevertheless, a prototype of the platform is available in a GitHub repository.

1 Introduction

Nowadays, the JavaScript (JS) language is no longer seen as a browser scripting language to validate forms and make AJAX calls to Web servers. In fact, the language has evolved in a consistent way and can already be used to create applications on the most popular

platforms. One of the great impulses for this growth was the appearance of Node.js, which allows developers to use JS throughout the stack (front and backend) of a Web application. But it's not just on the Web domain that JS is having a huge success. In fact, JS can already be used to create native/hybrid mobile (e.g. React Native, Ionic) and desktop applications (e.g. Electron, WinJS, NW.js). Last but not least, several game engines based on JavaScript can be used for making HTML5 games for desktop and mobile web browsers, supporting Canvas and WebGL rendering (e.g Phaser, Cocos2d).

Obviously, the rise of JS and its corresponding omnipresence led companies to start adopting the language since it allows their development teams the need to master a single language for the cross-platform development of their products. This growth has also re-activated the JavaScript community, being nowadays considered one of the most popular languages according to several studies[1]. At the same time, there is a concern from Schools to adjust their courses curricula to teach these skills not only at the language level, but also to adopt the most popular frameworks and tools that are now gravitating on the Web.

In this context, several online platforms have appeared in recent years aiming to foster the learning of JavaScript. These platforms, typically coupled in online learning platforms (e.g Udemy, Udacity), provide sophisticated UI and a very strong level of interaction, facilitating the progress of students through creative examples. Regardless of their popularity, these platforms have issues regarding interoperability with educational systems and flexibility in content management. For instance, teachers can only advise the use of such tools for training purposes and cannot use them to define specific learning activities and keep track on the evolution of students.

This paper presents LearnJS as a learning environment for the teaching-learning process of the JavaScript programming language. The platform allows two main use cases: teachers can contribute with new resources, combine existing resources into activities and distribute activities in learning management systems; students can access activities, solve exercises and receive automatic feedback. Both use cases have important points that should be emphasized. In the case of teachers, the activities created can include expository resources (e.g. PDF, videos) and evaluative resources (e.g. exercises). Also, gamification attributes (e.g. levels, hints, achievements, leaderboards, unlock levels and code skeletons) can be assigned to provide playful and engaged activities to students. In the case of the students the feedback returned by the platform is not only produced by dynamic evaluation (tests cases), but also by static code analysis through the use of linters which are responsible for the inspection of potential buggy code.

The remainder of this paper is organized as follows: Section 2 reviews the existing environments to learn JavaScript and focuses its attention on Web platforms. In this context several platforms are compared according to several criteria: interoperability, flexibility. In Section 3, the LearnJS architecture and its main components are presented. In this context, we expose the data and interoperability models. In Finally, we conclude with a summary of the main contributions and perspectives of future work.

2   Related Work

Learning computer programming can be a lonely, complex, and demotivating process [1, 3, 5]. These issues have been addressed in the last years, with the appearance of several on-line learning environments trying to leverage coding education and make it accessible to everyone,

---

[1] `https://www.tiobe.com/tiobe-index/`

Table 1 MOOCs features comparison.

	Path	Resources (expositives)	Resources (evaluation)	Social
EdX	Yes	Videos	Quiz	F/D
Coursera	Yes	Videos	Quiz/Puz	UP/F/D
Udacity	Yes	Videos	Quiz	Forum
CodePlayer	No	Videos	No	Com/Rec
CodeAcademy	Yes	ICE	ICE	Ach/Badges
Code.org	Yes	Videos	ICE/Puz	Levels
TreeHouse	Yes	Videos	ICE/Puz	Badges
CodeSchool	Yes	Videos	ICE	F/Badges

even those with absolutely no coding experience or knowledge [7, 4]. These environments come in various formats ranging from non-interactive approaches (e.g. YouTube channels, blogs, books) to integrated and interactive solutions (e.g. intelligent tutors, online coding environments).

Nowadays, there is an enormous demand on the technology sector to be up to date with the latest frameworks and languages. Regardless whether you are a coding newbie or a mature developer, you have several options, besides a computer science degree, to improve your programming skills. In this realm, MOOCs (Massive Open Online Courses) and Online Coding Bootcamps are two increasingly popular options for learners to improve their development skills and find work within a relatively short amount of time. While these two are excellent alternative learning contexts, the two options still have very distinct differences [2].

A MOOC is an online course, usually available without charge, where learners can choose their own learning pace and direction. MOOCs are free educational courses often delivered by renowned university professors that typically feature a mix of downloadable readings, quizzes, discussion boards, video content and peer-to-peer assessment. The goal of MOOCs is to reach a much larger audience than traditional courses can accommodate. Often, MOOCs offer certificates for a fee which are awarded on successful completion of a course, and transferable college credit.

An Online Coding Bootcamp, on the other hand, is an intensive and paid course, usually eight to twelve weeks in duration, which offers hands-on training, career guidance and job assistance. These types of platforms involve a greater time commitment for the learner and are more suitable for those who wants to quickly master a specific language (or stack) and get a technical job.

Table 1 compares a few of the most popular online learning platforms/MOOCs on a set of features and tools.

Most of the MOOCs offer learning paths with a list of courses to work through. This feature is very important, especially for those new to programming. Despite the existence of paths, the studied MOOCs don't offer a very rigid structure allowing learners to choose several learning paths during the course. The materials of the courses come in several flavors and are organized in two types: 1) expository resources, such as videos (the most popular format) and HTML/PDF tutorials and 2) evaluation resources, such as interactive code exercises, quizzes and puzzles. Almost all platforms offer videos as a way to disseminate knowledge. These videos include the resolution step-by-step of exercises. This way, learners gain some theoretical/practical skills that can be later consolidated and applied in coding challenges.
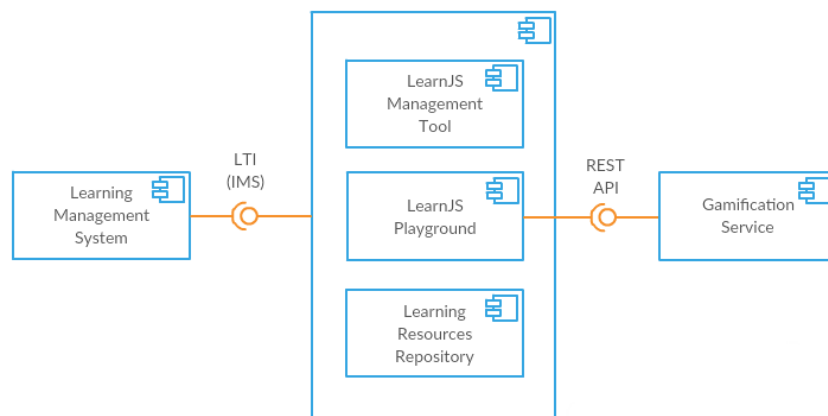
■ **Figure 1** LearnJS component diagram.

These challenges can be run inside of interactive coding exercises components (ICE) giving feedback and support to the learner during the resolution of the exercise (e.g. CodeAcademy, Treehouse). Most of these components are based on cloud IDEs (e.g. Cloud9, Codeanywhere) and integrate some tools like resources sequence, chat and video visualization. Regarding gamification and social features, most platforms adhere to the same components, such as forums (F), learning dashboards (D), user profiles (UP), comments (COM), recommendation (REC), levels and badges. For instance, CodePlayer offers a different approach to learning code by playing code like a video, helping people to learn front-end technologies quickly and interactively. The platform also includes a commenting tool and links to related walkthroughs. CodeAcademy includes a user progress dashboard informing of the current state of the learner regarding its progress in the courses. This platform enhances the participation in the courses by also including achievements (ACH) that are rewarded with badges and users are also able to share completed projects with the rest of the site community and potentially showcase their skills to employers. Except for Code.org, all the platforms have a strong presence in the mobile world, with app versions for Android and iOS.

3   LearnJS

In this section we present LearnJS, a simple and flexible online playground for the teaching and learning of the JavaScript language. The architecture of learnJS is depicted in Figure 1.

At its core, LearnJS is composed by two components used by the two system user profiles:

- **Teachers**: use the **LearnJS Management Tool** to create/select resources to/from the Learning Resources Repository in order to compose a learning activity. Next, they deploy the activity in a Learning Management System.
- **Students**: launch the activity in the LMS and solve it using the **LearnJS Playground**. Beyond the internal gamification features, the playground can benefit from other Gamification Services to foster student's competitiveness and engagement.

The purpose of LearnJS is also to integrate an e-learning ecosystem based on an LMS (e.g. Moodle, Sakai, BlackBoard). For this, it benefits from the interoperability mechanisms to provide authentication directly from the LMS and to submit exercises grades back to the LMS, using the Learning Tools Interoperability (LTI) specification.

In the following sections we detail these two main components in the LearnJS ecosystem: the management tool and the playground.

3.1 LearnJS Management Tool

The LearnJS Management Tool is a Web-based component which will be used by teachers/instructors to submit resources and aggregate them to obtain a composite learning activity. The next section will detail the main aspects of this management tool, more precisely, the GUI component and the resource and activity schemata.

3.1.1 GUI component

The LearnJS Management Tool is a Web-based component based on HTML5 Canvas. Its main purpose is to provide a flexible way for teachers to contribute with new learning resources and allow their aggregation and gamification to define playful learning activities. The final result of this aggregation is a LearnJS manifest with all the necessary information for the correct functioning of the activity in the student's playground.

Another feature of this tool is the capacity for sharing and grading activities. This feature will allow a teacher to share a previously created learning activity in the public space of the LearnJS community. With the grade feature, instructors could score a given activity taken into account the experience that they have with it. This grading will influence the results list after searching.

3.1.2 The resources schema

Teachers can use the LearnJS Management Tool to contribute with new resources. The supported resources in LearnJS follow Sweller and Cooper [6] paradigm based on a learner-centered approach to define a constructivist learning model. This model foster the learning by viewing and learning by doing approaches where educational resources, either expository or evaluative, play a pivotal role. Thus, in LearnJS, resources have two flavours:

- Evaluative: JavaScript challenges to be solved by coding;
- Expositive: Videos or PDF files showing how to master a specific topic.

In this moment, we do not have yet the GUI component finalized. Thus, the submission of a new learning resource should be made through the upload of a JSON file which should comply with the LearnJS official resource schema formalized by a public JSON Schema[2]. The example on Listing 1 shows an evaluative resource for the calculation of a number's factorial.

The JSON document has a simple structure. It contains basic properties for identification and metadata purposes. One of the most important properties is the **type** property. It can assume one of two values: document or exercise. The former requires the **url** property to be set. In this case, the system will load the resource located in that URL. The later requires filling the **exercise** property. This property is composes by the following sub-properties:

- **statement**: the exercise statement formatted in plain text or HTML;
- **hint**: a set of hints to help students to overcome the challenge. By default, they are blocked;
- **code/skeleton**: code skeleton defined by the teacher. Only available by gamification;
- **code/solution**: solution of the challenge submitted by the teacher. Used for input tests injection. Only available after success completion of the exercise by the student;
- **code/tests**: test cases. The input tests are inject in the student's solution and the outputs compared with the provided output tests or with the output generated by the teacher solution.

---

[2] `https://github.com/rqueiros/learnJS`

■ **Listing 1** Resource JSON instance template file.

```
{
 "id":"http://learnJS/resources/125412",
 "title": "Calculate factorial of a number",
 "url": "",
 "type": "exercise",
 "metadata": {
   "author": "Ricardo Queiros",
   "date": "19-04-2018",
   "level": "intermediate",
   "tags": ["recursivity","math"]
 },
 "exercise": {
   "statement": "Create a function that receives one number and
       returns its factorial",
   "hint": ["Verify special cases like 0 that should return 1"],
   "code": {
     "skeleton": "function factorial(x){ return;}",
     "solution": "function factorial(x){if(x===0){return 1;}
       return x*factorial(x-1);}",
     "tests": [{"in": "4","out": "24"},{"in": "0","out": "1"}]
   } } }
```

.

3.1.3   The activity schema

Teachers can also perform other operations in the management tool, such as the creation of activities.

An activity combines a set of resources of several types (evaluative, expositive) with gamification attributes. Listings 2 shows an activity JSON instance for learning JavaScript arrays.

An activity JSON file is composed by several properties. We highlight two:

- **levels**: can be considered as sub-activities composed by a set of resources identified in the **resources** sub-property. Students should see and solved the respective resources of the level. The completion of the level and the respective unlock of the next level is granted after the student solved a specific percentage of evaluative resources defined in the **perc** property of the level.

- **gamify**: a set of attributes that can be assigned to resources. After a success completion of an evaluative resource, students can be awarded in multiple forms. Hence, the **award** property can have one of the following values:

  - **HintExtra**: gives an extra point to the learner. The learner can spend the hint points on any exercise by unhiding the hint associated;

  - **ShowNextSkeleton|ShowAllSkeleton**: gives the learner the ability to unhide the code skeleton associated to the next (or all) gamified resources;

  - **UnlockLevel|UnlockAllLevels**: gives the learner the ability to unlock the next (or all) level.

■ **Listing 2** Learning activity JSON instance.

```
{
 "id": "http://learnJS/activities/129387",
 "title": "Learn the basics of Arrays",
 "metadata": {
   "author":"Ricardo Queiros",
   "date":"19-04-1975",
   "level":"basic",
   "tags":["arrays"]
 },
 "levels": [
  {"id":"1", "name":"Basic operations", "perc":"75",
          "resources": ["...resources/125412", "..."]},
  {"id":"2", "name":"Sort", "perc":"50", "resources":["..."]}
 ],
 "gamify": [
   {"resource": ".../resources/125412", "award":"HintExtra"},
   {"resource": ".../resources/225232", "award":"ShowNextSkeleton"}
 ]
}
```

3.2 LearnJS Playground

The LearnJS Playground is a Web-based component which will be used by students/learners to browse learning activities and interact with the compound resources. Here students can see videos of specific topics and solve exercises related with those topics with automatic feedback on their resolutions. The architecture of the playground is shown in Figure 2.

The playground is composed by three main components:

1. **Editor**: allows students to code their solutions in a interactive environment;
2. **Evaluator**: assess the student's solution based on static and dynamic analyzers;
3. **Gamification Engine**: gamifies the learning activity with the management of levels and several awards.

For the **Editor component**, the playground uses Ace (maintained as the primary editor for Cloud9 IDE) which can be easily embedded in any web page and JavaScript application. The editor is properly configured for the JavaScript language and supports the Emmet toolkit for the inclusion of dynamic JavaScript snippets.

Ace editor can display errors on the editor itself but does not handle language dependencies. A parser needs to be used to detect errors and determine their positions on the source file. There are several tools that can improve code quality. One of such cases is code linters. Linters (e.g JSLint, JSHint) can detect potential bugs, as well as code that is difficult to maintain. These static code analysis tools come into play and help developers spot several issues such as a syntax error, an unused variable, a bug due to an implicit type conversion, or even (if properly configured) coding style issues. LearnJS uses JSHint to accomplish this behavior. While static code analysis tools can spot many different kinds of mistakes, they can not detect if your program is correct, fast or has memory leaks. For that particularly reason, LearnJS combines JSHint with functional tests (based on test cases). For this kind of tests, and since the code is written in JS and the context is the browser, we use a simple approach by iterating all the case tests and applying the eval function for tests injection.
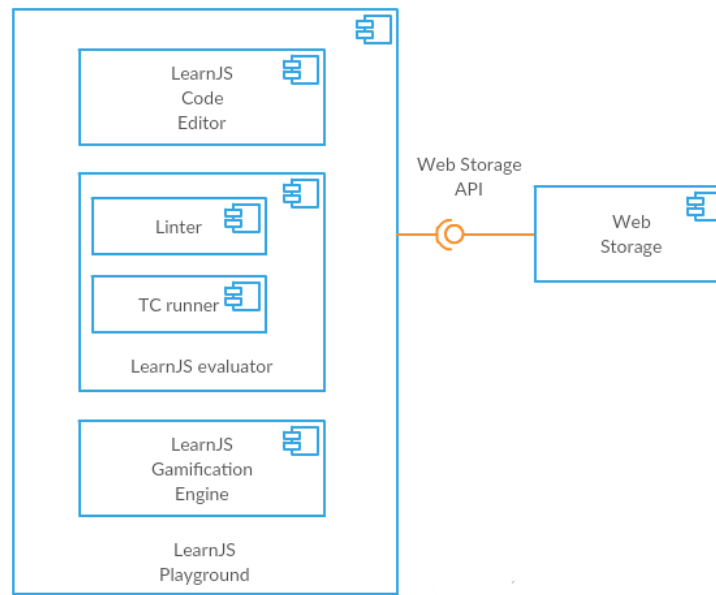
■ **Figure 2** LearnJS Playground component diagram.

Both analyzers (linter and Test Case runner) are subcomponents of the **LearnJS evaluator component** that runs exclusively on the client side. This approach avoids successive round-trips to the server which affects negatively the user experience.

Lastly, the **Gamification Engine component** is responsible for loading/parsing the LearnJS manifest and fetching resources from the learning resources store. If levels are defined, the engine sequences and organizes the resources properly. Upon completion of evaluative resources from students, the engine deals with all the logic associated with the respective awards by unhiding/unlocking features of next challenges. Finally, the component send the results back to the server.

At this moment, we have a simple running prototype. The source code is available at a GitHub repository. Figure 3 shows the frontend GUI of the playground.

4   Conclusions

In this paper we present LearnJS as a flexible playground for JavaScript learning. Since we are in the beginning of implementation, the paper stresses the design of the platform divided in two main components: the management tool and the playground. In the former, teachers can contribute with new exercises and bundle related exercises in learning activities. All these entities were formalized using JSON schemata. The later, allows students through a sophisticated and interactive UI, to see and solve educational resources (mostly, videos and exercises). In order to engage students, the platform can be configured to gamify resources through the subgrouping of activities in levels, the assignment of awards and the exhibition of a global leaderboard.

The main contributions of this work is the design of a platform with interoperability concerns in mind and the respective schemata for the simple concepts of educational resources and activities.

As future work we intend to create a more mature prototype by creating a introductory course for novice students to learn JavaScript. Then, for validation purposes, we intend to
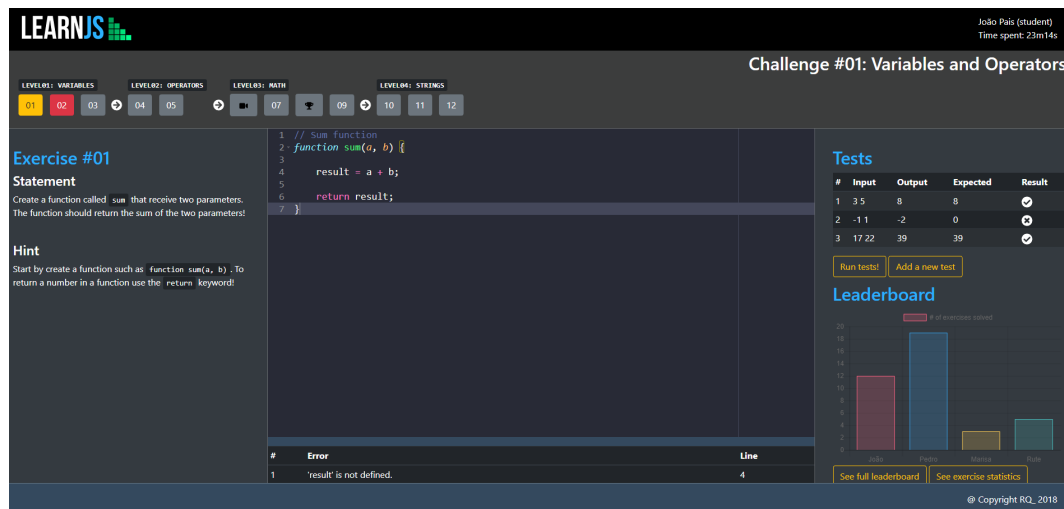
■ **Figure 3** LearnJS Playground UI.

use the platform in real classes and receive student's feedback. After this process, our idea is to work on the management tool. Regarding the playground, our intentions is to maintain it very simple, avoid at maximum the communication with the server and improve the game mechanics of the engine.

───── **References** ─────

1   Kirsti M. Ala-Mutka.   A survey of automated assessment approaches for programming assignments.   *Computer Science Education*, 15(2):83–102, 2005.   `doi:10.1080/08993400500150747`.

2   Gemma Church. MOOCs versus coding bootcamps. `https://www.class-central.com/report/moocs-versus-coding-bootcamps/`, 2016. [Online; accessed april 19th, 2018].

3   Jackie O'Kelly and J. Paul Gibson.   Robocode & problem-based learning: A non-prescriptive approach to teaching programming.   *SIGCSE Bulletin*, 38(3):217–221, 2006. `doi:10.1145/1140123.1140182`.

4   Pedro Xavier Pacheco and António Coelho.   Computer-based assessment system for e-learning applied to programming education. In *4th International Conference of Education, Research and Innovation*, pages 3738–3747, 2011.

5   Anthony Robins, Janet Rountree, and Nathan Rountree.   Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003. `doi:10.1076/csed.13.2.137.14200`.

6   John Sweller and Graham Cooper. The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1):59–89, 1985. `doi:10.1207/s1532690xci0201_3`.

7   Elena Verdú, Luisa M. Regueras, María J. Verdú, José P. Leal, Juan P. de Castro, and Ricardo Queirós. A distributed system for learning programming on-line. *Computers and Education*, 58(1):1–10, 2012. `doi:10.1016/j.compedu.2011.08.015`.

Moozz: Assessment of Quizzes in Mooshak 2.0

Helder Correia
CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal
up201108850@fc.up.pt
 https://orcid.org/0000-0002-7663-2456

José Paulo Leal
CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal
zp@dcc.fc.up.pt
 https://orcid.org/0000-0002-8409-0300

José Carlos Paiva
CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal
up201200272@fc.up.pt
 https://orcid.org/0000-0003-0394-0527

Abstract

Quizzes are a widely used form of assessment, supported in many e-learning systems. Mooshak is a web system which supports automated assessment in computer science. This paper presents Moozz, a quiz assessment environment for Mooshak 2.0, with its own XML definition for describing quizzes. This definition is used for: interoperability with different e-learning systems, generating HTML-based forms, storing student answers, marking final submissions and generating feedback. Furthermore, Moozz also includes an authoring tool for creating quizzes. The paper describes Moozz, its quiz definition language and architecture, and details its implementation.

1 Introduction

Mooshak [5] is a web system that supports automated assessment in computer science. It evolved from a programming contest management system, supporting different contest models, to a pedagogical tool used in introductory computer science courses. Although Mooshak was initially targeted for text-based computer programming languages, it was later extended to support visual languages, such as EER (Extended Entity-Relationship) and UML (Unified Modeling Language).

Quizzes are a widely used form of assessment, not only in computer science, and they are widely supported among e-learning systems. Quizzes can also be used in computer science

contests, in particular in those targeted to younger students, to develop their computational thinking skill, as is the case of Bebras [2]. Hence, a system supporting automated assessment in computer science, both in competitive and pedagogical settings, should also have a quiz assessment environment.

In fact, the previous version of Mooshak already has an incipient form of quiz assessment. Nevertheless, it lacks support for several types of questions, standard quiz interoperability languages, and integration with other assessment modes. Meanwhile, version 2.0 has a pedagogical environment, named Enki [6], that integrates different kinds of assessment in a single course, including code and diagram assessment.

This paper presents Moozz, an assessment environment for quizzes integrated into Mooshak 2.0. Moozz supports all the standard question types, including multiple choice (select one or multiple), gap filling, matching, short answer, and essay, as well as questions with media files (e.g, images, sounds, and videos). Moozz uses its own XML definition for describing quizzes, named Moo, providing eXtensible Stylesheet Language Transformations (XSLT) to convert to and from other quiz formats, such as IMS Question and Test Interoperability specification (QTI) [8] and the GIFT format[1].

An authoring tool is also included in Moozz to facilitate the creation of quizzes. This tool allows to import and export quizzes in the supported formats, insert multimedia content, and add questions, answers and feedback messages.

The remainder of this paper is organized as follows. Section 2 presents Mooshak 2.0, and the formats and standards supported by Moozz. Section 3 describes the architecture of Moozz and details its implementation. Finally, Section 4 summarizes the contributions of this work and identifies opportunities for future developments.

2   Background

This paper presents Moozz, a quiz assessment environment that aims to integrate in the pedagogical tool of Mooshak 2.0, named Enki [6]. This environment has several features, such as the compatibility with IMS QTI specification and the GIFT format, and the support for Bebras quiz competitions. This section aims to provide some background on Mooshak, particularly on the tool in which Moozz integrates, and to describe the supported formats.

2.1   Mooshak 2.0

Mooshak [5] is a system for managing programming contests on the web, which provides automatic judging of submitted programs, submission of clarification requests, reevaluation of programs, tracking of printouts, among many other features. It supports a wide range of programming languages, such as Java, C, VB, and Prolog.

Even if Mooshak was initially designed to be a programming contest management system for ICPC contests, educators rapidly found its ability to assist them in programming courses [3] to give instant feedback on practical classes, evaluate and mark assignments, among other uses. This has motivated the development of several extensions specifically for learning, such as a plagiarism checker and an exam policy.

Recently, Mooshak was completely reimplemented in Java with Graphic User Interfaces (GUIs) in Google Web Toolkit (GWT). Besides the changes in the codebase, Mooshak 2.0 gives special attention to computer science learning, providing a specialized computer science languages learning environment – Enki [6] –, which not only supports exercises using typical programming languages, but also diagramming exercises.

---

[1] `https://docs.moodle.org/25/en/GIFT_format`

Enki blends assessment and learning, integrating with several external tools to engage students in learning activities. Furthermore, Enki's GUI mimics the aspect of an IDE, and attempts to achieve their powerful extensibility. As a typical IDE, such as Eclipse and NetBeans, the GUI of Enki is divided into regions, each one containing several overlapping windows, organized using tabs. These regions are resizable, and their windows can be moved among different regions. A window holds a unique component, capable of communicating with other components. Therefore, it is possible to add any number of components required by a specific assessment environment and link them to the evaluation engine with relative ease. This has already been done for the diagram assessment.

2.2 Question and Test Interoperability (QTI)

The IMS Question and Test Interoperability (QTI) specification describes a data model for representing question and test data, as well as their corresponding results. This specification enables authoring and delivering systems, question banks, and Learning Management Systems (LMSs) to exchange question and test data [8]. Among other things, this common format can facilitate populating question banks, transmitting results and information about the learner between the various components of an e-learning ecosystem, and incorporating questions designed by several IMS QTI users into a single assessment.

The IMS QTI uses XML to store information about assessments. Its data model can be seen as a hierarchy of elements whose contents and attributes are XML tags [7]. There are three key elements in this model: `assessment`, `section` and `item`. The `assessment` element contains a set of questions, which can be organized using `section` elements. The `section` element indicates a group of questions, enabling authors to separate each subtopic and calculate the score obtained for each section as well as the overall score. An `item` is a question with all the associated data, such as score, answers, layout and feedback.

The results of the IMS QTI are specific to a participant, but can contain data of more than one assessment. The core data structures for reporting results are the `summary`, which contains global statistics of the assessment, such as the number of attempts, and the results of the internal tree of the `assessment`, `section` and `item` elements.

2.3 Bebras

Bebras is a community building model for concept-based learning of informatics [2]. It is designed to promote informatics learning in school through short tasks about simple concepts [1]. These tasks are the main component of Bebras. They are generally accompanied by a story or media element, to attract the attention of the children, and try to teach one or more concepts of informatics. Besides covering a wide range of topics, these tasks can be designed to help in the development of core computational thinking skills, such as abstraction, decomposition, evaluation and generalization.

From the practical point of view, Bebras tasks are just quiz questions with multimedia elements. The Bebras model can be used both in competitive and learning environments. Furthermore, it has already been used in several individual and team competitions across the globe.

2.4   General Import Format Template (GIFT)

The General Import Format Template (GIFT)[2] is a format created by the Moodle community to import quiz questions from a text file using markup language [4]. It supports multiple-choice, true-false, short answer, matching, fill in the blank and numerical questions.

The markup language of GIFT uses blank lines to delimit questions. Questions are written with the following syntax `::title::  question { answers }`. The syntax of the answers depends on the type of question. For instance, in multiple choice questions the correct answer(s) are prefixed with an equal sign (=) and the wrong answers with a tilde. To add feedback, an hash (#) can be used after each answer followed by the feedback message. Comments are preceded by double slashes (//) and are not imported. A full description of the language can be found on the Moodle page dedicated to the format.

3   Moozz

Moozz is an assessment environment for quizzes. It supports multiple choice (select one or multiple), gap filling, matching, short answer, and essay questions, as well as questions with media files. Moozz has its own XML language, named Moo, for storing and interchanging quizzes. Moo can be converted to and from different formats, such as GIFT and IMS QTI. Hence, the questions present in assessments can be saved in a question bank and reused in other assessments. It also contains an authoring tool for creating quizzes complying with Moo.

3.1   Authoring Tool

Moozz provides an authoring tool for quizzes. This tool can import and export quizzes in one of the following formats: Moo (XML), GIFT (plain text), IMS QTI (XML), and JSON. Besides that, it allows to create question groups, and add, edit, and remove questions. Questions can also have notes for each possible choice, question, or group. The quizzes are stored in Moo XML in Mooshak.

The GIFT format was extended to support the concept of groups in Moozz. Each group starts with `$name:numberQuestion`. `name` is the name of group whereas `numberQuestion` is the number of questions in the group to be selected for the exam. Two blank lines must be used to separate two groups of questions and one blank line must be left to separate each question from the next one.

Moozz supports several kinds of questions, such as: single-select answer, multiple-select answers, short answer, numerical, fill in the blank, matching, and essay questions. In `single-select answer` questions only one alternative can be marked as right and erroneous responses can be scored negatively. By default, a wrong answer has score zero and the correct answer scores one, but this can be modified in the Quiz Editor. A `multiple-select answer` question is used when two or more answers must be selected in order to obtain full credit. The multiple-select answer option is enabled by assigning partial answer weight to multiple answers, while allowing no single answer to receive full credit. The rating of each option can be defined and by default, a wrong answer has score zero. In `short answer` type, all the possible answers must be written, and it will be 100% credited if it matches exactly any of the correct responses, or zero, otherwise. A `numerical question` is similar to a `short answer` question, but the answer is a number. Numerical answers can include an

---

[2] `https://docs.moodle.org/25/en/GIFT_format`

Figure 1 Screenshot of the Moozz authoring tool.

error margin, an interval and a precision for a correct answer. A `fill in blank` question is like a `short answer` question, but the answers are presented in an HTML element select. In the `boolean question` type, the answer indicates whether the statement is true or false. There can be one or two feedback strings. The first is shown if the student gives the wrong answer, and the second if the student gives the right answer. In `matching` questions, there are two arrays. One array with the keys and another with the values. Each key matches one and only one value. These questions do not support feedback messages. Finally, an `essay question` allows any text response, and is graded manually.

Figure 1 presents the first version of the authoring tool embedded in Mooshak administrator GUI.

3.2 Architecture

There are four types of XML files stored in Moozz which are Moo-compliant: `QuizDb`, `Exam`, `Answers`, and `CorrectedExam`. `QuizDb` is the question bank XML file containing all the questions used in assessments, which is created in the authoring tool of Moozz or imported in one of the supported formats. `Exam` is the XML file that contains the subset of questions of an actual exam. `Answers` has the answers of a student to an exam. `CorrectedExam` has the exam with feedback, classification and grade for each question.

Some of these files are generated from each other during the quiz assessment workflow (e.g., `Exam` is generated from an XSL Transformation applied to `QuizDb`). The `Exam` and `CorrectedExam` are also transformed into an user-friendly format to be displayed to the student using XSLT. Therefore, most of the work in Moozz consists of XML manipulations. Figure 2 presents the architecture of Moozz, particularly the transformations conducted in its core.

When the user request for a new exam, a transformation `QuizDbToExam` is applied on `QuizDb`, which is present on the problem directory. This transformation aims to select randomly N questions from the whole question bank. The outcome of this transformation is an `Exam` XML file, which is stored in the submission directory reserved to the current participant for subsequent requests. Before being sent to user, this XML is transformed into HTML through an `ExamToHTML` transformation. After solving the exam, answers are sent to Moozz in JSON and converted to XML in a Java class `JSONHandler`. The result is an `Answers`
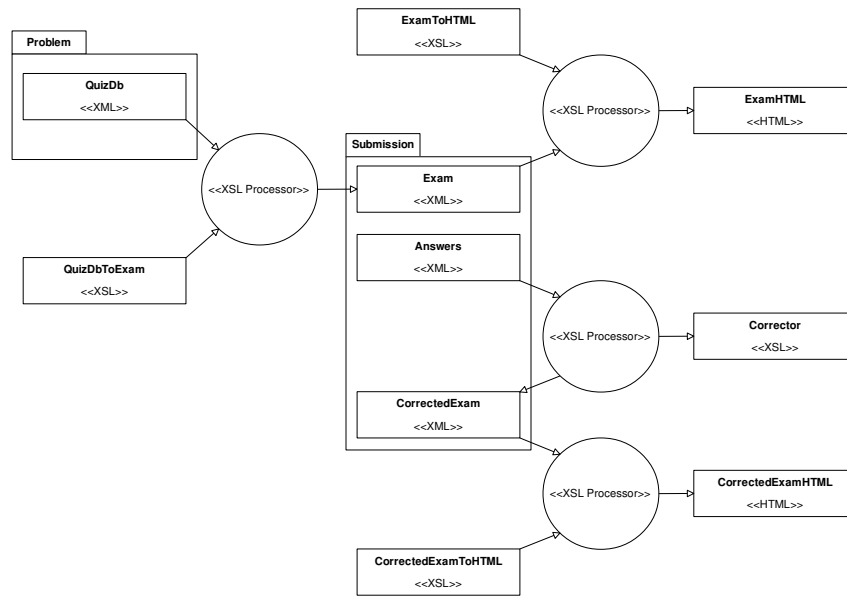
■ **Figure 2** Diagram of the architecture of Moozz, highlighting the XSL Transformations carried internally.

XML file, which is also stored in the submission directory. The quiz evaluation is then executed. The evaluation consists of applying a `Corrector` transformation to `Answers` XML. This transformation outputs a `CorrectedExam`, which is saved in the same directory. Finally, `CorrectedExam` XML is converted to `CorrectedExamHTML` through `CorrectedExamToHTML` to present the feedback to the student.

3.3   User Interface

The client-side of Moozz follows the Model-View-Presenter (MVP) design pattern, integrating seamlessly in Enki within a single window. Since Enki uses GWT, the Viewer is also implemented with it. In this sense, the component defines a Java interface `MoozzView` with methods to update the view, such as `setQuiz(String html)`, and a class named `MoozzViewImpl` that implements the interface and displays the quiz. The presenter part is implemented in `MoozzPresenter`. This class receives the commands inputted by the user in the view, and invokes the necessary methods on the RPC interface of the Moozz service.

The data received from the server is either an `ExamHTML`, if the exam is not solved, or a `CorrectedExamHTML`, if the exam was already submitted. These HTML files are just an excerpt of an HTML, not a complete HTML page, containing the formatted elements to be displayed to the user. The excerpt is inserted into the container reserved for the quiz, after some Javascript pre-processing steps and CSS styling to make it more user-friendly. The answers submitted by the students are sent in an XML-formatted string complying with the Moo language.

On multiple choice questions, feedback is displayed only for the selected answer. For true-false questions, there can be one or two feedback strings. The first is shown if the student gives the wrong answer. The second if the student gives the right answer. Figure 3 presents an example of feedback in Moozz.
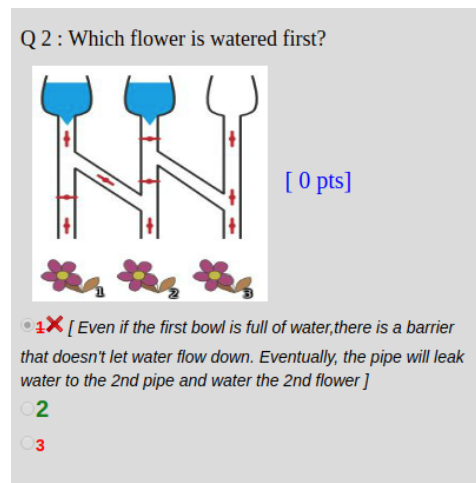
■ **Figure 3** Example of feedback presented in Moozz.
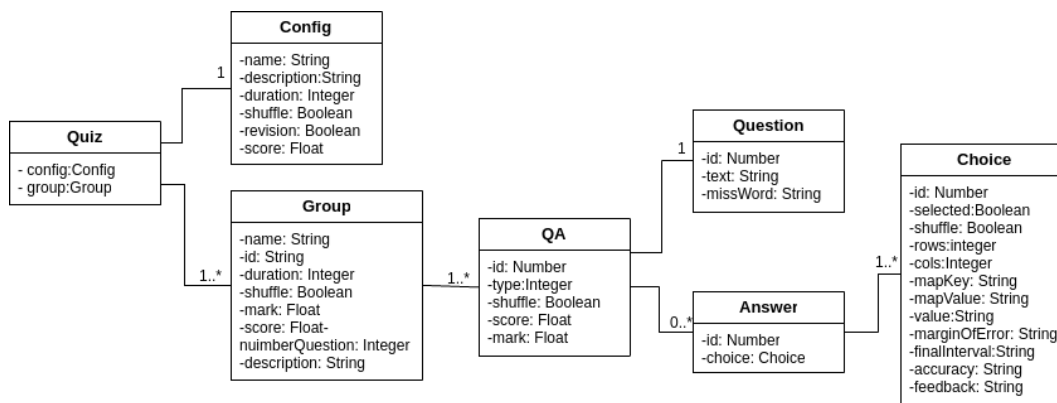
■ **Figure 4** Data model of the Moo language.

3.4 Moo Language

Moozz is able to import quizzes in different formats, therefore it is required a common quiz format, capable of storing the quiz and its configurations (e.g., time and number of questions per exam). A natural candidate for this role is the Question and Test Interoperability (QTI) standard. However, QTI revealed to be too complex and does not support configurations needed for quizzes in Mooshak, so a new language based in QTI is proposed, named Moo. As QTI, Moo is an XML language with its own XML Schema definition. As depicted in the simplified data model of Figure 4, Moo stores questions and settings such as the duration, and name of the quiz. Questions are organized in groups and each group stores information, such as the name, grade, and number of questions to appear on the exam.

Questions and answers of a group are stored in a type called `QA`. This type saves the question and its answers (if applicable) as well as configurations, such as the name of the question, the type, and the score, which is the sum of the positive scores of the answers. Each `QA` has one or more elements of type `Choice`. The `Choice` elements save different data, according to the type of the question. For example, in multiple, single, short-answer and boolean types, it includes the response text, feedback for each option, score and mark. The recorded data for numeric types depends on their subtype: exact answer (response value and

the margin of error), range answer (initial and final interval values), and precision answer (value and accuracy). In questions of type matching, it saves the key and the value. The essay type just saves the question, since the answer is a free-text introduced by the student. The essay type does not support feedback. The questions and answers texts accept inline HTML tags for including media or text formatting.

4   Conclusions and Future Work

Mooshak is a system that supports automated assessment in computer science. It has been used both in competitive and learning environments, supporting the assessment of visual and text-based programming languages. This paper presents an assessment environment for quizzes in Mooshak 2.0, named Moozz. Moozz supports all the standard question types, including multiple choice, true/false, short answer, numerical, fill in the blank and matching, and questions with media formats. It uses XSL Transformations to support the most common quiz formats, namely IMS QTI and GIFT.

Moozz includes a quiz authoring tool that is embedded into the administrator GUI of Mooshak. This editor is capable of importing and exporting quizzes in different formats, inserting multimedia elements, and add any of the supported question types with feedback information for each answer.

This environment is a work in progress. Currently, the development phase is almost completed, only missing the XSL Transformation to comply with IMS QTI. The next phase is the validation, which will be conducted in a real exam scenario with text, visual and quiz based exercises.

References

1   Valentina Dagienė and Sue Sentance. It's computational thinking! Bebras tasks in the curriculum. In *International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 28–39, 2016.

2   Valentina Dagiene and Gabriele Stupuriene. Bebras-a sustainable community building model for the concept based learning of informatics and computational thinking. *Informatics in Education*, 15(1):25, 2016.

3   Ginés Gárcia-Mateos and José Luis Fernández-Alemán. A course on algorithms and data structures using on-line judging. *ACM SIGCSE Bulletin*, 41(3):45–49, 2009. `doi:10.1145/1562877.1562897`.

4   Gaurav Kumar and Anu Suneja. Using Moodle – an open source virtual learning environment in the academia. *International Journal of Enterprise Computing and Business Systems*, 1(1):1–10, 2011.

5   José Paulo Leal and Fernando Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

6   José Carlos Paiva, José Paulo Leal, and Ricardo Alexandre Queirós. Enki: A pedagogical services aggregator for learning programming languages. In *Conference on Innovation and Technology in Computer Science Education*, pages 332–337, 2016.

7   Niall Sclater and Rowin Cross. What is IMS question and test interoperability. *Retrieved*, 7(22), 2003.

8   Colin Smythe, Eric Shepherd, Lane Brewer, and Steve Lay. IMS question & test interoperability: an overview, 2002. Final Specification, version 1.2.

Raccode: An Eclipse Plugin for Assessment of Programming Exercises

André Silva
Faculty of Sciences, University of Porto, Portugal
up201007410@fc.up.pt
https://orcid.org/0000-0002-7663-2456

José Paulo Leal
CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal
zp@dcc.fc.up.pt
https://orcid.org/0000-0002-8409-0300

José Carlos Paiva
CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal
up201200272@fc.up.pt
https://orcid.org/0000-0003-0394-0527

Abstract

IDEs are environments specialized in support during the development of programs. They contain several utilities to code, run, debug, and deploy programs quickly. However, they do not provide the automatic assessment of programming exercises, which is required in both learning and competitive programming environment. Therefore, IDEs are often underestimated in these contexts and replaced by basic code editors. Yet, IDEs have unique features which are essential for programmers, such as the debugger or the package explorer. This paper presents Raccode, a plugin for assessment of programming exercises in Eclipse. This plugin integrates with Mooshak to combine the diverse capabilities of an IDE, like Eclipse, with the automatic evaluation of exercises, clarification requests, printouts, balloons, and rankings. It can be used both in competitive and learning environments. The paper describes Raccode, its concept, architecture and design.

1   Introduction

Integrated Development Environments (IDEs) are applications specialized in supporting programmers during the development of software, either simple or complex. They are designed to include all programming tasks in a single application. Therefore, IDEs provide a central interface containing every tool that a developer should need, including a code editor, a package explorer, a compiler, a debugger, and several build automation tools.

In competitive and learning programming environments, the automatic assessment of submitted programs is essential. Students and contest participants need timely feedback on their attempts, that can not be guaranteed by human judges. Several tools have been developed to provide this feature, which generally include embedded code editors trying to mimic IDEs. Hence, IDEs are often underestimated and set aside in these contexts to streamline the process.

However, this decision can have very negative outcomes. Besides not adapting future programmers to the tools that they will have to use later, it can slow down the development of the right solution to an exercise. For instance, the debugger can help to find bugs in programs, by running it step by step, stopping at some event or specified instruction (i.e., a breakpoint), and tracking the values of variables.

This paper presents Raccode, an Eclipse plugin that combines the several features of an IDE with the key characteristics of a tool that provides automatic assessment of programming exercises both in competitive and learning environments. Raccode integrates the REST API of Mooshak 2.0, providing Eclipse with automatic evaluation of programming exercises, support for clarification requests, tracking of printouts and balloons, rankings list view, among many others.

Eclipse is one of the most used IDEs among software developers. As an open source software, developers can contribute to the project or share their own products in the form of a plugin [10]. Eclipse has an environment called Eclipse Marketplace that provides an extensive listing of Eclipse-based solutions, such as plugins and bundles, which can be installed directly from the workspace using the Marketplace Client. These features together with the environments that Eclipse offers, such as Rich Client Platform – RCP [5, 3] – and Plug-in Development Environment – PDE [6] –, have led us to choose it as the first IDE to integrate Raccode. However, support for other IDEs is already planned.

The remainder of this paper is organized as follows. Section 2 reviews some systems that combine automatic evaluation with IDE-like features. Section 3 describes Mooshak and details its REST API, included in version 2.0. Section 4 presents Raccode, its concept, architecture, and design. Finally, Section 5 summarizes the main contributions of this work and next steps.

2   State of the Art

There are several online platforms and tools, such as CodeChef[1], CodinGame[2], or Enki [8], that combine automatic assessment of programming exercises with an IDE-like user interface and some of its features. However, they just include a very limited set of features when compared to IDEs, including a code editor with a weak support for code completion and a console log where the output is displayed.

---

[1] `https://codechef.com/`
[2] `https://codingame.com/`

CodeChef is an online competitive programming platform. It supports more than 35 programming languages, including Java, Javascript, C, C++, Python, and Pascal. The development environment contains a code editor, a selector for programming language, and a widget to provide custom inputs. The platform has two modes: practice and contest. The practice mode categorizes problems by difficulty, allowing users to solve at their own rhythm. The contest mode proposes a series of problems for participants to solve under specific time constraints. Users are ranked according to the number of problems solved, breaking ties with the total amount of time spent solving them. Every month more than 30 programming contests are realized. Users can award ranking points in both modes.

CodinGame is an online platform where programmers can learn and compete through game-based challenges. Most of these challenges require the user to develop a software agent to control the behavior of a character in a game environment, and provide a 2D game-like graphical feedback. The agent programmed by the player must pass all test cases (public and hidden) to solve the puzzle. Players can choose one of the more than 20 programming languages available. Once the exercise is solved, players can access, rate, and vote on the best solutions. The interface to develop agents presents the statement of the exercise on the left as well as the movie player, and the code editor and test cases on the right. The widget containing the test cases supports custom input and displays the output log, once the test runs.

Enki is a web-based learning environment with an IDE-like graphic user interface. It integrates with several kinds of tools. These tools include a gamification service to provide gamification features to students, an educational resources sequencing service to offer different learning paths, and an evaluator engine to give automatic feedback to students' solutions. The user interface includes windows for a code editor, a console log output, an error list, a test case editor, and a ranking list.

3 Mooshak 2.0 REST API

Mooshak [4] is a web-based system for automatic assessment in computer science. It was primarily designed for managing programming contests, such as ICPC contests, but the need for automatic assessment in pedagogical contexts has led to its adoption in computer science education. Since then, Mooshak assists educators in programming courses, providing instant feedback on practical classes and exams.

Recently, Mooshak 2.0 has been released. This version is a complete reimplementation in Java and Google Web Toolkit (GWT) of the initial codebase. However, it also adds several new features, including a learning environment – Enki [8] –, a diagram assessment environment – Kora [1] –, and a REST (Representational State Transfer [2]) API.

The REST API of Mooshak 2.0 uses Jersey[3], an open-source framework that is the reference implementation of the Java API for RESTful Web Services, extending it with several features to further simplify RESTful service. Jersey provides a Core Server to build annotation-based RESTful services, and to support JSON and the Java Architecture for XML Binding. It also includes a Core Client to facilitate the communication with REST services.

Mooshak 2.0 REST API contains endpoints for authentication and authorization (`auth`), `contests`, `problems`, `questions`, `printouts`, `balloons`, `languages`, and `submissions`. Most of the endpoints consume and produce JSON and XML, but some require other

---

[3] `https://jersey.github.io/`

Table 1 Main endpoints of the REST API of Mooshak.

Method	Endpoint	Consume	Produce	Description
POST	auth/login		JSON/XML	Authentication into a contest
GET	data/contests/**contestId**/rankings		JSON/XML	View rankings of a contest
GET	data/contests/**contestId**/problems		JSON/XML	List all problems of a contest/course
GET	data/contests/**contestId**/problems/**problemId**/view		JSON/XML	View a problem of a contest/course
POST	data/contests/**contestId**/problems/**problemId**/evaluate	form-data	JSON/XML	Evaluate a program of a contest/course
GET	data/contests/**contestId**/languages		JSON/XML	List all languages of a contest/course
GET	data/contests/**contestId**/languages/**languageId**		JSON/XML	Get a language of a contest/course
GET	data/contests/**contestId**/submissions/**submissionId** /evaluation-summary		JSON/XML	Get the summary of an evaluation
GET	data/contests/**contestId**/questions	JSON/XML	JSON/XML	List all questions of a contest/course
POST	data/contests/**contestId**/questions	JSON/XML	JSON/XML	Create a question in a contest/course
PUT	data/contests/**contestId**/questions/**questionId**	JSON/XML	JSON/XML	Update a question in a contest/course
POST	data/contests/**contestId**/printouts	form-data	JSON/XML	Create a printout in a contest
POST	data/contests/**contestId**/balloons	JSON/XML	JSON/XML	Create a balloon in a contest

formats, such as Form Data (e.g., the evaluation endpoint receives a file as input). The most important endpoints are summarized in Table 1.

The authentication to the API uses JWT (JSON Web Token)[4], a compact URL-safe form of representing claims that are transferred between two parties. Once the REST endpoint for login receives a request from a client, it extracts the user name, password and domain to which the user is attempting to connect, and leverages the work on the `AuthManager` which checks the credentials against the database. If it succeeds, a JWT Token is generated and sent back to the client, otherwise an error is returned. Thereafter, all requests must have the JWT Token in the `Authorization` header.

4   Raccode

The operation of the Raccode can be summarized by the sequence diagram presented in Figure 1. It starts by connecting to a server, using an host and port defined in Eclipse preferences. If no problem occurs and the connection is successful, this data is stored in the registries, so that it is not necessary to configure the server every time the plugin is started. From this moment, the user can proceed with the authentication in a contest.

There are two ways to login into a contest: via Eclipse wizards menu, following the path `File -> New -> Other... -> Raccode -> Login`; or just clicking on the button present on the toolbar with the label `New Problem`. If the user chooses a problem from a different contest, he needs re-authenticate. Nevertheless, the token given by server is also stored in a registry and, if it is the same among different contests, it is just needed to select the other

---

[4] `https://tools.ietf.org/html/rfc7519`
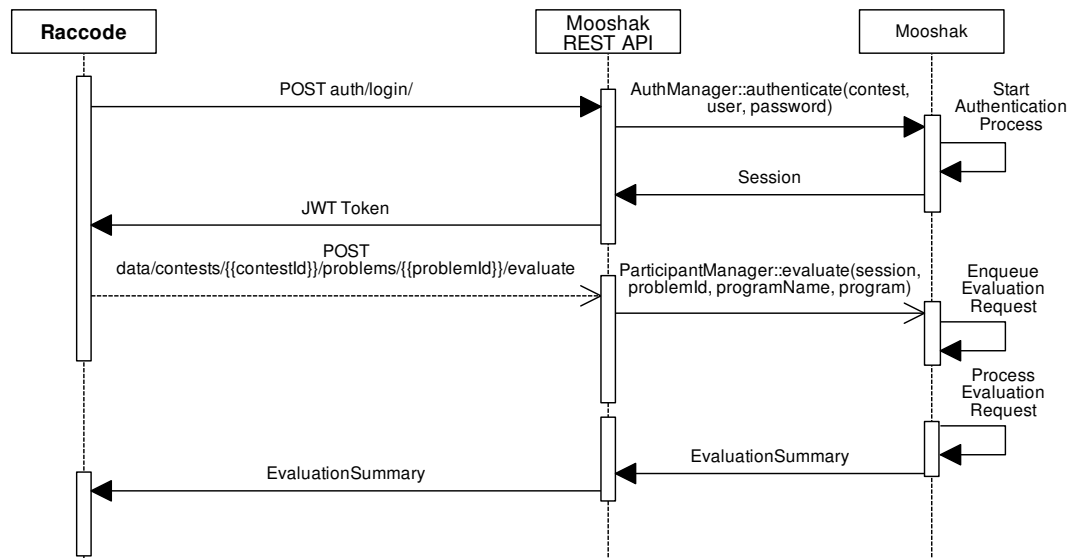
Figure 1 UML sequence diagram of an evaluation.

contest and click `Login`. If the problem belongs to the same contest, no action is required since Raccode keeps the JWT token given to the user for an hour, refreshing it when the time expires but the user remains active.

The second page of the login wizard presents a menu with the available problems and languages, allowing the user to select a problem. When pressing the `Finish` button, the project is created and added to Package Explorer. If the local machine does not have the selected language installed, an error message is presented explaining the situation and the project is not created.

The Problem view presents the problem statement, either in HTML or PDF. While solving the problem, the user can test his program with test cases received from the server (public tests) or its own test cases (user-defined tests). If the outputs do not match, he can always use the debugger to identify the problem.

Finally, to submit the code the user can use the menu `Raccode -> Submit` or click on the `Submit` button on the Eclipse toolbar. The program is then sent to the evaluator on server and a summary is returned, giving the user feedback and informing him if his program was accepted or not.

Raccode presents several tools on the perspective, including the ranking of the contest, the progress of the user (e.g., the number of attempts for the current problem, problems solved, among others), a listing of questions and answers of the selected problem as well as a means to submit questions, and listings for balloons and printouts if it is a contest.

In a general way, the Raccode plugin makes the `bridge` between Mooshak 2.0 and Eclipse. The following subsections present how Raccode integrates with the Mooshak 2.0 REST API and its design.

4.1 API integration

Mooshak has a REST API that enables a client to send HTTP requests to the server and get information easily. The documentation of the API describes how requests should be made, providing an example of a request and a response for each endpoint. Raccode consumes this API exactly as documented, displaying a message in a label or in a pop-up window if an exception occurs while making a request.

The project structure is split into two parts: one for requests and another for the User Interface (UI). By dividing the project into these two parts, it provides extensibility for other IDEs, since the API integration can be reused. The only code that needs to be modified is the UI part. The HTTP requests are performed using `HttpsURLConnection` library. The UI was developed using Standard Widget Toolkit (SWT) [9], a graphical widget toolkit to use with the Java platform. The aspect of the perspective is described in the next subsection.

The `request` package is divided in several classes corresponding to the resources being consumed from the REST API. Each of these classes leverages the request on the adequate method of the `RequestSender`, depending on the HTTP method being used. For instance, the `Auth` class contains methods to authenticate/authorize users. The `login` method invokes the `post` method of the `RequestSender`, which issues the request and processes the response to JSON, returning it. If everything goes fine the user gets logged in, and the response contains a token with a duration of an hour. During this time, every time the user submits a solution or wants to choose a new problem, he doesn't need to log in again. In the same way, if the user doesn't close the application or make logout past this hour, the token is only refreshed and the user can keep going to work. This is how Raccode works directly with Mooshak, making requests to obtain all resources needed for the good functionality of the plugin.

4.2   Design

In terms of design, Raccode uses a perspective to present the many tools needed to solve a problem. The default design aspect of Raccode includes a text editor, a list of problems, a package explorer, the console, the rankings list, and a Q&A (questions and answers) window. The toolbar of the perspective adds two buttons, one to get a new problem and another to submit a solution. As previously described, the new problem button opens a wizard that asks for user authentication, if it is not authenticated yet. Once logged in, and continuing in the same wizard, all problems in the contest become available and the user can choose which of them wants to solve, and in which language.

Furthermore, there is a page in Eclipse preferences to configure the connection to the server, in which Mooshak is added by default. Figure 2 presents the distribution of the several views, which were strategically positioned to fit the common Eclipse perspectives. Three of these windows are recognizable by any Eclipse user: package explorer, text editor, and terminal (console). The important here is to present the other three. The `Problem` view shows the problem statement, with the description, and some examples of input and output. The `Ranking` view allows the participant to have a notion of his progress (which problems solved with success, how many submissions were made, which problems remain to be resolved, etc). Furthermore, it is also useful to know the progress of the other participants, since it is possible to make competitions using Raccode. Each submission made by a participant is only accepted to run on the evaluation machine if the program has some difference from the previous program. After compiled and tested, a feedback message is returned, which can be `Accepted`, `Runtime Error` or `Compile Time Error`, for example.

When a participant of a contest has some doubts about a problem, he can submit a question through the `Q&A` view for teachers/judges to clarify their issues. If Mooshak is in the learning mode, other students can also answer the questions from their peers also.

Also, an a how-to manual about the work environment is provided, with information about submission of programs, the meaning of the feedback messages, troubleshooting, among others.
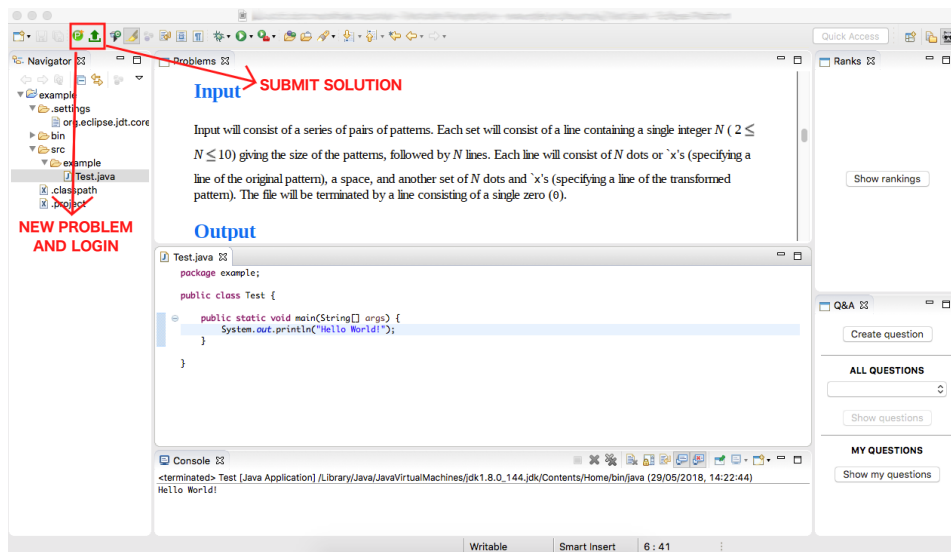
Figure 2 Raccode perspective: distribution of the components (open to changes).

5 Conclusions

At the moment, Raccode is a work in progress. The connection between the REST API of Mooshak 2.0 and Eclipse is almost completed, only missing the submission and automatic creation of project parts. The design of the application is also complete with all views and wizards created, as well as the integration of Eclipse tools in the perspective.

Raccode has some issues with language verification. For instance, when the user wants to solve a problem in C++, it can not verify if the user has the required libraries installed to compile it.

Regarding improvements, the goal of Raccode is to integrate with other IDEs, since the market for IDEs is quite large. Another future improvement concerns the automatic installation of the required environment for compiling programs exactly as in Mooshak.

Raccode will be tested in an open environment with students from the Department of Computer Science of the Faculty of Sciences of the University of Porto (DCC-FCUP). This experiment aims to compare Enki with Raccode environment, in a small open course with a series of problems. The feedback will be taken through an online questionnaire based on the Nielsen's model [7], in Google Forms.

References

1   Helder Patrick de Pina Correia. *Avaliação de diagramas no Mooshak 2.0*. Master's thesis, Universidade do Porto, 2017.
2   Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.
3   Andreas Kornstadt and Eugen Reiswich. Composing systems with Eclipse rich client platform plug-ins. *IEEE Software*, 27(6):78–81, 2010.
4   José Paulo Leal and Fernando Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.
5   Jeff McAffer, Jean-Michel Lemieux, and Chris Aniszczyk. *Eclipse rich client platform*. Addison-Wesley, 2010.

6   Wassim Melhem and Dejan Glozic. PDE does plug-ins. Technical report, IBM Canada Ltd., 2003.

7   Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213, 1993.

8   José Carlos Paiva, José Paulo Leal, and Ricardo Alexandre Queirós. Enki: A pedagogical services aggregator for learning programming languages. In *Conference on Innovation and Technology in Computer Science Education*, pages 332–337, 2016.

9   Matthew Scarpino, Stephen Holder, Stanford Ng, and Laurent Mihalkovic. *SWT/JFace in action*. Manning, 2005.

10 Lars Vogel. *Contributing to the Eclipse IDE Project: Principles, Plug-ins and Gerrit Code Review*. Lars Vogel, 2015.

eOS: The Exercise Operating System

Rui Mendes

Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
azuki@di.uminho.pt
 https://orcid.org/0000-0002-5321-6863

José João Almeida

Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
jj@di.uminho.pt
 https://orcid.org/0000-0002-0722-2031

Abstract

We present an architecture for a system for creating, adapting and evaluating programming exercises for students. The system is capable of generating exercise skeletons, automatically creating inputs and outputs, provide a way of creating a large number of exercises programmatically and allowing students to solve them while giving them feedback. Furthermore, it allows the creation of special comparators that can check whether the output of a given submission is equivalent to the expected one or simply check whether the above mentioned output corresponds to a correct solution.

1 Introduction

Evaluating students' performance in programming involves creating a large number of programming exercises and tools for estimating how well they solve them. The task of creating programming assignments forces teachers to devise these problems and to write them in a systematic fashion, not only concerning the task descriptions but also how they are evaluated. The usual methodology involves creating several scenarios that cover all the cases and check if the submissions solve them correctly. Thus, the team needs to create the inputs and corresponding outputs covering those cases. Furthermore, in many cases the formulation is rendered more difficult because there are several correct answers and this usually involves further complicating the problem by establishing a specific ordering (e.g., we want the first solution in the lexicographic order) or artificially simplifying the problem in order to get a deterministic answer (e.g., asking for the length of the minimum path length in a graph instead of one of the paths).

The goal of `eOS` is to help in this task. `eOS` will help create assignments by automatically generating program inputs and even getting the outputs by automatically generating them from the inputs by means of a solution. Furthermore, it is capable of handling comparators for increasing the ability of ascertaining whether a given solution is correct.

What sets `eOS` apart from other systems like CodeBoard [4], Stepik [15] or Mooshak [12] is the fact that it provides ways to programmatically create exercises by using scripting tools instead of using a web interface that, while user friendly, is time consuming when it is necessary to create many, often similar, problems. The second advantage is the fact that

it is easy to use special comparators thus allowing the creation of programming exercises without having to artificially modify the system in order to get a single, deterministic answer to a given input. The third advantage is being able to create problems that involve creating one or more functions in a given, larger program.

2   On the automatic evaluation of programming assignments

It is not easy to write programs that are capable of automatically evaluating code. This is due to the fact that the automatic analysis of code is difficult. It is extremely hard to write a system that is capable of understanding what a piece of code does without running it and even to know if it will terminate. The usual approach for the automatic evaluation of programs is somewhat similar to unit testing [10]: it sets up a given environment and runs the program and analyzes its output. Most existing systems are either language dependent or often prefer to have the program read inputs from the *standard input* and write its result to the *standard output*. The authors are especially interested in systems that work in a wide spectrum of languages and thus favor the latter approach.

At the Department, the teaching staff uses automated evaluation in several courses and languages: we have been using such systems with `C`, `Python`, `Perl` and courses like introduction to programming, algorithms and complexity and bioinformatics. We also have courses that use other programming languages (e.g., `Java`) and subjects (e.g., machine learning, evolutionary computation) but don't use such systems mainly because the task of creating programming assignments is quite cumbersome and current systems are not capable of dealing with non determinism or the stochastic behavior inherent to such systems.

2.1   Overview of existing systems

We have extensively used Mooshak[12] and also Codeboard [4] or Stepik [15]. Mooshak is a system for managing programming contests on the Web that is used in the Maratona Inter-Universitária de Programação (MIUP) which is a 5 hour Portuguese programming contest that is part of the ACM International Collegiate Programming Contest for teams of three students. When a contest is created, Mooshak allows administrators to create problems. Creating a problem involves using a web interface with the mouse and keyboard, selecting things like the description, timeout, problem letter (from A to Z), if there are static or dynamic correctors and other details. Creating tests involves, again using the web interface, to create several tests where one selects what is the input, the output, context and number of points. One of the authors was involved in preparing the last MIUP contest and the task of setting up the system is somewhat daunting because the contest involved nine problems each with more than 10 test cases. It is possible to create special correctors (somewhat similar to what is presented in section 3.5) but the functionality is not well documented and it is almost never used. Thus, one is limited to check whether the expected and obtained outputs are exactly equal, to the point of there being a specific error message for the fact that both match if one removed all whitespace. Its limitations also involve that using floating point is avoided to the maximum and the authors have had several problems in the past when creating assignments that use floating point, to the point where these exercises are either simply not used or involve much more information about truncating errors and rounding.

Stepik was thought for creating courses and allows the creation of programming assignments. The programming assignments are created in Python and involve creating functions that generate program inputs, outputs and comparators between the expected and obtained outputs like our system. However, it is hard to manage the assignments, including moving them and each assignment must be created or edited using a web interface involving using the mouse and keyboard.

Codeboard can perform automatic evaluation by creating a project that outputs a special string as the last line of output. This involves creating the project using an IDE in the web interface and, for each assignment and language, creating a part of the code that grades the system. It is also possible to use of Java-JUnit, Haskell-HSpec or Python-UnitTest to help evaluate assignments. However, assignments are still created one at a time using a web interface and furthermore, they seem to be language specific.

2.2 Creating programming assignments

Creating programming assignments often involves:
1. Write the description of the assignment;
2. Create the inputs that will be passed to the programs submitted by the users in order to estimate their correctness;
3. Create the expected outputs;
4. Describe how close the outputs of the submission match the expected ones.

Most of these steps often involve repeatable effort. Creating the inputs often depends on the type of problem. For instance, problems over sequences or lists involve creating them according to a given rule (e.g., generate integer lists with random elements over a given interval) while most problems concerning graphs involve creating a graph with some characteristics (e.g., a geometric graph [13]). In order to create a candidate solution to a problem one needs to create some sort of algorithmic pipeline.

For instance, when creating a problem that involves path-finding, it is necessary to generate graphs, compute the distances between nodes, apply a shortest path algorithm (e.g., Dijkstra [3] or Floyd Warshall [5]) and present a path. Then, it is necessary to compare the path produced by the submission with the expected outputs in order to estimate the correctness of the solution.

This programming task's difficulty could even be tweaked by either providing the edges of the graph along with their weights or by providing the geometric coordinates of the vertexes or even some more indirect way of representing them.

The task of ascertaining the correctness of the submissions also involves repetition. If the expected output is for instance, a set of values, it is necessary to verify whether the user's submission provides the correct output but simply using a different ordering or, in the case of a graph path, if it is a valid one of the correct length.

Even writing the problem description is not without its fair share of repetition. In fact, the description of the program inputs and how the program should print its output could be reused for similar problems.

Often, it is necessary to create several problems using the same data structure. This means that it is usually possible to create an input generator for that data structure and use it in several problems. If one is able to combine this with a library that solves all the necessary tasks, it should be easy to create several programming tasks that will be able to correctly evaluate the students' ability to solve them.

3 Framework

The aim of **eOS** is to provide a framework, written in Python, to help the teaching staff with the task of creating problem assignments. It uses a command called **eos** with the following sub-commands (cf. Fig. 1 for the **eOS** process flow):

register This command allows the registration of a new problem;

evaluate This command takes a problem and a submission and evaluates the submission;
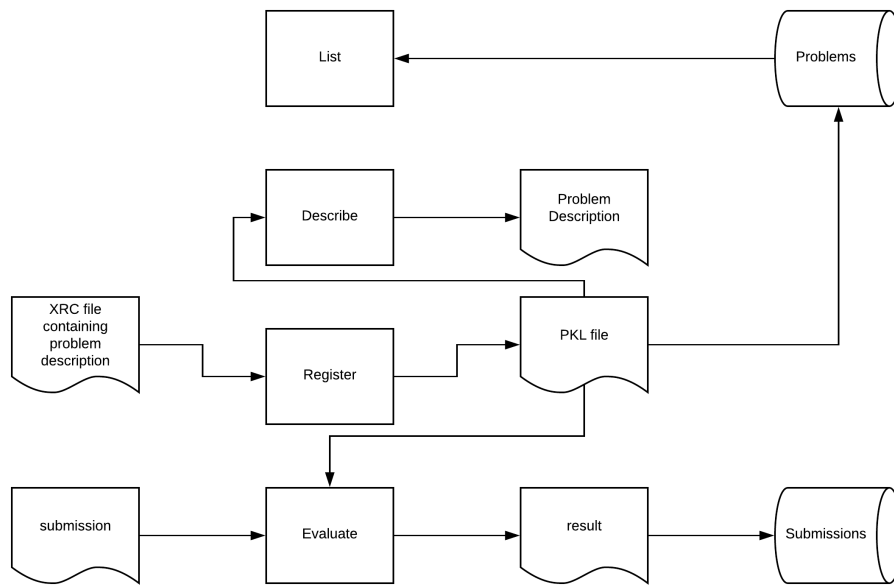
■ **Figure 1** Process flow of eOS.

list This command lists all problems;

describe This command describes a problem;

language This command allows the definition of a new programming language.

3.1 Registering problems

In order to register problems, we conceived a very simple Domain Specific Language (DSL) [11]. It allows users to describe all the aspects of the problem including:

import In order to import Python modules;

input output The input or output;

description A problem description;

parse_input parse_output A function that parses the input or output, it takes a string and returns a list of strings;

solution A function that solves the problem, it takes a string corresponding to one of the inputs and returns a string that is the corresponding output;

template In the case where the problem only asks for a part of the code;

lang The language of the problem;

ntests The number of tests to show to the user (0 by default);

timeout The number of CPU seconds allowed for the program to run on each test (1 by default);

comparator A function that takes the input, expected output and obtained one and returns `True` or `False`;

source_invariant A function that takes the language used by the submission and the program source and returns `None` in case it can be accepted and a string containing an error message in case the program cannot be accepted.

Each commands starts with a `#` (e.g., the command is `#import`) and accepts one or more lines. These commands can be supplied with an exclamation point suffix in order to evaluate the argument of the command assuming it is Python code. In case the arguments supplied to a command consist of a a single word (i.e., they have no whitespace), the systems searches for a Python function with that name.

3.2 Describing inputs and outputs

In order to describe inputs, one uses the `#input` keyword. This involves creating a list of strings where each string corresponds to one of the inputs. This keyword can accept a function, given by its name, that produces the list. In this case, if the function has a docstring [2], it will be automatically added to the problem description.

By default, the input is several lines where each line corresponds to one input. In case something else is necessary, one can use the keyword `#parseinput` for supplying a function that performs the necessary parsing and produces the list of strings. It is also possible to supply the list of strings directly by appending an exclamation point to the input command. As an example, the following lines supply the same input:

```
#input
there is no place like home.
hello there!
#input! ['there is no place like home.\n', 'hello there!\n']
```

One can specify the output in exactly the same way or by supplying a solution by using the `#solution` keyword. This is a Python function that can be given by name and that takes a string corresponding to one input and produces a string corresponding to one output. If the solution is specified and the function has a docstring, it will be automatically used in order to document how the output is specified. No documentation of the input or output is performed if a template is used (since in this case the user does not have to worry about it).

3.3 A simple example

Let us assume that we aim to create a problem where the user has to read a sequence of integers and has to compute the longest increasing subsequence. We would need to write a text file, e.g, `lis01.xrc` that could have the following:

```
#description
Write a function called lis that takes a list of numbers and returns the
size of the longest increasing subsequence of non-consecutive integers
found.
#input
3 1 2 4
7 2 1 3 2 3 7 2 4
1 1 2 2 3 3 3 4 4 4 5 5 5 6
#output
3
4
14
#lang python
#template
[[function]]
```

```
lst = list(map(int, input().split()))
print(lis(lst))
#ntests 1
```

We define a template, that the submission must be in Python and that only the first of the three tests is shown to the user as feedback. It is also possible to supply the input by means of a function that returns a list of strings where each string is a test case and the output as a function that yields a list of strings. Templates can be used for other languages as long as they are defined (cf. section 3.8).

Instead of the output, we can supply a solution that is a function that computes the output given the input. In this case, we could have the following in file `lis02.xrc`:

```
#import example
#description
Write a program that reads a line containing a sequence of numbers
separated by spaces and prints the size of the longest increasing
subsequence of non-consecutive integers found.
#input get_input
#solution solve_lis
#ntests 2
#timeout 2
```

The module `example.py` contains the functions `get_input` and `solve_lis`. The function `get_input` doesn't take any arguments and returns a list of strings while the function `solve_lis` takes a string, which is an input and returns a string that is the output. If `get_input` has a docstring, it will by automatically appended to the problem description. The output description can also be taken from the docstring of the function `solution`. We also specify that the timeout is 2 seconds (instead of the 1 second default) and that the first two tests are supplied to the user as feedback. This problem asks the user to write the whole program, including reading the input and writing the output and accepts solutions in any language.

If we want to register the problem `lis02.xrc` given above and evaluate a solution in a file called `sub01.py` we would write:

```
$ eos register lis02
Write a program that reads a line containing a sequence of numbers
separated by spaces and prints the size of the longest increasing
subsequence of non-consecutive integers found.
The input consists on a single line containing several integers
separated by spaces.
The output consists of a single integer representing the length
of the longest increasing subsequence of non-consecutive integers
found.
Input 1:
3 1 2 4
Output 1:
3
Input 2:
7 2 1 3 2 3 7 2 4
Output 2:
```

```
4
$ eos evaluate lis02 sub01.py
Ok!
```

Notice that since the functions `get_input` and `solve_lis` have docstrings, their documentation is added at the end of the problem description. If we evaluate a submission that doesn't pass all tests, the output will indicate a feedback, if available, how many tests were run, how many were correct, how many had errors and how many timed out.

```
$ eos evaluate lis02 sub02.py
correct: 1
error: Timeout
errors: 0
timeouts: 10
total: 11
```

The file `sub02.py` corresponds to a naive implementation and, as such, it can only solve 1 of the 11 problems in the CPU time given. We realize that our program wasn't able to solve 10 problems due to timeout.

3.4   Parsing the input or output

In case we wanted to create a task for counting the number of lines in the input, we would want a different way of specifying input since in this case, the input would be several lines. For this, we could use a function called `get_paragraphs` defined in `parse_inputs.py` that splits the text on blank lines. Notice that, in this case, no tests are shown to the user in case of failure.

```
#import parse_inputs
#description
Write a program that counts the number of lines in the input.
#parse_input get_paragraphs
#input
d

e
f
#output
1
2
```

3.5   An example with special comparators

In many cases, it is more interesting to use a special comparator because there are many ways of supplying the same solution. For instance, if we ask for a path between two vertexes in a graph, there can be several paths between these nodes even if we are only interested in the shortest one. In this case, we have to supply a `comparator`.

```
#import graphs
#description Write a program that reads a graph and two nodes and writes
the shortest path between the two nodes.
```

```
196
1 2 249
1 3 421
1 4 426
2 3 172
2 4 193
2 83 184
...
99 71 125
99 72 101
100 18 123
100 20 110
34 2
```
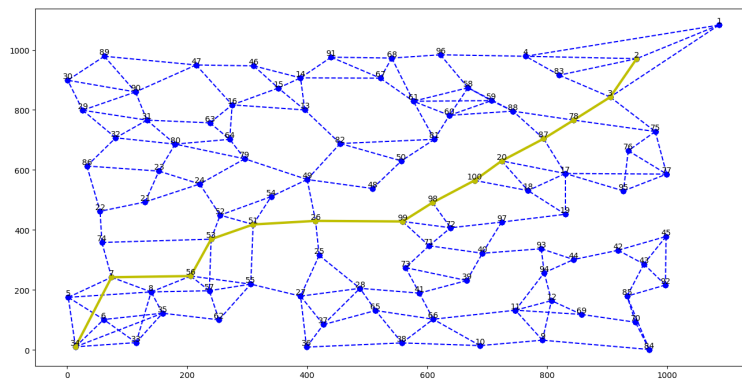
Figure 2 Example of a generated graph with 100 vertexes and a path of minimum length from 34 to 2 and the corresponding input. The last line of the input corresponds to the origin and destination vertexes (34 and 2 respectively). There is another path of the same length that starts with [34, 33, 8, 56]. The figure was automatically created by the generator.

```
#input! generate_path_problems(10, 100)
#solution get_shortest_path
#comparator same_path_length
```

In this case, the function `generate_path_problems` was defined in `graphs.py` and generates 10 graphs with 100 nodes each along with the figures depicting the generated graphs (cf. Fig 2). The graph appears in the input with one line with an integer for the number of edges and one line for each edge with two node ids and the corresponding weight separated by spaces and a final line with the origin and destination nodes separated by a space. Notice that in this case we used `input!` because we have to evaluate the input as it is not simply the name of a function. The function `same_path_length` takes three arguments: the input, the expected output and the obtained one and should return `True` if both paths are from the same two vertexes and have the same length or `False` otherwise.

Another advantage of using comparators is that they can verify whether a given output is correct. As a rather contrived example, we could ask for a sorting algorithm and simply use the comparator to check whether the output produced by the submitted program contained all the elements in the input and they were ordered. Thus, we could simply supply the input and the comparator even though we didn't have a function that produced the output. The comparator can also be used to check if a given heuristic produces a solution of an acceptable quality (e.g., using the A* algorithm [9]).

3.6 Source invariants

In some cases, we may want to design a problem where the user cannot use a given function or library because we want to evaluate a given algorithm. For instance, let us suppose that we want to evaluate submissions that implement hash tables. In this case, we could create a function in Python similar to this one:

```
import re
def check_for_hash(lang, src):
RE = {'Java': r'java.util.(Hashtable|HashMap)',
```

```
       'C'   : r'#include\s*<search.h>'            }
    for line in src.splitlines():
       if re.search(RE[lang], line):
            return 'You may not use a library that implements hash tables'
```

and use it in the definition of a problem by adding `#source_invariant check_for_hash`.

3.7 Batch creation of exercises

The main advantage of `eOS` is to provide a programmatical way of creating several exercises. We will illustrate this concept by creating several problems that compute statistics over lists of integers.

```
problems = 'minimum maximum mean median variance'
text = """
#import my_list
#import fun_list
#description
Write a program that reads a line containing several integers separated
by whitespace and outputs their {name}
#input! gen_lists(range(1, 11))
#solution sol_{name}
"""
fun = """
def sol_{name}(inp):
    lst = list(map(int, inp.split()))
    return str({module}.{name}(lst))
"""
with open('fun_list.py', 'w') as FL:
    print('import my_list', file = FL)
    for num, prob in enumerate(problems.split()):
        with open("prob{:02}.xrc".format(num + 1), "w") as F:
            print(text.format(name = prob), file = F)
        print(fun.format(module = 'my_list', name = prob), file = FL)
```

`gen_lists` is a function that generates a random list of the given size. Thus, the specified input creates one list of each size ranging from 1 to 10. In this example, we first create 5 functions in a file called `my_list.py` (minimum, maximum, mean, median and variance) and this Python script creates a file called `fun_list.py` with a version of these functions that includes parsing the input and outputing the result as a string. This is done using the `fun` template that is used to create functions with the prefix `sol_`. This script also creates 5 files with names `prob01.xrc`, ..., `prob05.xrc` using the template `text` corresponding to these 5 problems.

By executing this script and subsequently calling:

```
for prob in prob0?.xrc; do eos register $prob; done
```

on the command prompt, we register the 5 problems into the system and can then use them. This is a proof of concept that will soon be a part of the system to facilitate its reuse. Nevertheless, this strategy can easily be adapted to generate more exercises by adapting this script to other needs.

3.8   Defining new languages

`eOS` is able to evaluate problems defined in any language as long as one knows how to take a solution and *compile* it in order to create an executable and how to *run* it. The keyword `eos language` allows users to define new languages.

In order to define a new language, the user needs to define the following aspects:

name The language name;

compile The Unix shell command that compiles the program and creates the executable using `file` for the filename of the submission (this field may be omitted in case of an interpreted language);

run The Unix shell command that runs the program;

extension The extension or extensions for this language.

For instance, in order to define `C++`, one could create the following file called `lang_cpp`:

```
#name C++
#compile g++ -std=c++11 -Wall -Wextra -Werror [[file]]
#run ./a.out
#extension cc cpp cxx
```

And subsequently run the command `eos language lang_cpp`. This command only needs to be run once per system. As a subsequent example, let us imagine the scenario where we want to create exercises for a course where we want to evaluate the use of `flex` and `bison`. We can create a custom language `FlexBison` that expects a file, terminated by the extension `flbi`, archived with `tar` and compressed with `bzip2` containing two files, one named `lex.l` and another named `gram.y`:

```
#name FlexBison
#compile
tar xjf [[file]]
bison -d gram.y
flex lex.l
g++ gram.tab.c lex.yy.c -lfl -o parser
#run ./parser
#extension flbi
```

4   Discussion

Existing systems provide user friendly interfaces for creating problems (e.g. [15, 4, 12]). This is quite useful for creating exercises since it allows users to create them using a user friendly interface. However, there are several advantages to having a programmatic way of generating exercises. The first and foremost is the creation of a batch procedure for generating many exercises.

Thus, it is possible to use a library for generating data structures (e.g., trees or graphs) and create a large number of exercises that involve creating, updating or traversing the data structure or using frequently taught algorithms over that structure (e.g., path between two vertexes, checking whether it is a direct acyclic graph or a connected graph).

When devising problems, it is important to reserve inputs that test all functionalities of the task besides the ones shown to the users in order to prevent them from simply creating a program that prints the correct output given the input. As we intend to use this system both for helping students learn programming as to evaluate their success, it is possible to show all inputs to the students but we must caution the teaching staff about the obvious drawback involved.

It is very important to create inputs that correctly test the task being evaluated including all normal inputs and hedge cases (e.g., in the case of sorting a list, it is necessary to have a case using the empty list, a list with one element, lists with several elements, lists with repeated elements and lists with a very large number of elements). Inexperienced people can be tempted to only supply a few well behaved cases and later realize that very bad solutions were accepted. As a degenerate example, if all inputs for our sorting problem use lists of three integers, it is possible that a system using a few conditional statements and no loops is accepted. There are many more considerations when evaluating problems by supplying inputs and outputs, mainly it is necessary to ensure whether there are simpler algorithms than the ones we intend to evaluate that can solve the problem [6].

The main advantages of `eOS` are the following:

- It was thought with the Unix philosophy in mind;
- It is easy to create scripts for the batch generation of exercises;
- It allows importing Python modules to help create exercises;
- Most of the keywords accept a function that generates the needed value;
- It provides feedback to the user when it fails, and the problem setter can configure how many of the inputs are shown to the user;
- It automatically includes docstrings of functions used to generate the input and output to the program description thus sparing the problem setter to have to write them;
- It provides facilities for evaluating a program or several functions that are included in a larger program;
- It can evaluate code in almost any language and each problem can potentially receive submissions in several languages;
- It is quite easy to create special comparators for specific similarity measures or for evaluating heuristics (including non deterministic and stochastic ones);
- It logs all submissions to the system.

The main shortcomings of `eOS` are the following:

- In order to use the system fully, the teaching staff needs to know how to program in Python;
- There is currently no web interface neither for the teaching staff to create problems nor for students to use the system;
- As in most similar systems, one must test all hedge cases of a problem;
- There is currently no way to assessing the complexity of a program, its style or if plagiarism;
- It needs to be fully tested in real world conditions and it needs stress tests to ascertain its security to attacks.

5   Conclusions

The framework presented in this work helps the teaching staff to quickly generate several programming assignments by introducing ways to automate this task. This automation is due to the fact that inputs and outputs can be supplied in several ways including by using generators, specific parsers and solutions that automatically generate outputs. The fact that it is possible to import Python modules makes it possible to easily extend the system to attend specific needs. Another advantage of this system is being capable of using special correctors to accept solutions based on user supplied metrics of equivalence or proximity.

Currently, we have implemented a mail filter that receives submissions by email from university accounts. This allows us to use the system in our courses and provides an easy

authentication of their submissions. We will evaluate this system in practice in the next semester in several courses in order to evaluate its performance. Some of the courses involved will use several programming languages and technologies (e.g., C, Python, Flex, Bison, BioPython, Machine Learning algorithms) and can involve up to 150 students.

Currently, `eOS` addresses some security concerns about compiling and running insecure code (e.g., forks, memory and disk limits) but more work is surely needed to mitigate all the possible security issues [7]. In the future, we intend to create a Web portal for students to submit their solutions by using this system and to specify a programmatic layer on top of the existing one to further automate the generation of exercises (cf. section 3.7). We aim to accomplish this by supplying several ways of generating inputs, outputs, solutions and special comparators by using several forms of functional composition. We intend to perform this by adding functionalities for documenting input generators and the ability of concatenating several generators in order to both create the input and automatically describe the input in the problem description.

We also intend to use this system for gathering statistics concerning the submission process that will allow us to use machine learning techniques to analyze the results and design intelligent tutors [1, 8, 16]. We also wish to incorporate the ability to check for code plagiarism by automating the task of using systems like MOSS [14].

References

1 Charu C. Aggarwal and Jiawei Han. *Frequent pattern mining*. Springer, 2014.

2 Guido van Rossum David Goodger. PEP 257 – Docstring conventions. Documentation, Python Software Foundation, 2001. URL: `https://www.python.org/dev/peps/pep-0257/`.

3 Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

4 Christian Estler and Martin Nordio. Codeboard: A web-based ide to teach programming in the classroom, 2018. URL: `https://codeboard.io/`.

5 Robert W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

6 Michal Forišek. On the suitability of programming tasks for automated evaluation. *Informatics in education*, 5(1):63–76, 2006.

7 Michal Forišek. Security of programming contest systems. *Information Technologies at School*, pages 553–563, 2006.

8 Karam Gouda, Mosab Hassaan, and Mohammed J Zaki. Prism: An effective approach for frequent sequence mining via prime-block encoding. *Journal of Computer and System Sciences*, 76(1):88–102, 2010.

9 Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

10 Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

11 Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: a systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

12 José Paulo Leal and Fernando Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003.

13 Mathew Penrose. *Random geometric graphs*, volume 5. Oxford university press, 2003.

14 Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.

15   Stepic team. Stepic.org: Cloud-based digital learning environment for computer science. ,
          `https://blog.stepik.org/`. URL: `https://stepik.org/`.
16   Mohammed J. Zaki. Spade: An efficient algorithm for mining frequent sequences. *Machine
          learning*, 42(1–2):31–60, 2001.

Construction of a Pushdown Automaton Accepting a Postfix Notation of a Tree Language Given by a Regular Tree Expression

Tomáš Pecka[1]

Department of Theoretical Computer Science, Faculty of Information Technology
Czech Technical University in Prague
Tomas.Pecka@fit.cvut.cz

Jan Trávníček

Department of Theoretical Computer Science, Faculty of Information Technology
Czech Technical University in Prague
Jan.Travnicek@fit.cvut.cz

Radomír Polách

Department of Theoretical Computer Science, Faculty of Information Technology
Czech Technical University in Prague
Radomir.Polach@fit.cvut.cz

Jan Janoušek

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

Abstract

Regular tree expressions are a formalism for describing regular tree languages, which can be accepted by a finite tree automaton as a standard model of computation. It was proved that the class of regular tree languages is a proper subclass of tree languages whose linear notations can be accepted by deterministic string pushdown automata. In this paper, we present a new algorithm for transforming regular tree expressions to equivalent real-time height-deterministic pushdown automata that accept the trees in their postfix notation.

1 Introduction

The theories of formal string languages and formal tree languages are important parts of computer science. Strings and trees are fundamental data structures. Tree languages processing has become very popular in the recent years. For example, we can find practical usages in the area of processing markup languages (like XML) or abstract syntax trees. Traditionally, problems on trees are solved using various kinds of tree automata [5]. However, trees can also be represented by strings, for instance in their prefix or postfix notation obtained by preorder or postorder traversal of the tree, respectively. It was proved by

Janoušek and Melichar [9] that the class of regular tree languages is a proper subclass of tree languages whose linear notations can be accepted by deterministic pushdown automata (PDAs). Thus, the standard (string) PDA is another suitable model of computation for processing regular tree languages in a linear notation. For example, algorithms processing XML with the use of PDAs have been investigated [10, 17].

Regular tree expressions (RTEs) are a natural formalism for the description of regular tree languages [5]. They are analogous to regular (string) expressions. It is well known that regular (string) expressions describe regular languages and can be converted to finite automata. In the case of trees, RTEs can be converted to corresponding finite tree automata.

Finite automata and regular (string) expressions are well-studied [8, 16]. A string language membership problem is a decision problem. Given a regular (string) expression E and a string w, decide whether w is in the language described by the regular (string) expression E. This problem can be decided by converting the expression to an equivalent finite automaton and running the automaton on the input word w.

Many algorithms deal with a problem of converting regular (string) expressions to finite automata in the string domain. Three algorithms by Brzozowski [4], Thompson [18] and Glushkov [7] (also known as a position automaton) are the basic ones. Antimirov's partial derivatives method [3] (which can be seen as a non-deterministic extension of Brzozowski's algorithm) must be also mentioned. Conversions by Glushkov's and Antimirov's can be done in polynomial time w.r.t. the number of occurrences of symbols in the regular expression.

The language membership problem for trees and RTEs is analogous: Given a regular tree expression E and a tree t, decide whether t is in the language described by the regular tree expression E. As in the string case, one can create a finite tree automaton (or a PDA) equivalent to the RTE E and let the automaton run on (linearised) tree t.

Algorithms for the conversion of RTEs to finite tree automata are inspired by the mentioned algorithms from the string domain. Antimirov's and Glushkov's algorithms were adapted to regular tree expressions by Kuske and Meinecke [11] and also later by Laugerotte et al. [12]. The finite tree automaton is constructed in polynomial time w.r.t. the size of the RTE in both adaptations. Thompson's algorithm was an inspiration for Polách [14], where RTEs are converted to PDAs.

This paper presents a new approach for the conversion of RTEs to PDAs. The presented algorithm was inspired by the Glushkov's algorithm [7] for regular (string) expressions. To create the equivalent PDA, the RTE is analysed similarly to Glushkov's algorithm. Resulting PDA accepting linearised trees described by the RTE is constructed in quadratic time w.r.t. the size of the RTE. The constructed automaton is a real-time height-deterministic PDA and therefore it can be always determinised [15].

The paper is organised as follows: Basic definitions are given in Section 2. The conversion algorithm producing the PDA is presented in Section 3. Section 4 discusses complexity improvements to the algorithm and to the size of the constructed PDA. Finally, the results are summarised in the conclusion.

2   Basic Definitions

2.1   Trees

A labelled, ordered and ranked tree over a ranked alphabet $\mathcal{A}$ can be defined based on the concepts from graph theory [1].

A *ranked alphabet* $\mathcal{A}$ is a finite nonempty set of symbols. Each symbol a is assigned with a non-negative integer *arity* denoted by arity(a). $\mathcal{A}_n$ denotes the set of symbols from $\mathcal{A}$ with arity n. The set $\mathcal{A}_0$ is nonempty. Notation $a2$ denotes symbol a with arity$(a) = 2$.
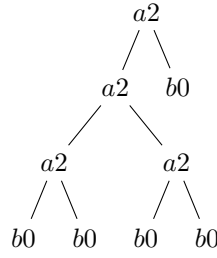
Figure 1 A directed, rooted, labelled, ranked and ordered tree over $\mathcal{A} = \{a2, b0\}$.

A *directed ordered graph* G is a pair (N, R) where N is a set of nodes and R is a set of ordered lists of edges. Elements from R are in the form $((f, g_1), (f, g_2), \ldots, (f, g_n))$, where $f, g_1, g_2, \ldots, g_n \in N$, $n \geq 0$. Such element indicates that there are n edges leaving f with the first edge entering node g_1, the second entering node g_2, and so forth. A sequence of nodes $(f_0, f_1, \ldots, f_n)$, $n \geq 1$ is a *path* of length n from node f_0 to node f_n if there is an edge from f_i to f_{i+1} for each $0 \leq i < n$. A *cycle* is a path where $f_0 = f_n$.

An *in-degree* of a node $f \in N$ is the number of distinct pairs (g, f), $g \in N$ in elements of R. An *out-degree* of $f \in N$ is the number of distinct pairs (f, g), $g \in N$ in elements of R. A node with the out-degree 0 is a *leaf*.

An ordered *directed acyclic graph (DAG)* is an ordered directed graph with no cycle. A *rooted DAG* is a DAG with a special node $r \in N$ called the *root*. The in-degree of r is 0, in-degree of every other node is 1 and there is just one path from the root r to every $f \in N$, $f \neq r$. A *labelled ranked DAG* is a DAG where every node is labelled by a symbol $a \in \mathcal{A}$ and the out-degree of a node $a \in \mathcal{A}$ equals to arity(a). A *directed, ordered, rooted, labelled and ranked tree* is rooted, labelled and ranked DAG. All trees in this paper are considered to be directed, ordered, rooted, labelled and ranked.

The *postfix notation* of a tree t denoted by $post(t)$ is defined recursively:
1. $post(t) = root(t)$ if $root(t)$ is also a leaf,
2. $post(t) = post(c_1) \cdot post(c_2) \cdots post(c_n) \cdot root(t)$, c_i are children of $root(t)$.

The postfix notation of a tree language L is defined as $post(L) = \{post(t) : t \in L\}$. A postfix notation of any subtree of t is a substring of $post(t)$. However, not every substring of a postfix notation of a tree is a postfix notation of its subtree [6].

▶ **Example 1.** Let t from Figure 1 be a directed, rooted, labelled, ranked and ordered tree with labels from ranked alphabet $\mathcal{A} = \{a2, b0\}$. The root of t is a node $a2$ with an ordered 2-tuple of children $(a2, b0)$. Postfix notation of t is $post(t) = b0\,b0\,a2\,b0\,b0\,a2\,a2\,b0\,a2$.

2.2 Regular Tree Expressions

Regular tree expressions (RTEs) are defined using a substitution operation as in [5]. The definition of the RTE is similar to the definition of the regular (string) expression.

RTEs are defined over two alphabets, $\mathcal{F}$ and $\mathcal{K}$. $\mathcal{F}$ is a ranked alphabet of symbols. $\mathcal{K}$ is a set of constants (symbols with arity 0), $\mathcal{K} = \{\Box_1, \Box_2, \ldots, \Box_n\}$, $n \geq 0$, $\mathcal{F} \cap \mathcal{K} = \emptyset$. This alphabet is used to indicate the position where substitution operations take place.

Firstly, the substitution, i.e. replacing occurrences of $\Box_i$ by trees from a tree language L_j, is defined. Let $\mathcal{K} = \{\Box_1, \ldots, \Box_n\}$ and t be a tree over $\mathcal{F} \cup \mathcal{K}$, and let $L_1, \ldots, L_n$ be tree languages. Then the *tree substitution* of $\Box_1, \ldots, \Box_n$ by $L_1, \ldots, L_n$ in t denoted by $t\{\Box_1 \leftarrow L_1, \ldots, \Box_n \leftarrow L_n\}$ is the tree language defined by the following identities:
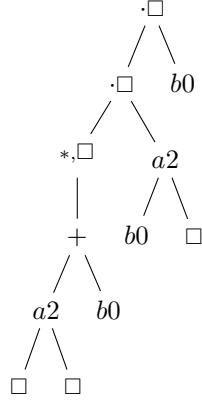
(a) A sample regular tree expression (RTE).
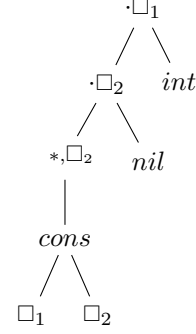
(b) A RTE denoting the language of integer lists in LISP.

🟨 **Figure 2** Examples of RTEs.

- $\square_i\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = L_i$, for $i = 1, \ldots, n$,
- $a\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = \{a\}, \forall a \in \mathcal{F} \cup \mathcal{K}$ with $\text{arity}(a) = 0$ and $a \neq \square_1, \ldots, a \neq \square_n$,
- $f(s_1, \ldots, s_n)\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = \{f(t_1, \ldots t_n)|t_i \in s_i\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}\}$.

The tree substitution can be generalized to languages: $L\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\} = \bigcup_{t \in L} t\{\square_1 \leftarrow L_1, \ldots, \square_n \leftarrow L_n\}$.

The operation *alternation* of L_1 and L_2 is denoted by $L_1 + L_2$. The result is a set of trees obtained by the union of regular tree languages L_1 and L_2, i.e. $L_1 \cup L_2$.

The operation *concatenation* of L_2 to L_1 through $\square$, denoted by $L_1 \cdot_\square L_2$, is the set of trees obtained by substituting the occurrence of $\square$ in trees of L_1 by trees of L_2, i.e. $\bigcup_{t \in L_1} t\{\square \leftarrow L_2\}$.

Given a tree language L over $\mathcal{F} \cup \mathcal{K}$ and $\square \in \mathcal{K}$, the sequence $L^{n,\square}$ is defined by the equalities $L^{0,\square} = \{\square\}$ and $L^{n+1,\square} = L \cdot_\square L^{n,\square}$. The operation *closure* is defined as $L^{*,\square} = \bigcup_{n \geq 0} L^{n,\square}$.

The RTE over alphabets $\mathcal{F}$ and $\mathcal{K}$ is defined as follows:

- the empty set ($\emptyset$) and a constant ($a \in \mathcal{F}_0 \cup \mathcal{K}$) are RTEs,
- if $E_1, E_2, \ldots, E_n$ are RTEs and $\square \in \mathcal{K}$, then: $E_1 + E_2$ is a RTE, $E_1 \cdot_\square E_2$ is a RTE, $E_1^{*,\square}$ is a RTE and $a(E_1, \ldots, E_n)$ is a RTE if $a \in F_n$ and $\text{arity}(n) > 0$.

RTE E represents a language denoted by $L(E)$ and defined by the following equalities:

- $L(\emptyset) = \emptyset$,
- $L(a) = \{a\}$ for $a \in F_0 \cup K$,
- $L(f(E_1, \ldots, E_n)) = \{f(s_1, \ldots, s_n) \mid s_1 \in L(E_1), s_2 \in L(E_2), \ldots, s_n \in L(E_n)\}$,

- $L(E_1 + E_2) = L(E_1) \cup L(E_2)$,
- $L(E_1 \cdot_\square E_2) = L(E_1)\{\square \leftarrow L(E_2)\}$,
- $L(E^{*,\square}) = L(E)^{*,\square}$.

The *size* of the RTE E (denoted by $|E|$) is the size of the syntax tree of E. The *number of occurrences* of symbols from $\mathcal{F}$ and $\mathcal{K}$ in the RTE E is denoted by $\|\mathcal{F}_E\|$ and $\|\mathcal{K}_E\|$, respectively.

▶ **Example 2.** Let $\mathcal{F} = \{nil, cons, int\}$ where $\text{arity}(cons) = 2$ and other symbols have arity 0. Let $\mathcal{K} = \{\square_1, \square_2\}$. Then the RTE from Figure 2b denotes the language of lists of integers in LISP: $\{nil, cons(int, nil), cons(int, cons(int, nil)), \ldots\}$

2.3 Pushdown Automata

Notions are used similarly as they are defined in [8].

A *nondeterministic pushdown automaton (PDA)* is a seven-tuple $(Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ where Q is a finite set of states, Σ is a finite set of input symbols (input alphabet), Γ is a finite set of pushdown store symbols (pushdown store alphabet), δ is a mapping from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*$ into a set of finite subsets of $Q \times \Gamma^*$, $q_0 \in Q$ is the initial state, $\perp \in \Gamma$ is the initial pushdown store symbol and $F \subseteq Q$ is a set of final states.

Triplet $(q, w, \alpha) \in Q \times \Sigma^* \times \Gamma^*$ is a *configuration of a PDA*. The initial configuration is $(q_0, w, \perp)$, $w \in \Sigma^*$. Relation $(q, aw, \beta\alpha) \vdash_M (p, w, \beta\gamma) \in (Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Sigma^* \times \Gamma^*)$ is a *transition* of a PDA M if $(p, \gamma) \in \delta(q, a, \alpha)$. $\vdash_M^k$ denotes the k-th power, $\vdash_M^+$ is the transitive closure and $\vdash_M^*$ is the transitive and reflexive closure. In strings representing the pushdown store in this paper, the top of the pushdown store is situated on the right.

A language accepted by PDA M can be defined in two distinct ways. PDA can accept

1. either by final states, then $L(M) = \{w : w \in \Sigma^*, \exists\gamma \in \Gamma^*, \exists f \in F, (q_0, w, \perp) \vdash^* (f, \varepsilon, \gamma)\}$,

2. or by an empty pushdown store, then $L(M) = \{w : w \in \Sigma^*, \exists q \in Q, (q_0, w, \perp) \vdash^* (q, \varepsilon, \varepsilon)\}$ and $F = \emptyset$.

A PDA is *deterministic* if the following conditions hold:

1. $|\delta(q, a, \gamma)| \leq 1, \forall q \in Q, \forall a \in (\Sigma \cup \{\varepsilon\}), \forall\gamma \in \Gamma^*$,

2. If $\delta(q, a, \alpha) \neq \emptyset, \delta(q, a, \beta) \neq \emptyset$ and $\alpha \neq \beta$, then α is not a prefix of β and β is not a prefix of α (i.e., $\alpha\gamma \neq \beta, \alpha \neq \beta\gamma, \gamma \in \Gamma^*$),

3. If $\delta(q, a, \alpha) \neq \emptyset, \delta(q, \varepsilon, \beta) \neq \emptyset$, then α is not a prefix of β and β is not a prefix of α (i.e., $\alpha\gamma \neq \beta, \alpha \neq \beta\gamma, \gamma \in \Gamma^*$).

The class of languages accepted by nondeterministic PDAs is exactly the class of *context-free languages*. Deterministic PDAs accepts *deterministic context-free languages*. This class is a proper subset of context-free languages.

A *height-deterministic PDA* is such PDA which on all of its runs on input word $w \in \Sigma^*$ leads to the same pushdown store height. Height-deterministic PDAs are a generalization of visibly PDAs [13, 15, 2]. A *real-time height-deterministic PDA* is such PDA that is height-deterministic and without ε-transitions. This class of PDAs is determinisable [13, 15].

3   Converting RTE to PDA

In this section, a new method of creating a real-time height-deterministic PDA from RTE is proposed. The constructed PDA accepts postfix ranked linear notation of trees.

3.1   Analysing RTE

To analyse the structure of the expression, the RTE has to be preprocessed similarly to Glushkov's algorithm. Firstly, every occurrence of symbol from $\mathcal{F}$ alphabet of the RTE is subscripted with an unique symbol. Subscripted RTE E is denoted as E'.

Functions First, Follow and Pos are defined to analyse the RTE E'. Function Pos returns a set of occurrences of symbols from $\mathcal{F}$ alphabet of E'. Function First computes a set of symbols that can be a root of a tree described by E'. Function Follow returns tuples of children of a given symbol. Unlike strings, a symbol can be followed by more than a single symbol. The size of the children tuple is defined by the arity of the symbol.

▶ **Definition 3.** Based on the definition of RTEs, the function First is defined recursively:

$$\text{First}(\emptyset) = \emptyset$$

$$\text{First}(a(E_1, E_2, \ldots, E_n)) = \{a\}$$

$$\text{First}(E_1 + E_2) = \text{First}(E_1) \cup \text{First}(E_2)$$

$$\text{First}(E_1 \cdot_\square E_2) = \begin{cases} \text{First}(E_1) & \text{if } \square \notin \text{First}(E_1) \\ (\text{First}(E_1) \setminus \{\square\}) \cup \text{First}(E_2) & \text{if } \square \in \text{First}(E_1) \end{cases}$$

$$\text{First}(E^{*,\square}) = \{\square\} \cup \text{First}(E)$$

▶ **Theorem 4.** *The function* $\text{First}(E')$ *returns the set of symbols that can be the root of any tree described by an RTE* E.

Proof. The proof is done by induction on the structure of the RTE: The basis: If $E = a(E_1, E_2, \ldots, E_n)$, $a \in \mathcal{F} \cup \mathcal{K}$, $n \geq 0$: Only a can be a root. If $E = \emptyset$: There is no root. Now, assume that the theorem holds for any E_1 and E_2. If $E = E_1 + E_2$: This operator unifies two sets of trees. Therefore the roots of trees from $L(E)$ are either from $\text{First}(E_1)$ or $\text{First}(E_2)$. If $E = E_1^{*,\square}$: The roots can be only elements from $\text{First}(E_1)$ or the substitution symbol $\square$. If $E = E_1 \cdot_\square E_2$: Initially, suppose $\square \notin \text{First}(E_1)$. Then the root must be from $\text{First}(E_1)$. If $\square \in \text{First}(E_1)$, then $\square$ gets substituted by the roots of the trees from E_2.   ◀

The function Follow returns a set of tuples of symbols which can be direct descendants (children) of a symbol $a \in \mathcal{F}$. The computation of the function is defined using Algorithm 1. The algorithm recursively traverses the syntax tree of the RTE and maintains a *substitution map*. The map contains roots of all possible trees that can be substituted for each $\square \in \mathcal{K}$. If $\square$ occurs as a child of the symbol a when computing $\text{Follow}(E', a)$, it gets substituted by elements of a substitution map for a given $\square$.

It is possible that $\square_2 \in \mathcal{K}$ is present in the $subMap[\square_1]$ of any node. Then it is required to include the contents of mapping for key $\square_2$ into $\square_1$ set of that node. Also, if $\square \in subMap[\square]$ then $\square$ element can be discarded from the set as it brings no new information.

For the purpose of proving the correctness of the computation, the algorithm can be split in a two pass algorithm. In the first pass, the substitution mapping for each node is computed. In the second pass, the computation of Follow can use the computed mapping.

▶ **Theorem 5.** *Algorithm 1 computes a substitution mapping of every node of the RTE.*

Proof. If the substitution operation takes place (in concatenation and iteration nodes), it alters the substitution map. The changes in substitution mapping come from the definitions of RTEs. Case $E_1 \cdot_\square E_2$: Roots from trees described by E_2 may appear in the place of $\square$ symbols in E_1. Therefore the mapping for the $\square$ symbol in E_1 is replaced. The substitutions for $\square$ symbols in E_2 are determined by the same mapping as in the parent node. Case $E_1^{*,\square}$: Symbol $\square$ is to be replaced by roots of E_1 (this implements the actual iteration) and the iteration is terminated by concatenating a tree from the right operand of the closest substitution or iteration node. In other cases, the existing mapping is simply passed to children as no substitution happens.   ◀

▶ **Theorem 6.** *Function* $\text{Follow}(E', a)$ *(defined by Algorithm 1) correctly returns a set of tuples representing all possible tuples of direct children of a node* a.

Proof. The proof by induction is straightforward with the use of the previous theorem.   ◀

Algorithm 1: Computation of $\text{Follow}(E', a)$ in a single pass.

---

1 Function Follow(E, a)
2 | **return** FollowRec(E, a, NewMap())
3 Function FollowRec(E, a, $subMap$)
4 | **switch** E **do**
5 | | **case** $E_1 + E_2$ **do**
6 | | | **return** FollowRec(E_1, a, $subMap$) $\cup$ FollowRec(E_2, a, $subMap$)
7 | | **case** $E_1 \cdot \square E_2$ **do**
8 | | | $subMapL \leftarrow subMap$ /* copy map */
9 | | | $subMapL[\square] \leftarrow$ First(E_2)   /* replace mapping for $\square$ */
10 | | | **return** FollowRec(E_1, a, $subMapL$) $\cup$ FollowRec(E_2, a, $subMap$)
11 | | **case** $E_1^{*,\square}$ **do**
12 | | | $subMap[\square] \leftarrow subMap[\square] \cup$ First(E_1)
13 | | | **return** FollowRec(E_1, a, $subMap$)
14 | | **case** $f(E_1, E_2, \ldots, E_n)$ **do**
15 | | | **if** $a = f$ **then return** ReplaceConstants($subMap$, $E_1, E_2, \ldots, E_n$)
16 | | | **else return** $\bigcup_{i=1}^{n}$ FollowRec(E_i, a, $subMap$)
17 | | **case** $\emptyset$ **do**
18 | | | **return** $\emptyset$
19 Function ReplaceConstants($subMap$, $E_1, E_2, \ldots, E_n$)
20 | $lst \leftarrow$ NewList()
21 | **for** E_i *in* $E_1, E_2, \ldots, E_n$ **do**
22 | | **if** $E_i \in \mathcal{K}$ **then** $lst \leftarrow$ Append(lst, $subMap[c]$)   /* child is a $\square$ */
23 | | **else** $lst \leftarrow$ Append(lst, First(E_i))
24 | **return** CartesianProduct(lst)

▶ **Example 7.** Let E be a RTE from Figure 2a. $\text{First}(E') = \{b0_2, a2_1, a2_3\}$. The results of the function First and the substitution map for individual nodes are illustrated in Figure 3. $\text{Follow}(E', a2_1) = \{(a2_3, a2_3), (a2_3, a2_1), (a2_3, b0_2), (a2_1, a2_3), (a2_1, a2_1), (a2_1, b0_2), (b0_2, a2_3), (b0_2, a2_1), (b0_2, b0_2)\}$. $\text{Follow}(E', a2_3) = \{(b0_4, b0_5)\}$. Follow of leaves is $\emptyset$.

3.2 Pushdown Automaton Construction

In the previous section, it was shown how to compute First and Follow sets. The First set determines what symbols are the last to be read in the postfix notation. The Follow sets store the information about the direct children of a node. This information is used to create transitions of the two state PDA that accepts by final state. The automaton reads a linear postfix notation of a tree with an end-of-string marker ⊣ appended to the end of the input. Technical helper functions φ and σ are presented first.

▶ **Definition 8.** Function φ maps an element (tuple) of $\text{Follow}(E', a)$ to a string of pushdown store symbols. The resulting string is ε if the size of the tuple is zero. Mapping σ strips the unique index from the subscripted symbol.

▶ **Example 9.** Let E' be a subscripted RTE and let $f = (a2_1, b2_2, c0_3)$ be a follow tuple of some node. Then $\varphi(f) = a2_1 b2_2 c0_3$. Also $\sigma(a2_1) = a2$.

First = $\{a2_1, b0_2, a2_3\}$
$\{\square \to \{\}\}$

First = $\{a2_1, b0_2, a2_3\}$
$\{\square \to \{b0_5\}\}$

First = $\{a2_1, b0_2, \square\}$
$\{\square \to \{a2_3\}\}$

First = $\{a2_1, b0_2\}$
$\{\square \to \{a2_1, a2_3, b0_2\}\}$

First = $\{a2_1\}$
$\{\square \to \{a2_1, a2_3, b0_2\}\}$

First = $\{\square\}$
$\{\square \to \{a2_1, a2_3, b0_2\}\}$

First = $\{b0_5\}$
$\{\square \to \{\}\}$

First = $\{a2_3\}$
$\{\square \to \{b0_5\}\}$

First = $\{b0_4\}$
$\{\square \to \{b0_5\}\}$

First = $\{\square\}$
$\{\square \to \{b0_5\}\}$

First = $\{b0_2\}$
$\{\square \to \{a2_1, a2_3, b0_2\}\}$
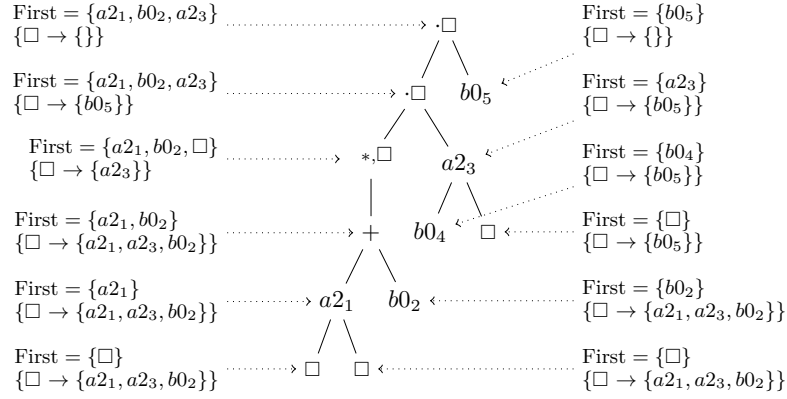
First = $\{\square\}$
$\{\square \to \{a2_1, a2_3, b0_2\}\}$

Figure 3 RTE E' from Figure 2a with First set and substitution mapping for each node.

Algorithm 2: PDA accepting linearised trees described by an RTE E.

---

input : RTE E.

output: PDA A such that $L(A) = post(L(E))$

Create PDA with the following properties:

- Set of states is equal to $\{q, f\}$,
- alphabet is equal to the $\mathcal{F}$ alphabet of E, and $\dashv$ symbol,
- pushdown store alphabet is equal to the symbols $\mathrm{Pos}(E') \cup \{\bot\}$,
- mapping δ can be created by these rules:
  - $\forall a_i \in \mathrm{Pos}(E')$, $\sigma(a_i) \in \mathcal{F}_0$, add transition $\delta(q, \sigma(a_i), \varepsilon) = \{(q, a_i)\}$,
  - $\forall a_i \in \mathrm{Pos}(E')$, $\sigma(a_i) \notin \mathcal{F}_0$, $\forall f \in \mathrm{Follow}(E', a_i)$, add $\delta(q, \sigma(a_i), \varphi(f)) = \{(q, a_i)\}$,
  - $\forall a_i \in \mathrm{First}(E')$ add transition $\delta(q, \dashv, \bot\, a_i) = \{(f, \varepsilon)\}$.

Resulting PDA is $A = (\{q, f\}, \mathcal{F} \cup \{\dashv\}, \mathrm{Pos}(E') \cup \{\bot\}, \delta, \bot, q, \{f\})$. Automaton accepts by the final state. The top of the pushdown store is on the right.

Roots of subtrees are stored on the pushdown store to keep track of which subtrees have been read so far. When the root of a subtree is read, its children have to be on the top of the pushdown store. They are replaced by a pushdown store symbol corresponding to the read symbol. PDA recognising postfix notations of trees described by RTE E ($post(L(E))$) is constructed by Algorithm 2.

▶ **Example 10.** This example expands on Example 7. The PDA $A = (\{q, f\}, \{a2, b0, \dashv\}, \{\bot, a2_1, b0_2, a2_3, b0_4, b0_5\}, \delta, \bot, q, \{f\})$ is constructed by Algorithm 2. δ is defined as follows:

$$\delta(q, a2, a2_3 a2_3) = \{(q, a2_1)\} \quad \delta(q, a2, b0_2 a2_1) = \{(q, a2_1)\} \quad \delta(q, a2, a2_3 a2_1) = \{(q, a2_1)\}$$
$$\delta(q, a2, b0_2 b0_2) = \{(q, a2_1)\} \quad \delta(q, a2, a2_3 b0_2) = \{(q, a2_1)\} \quad \delta(q, a2, b0_4 b0_5) = \{(q, a2_3)\}$$
$$\delta(q, a2, a2_1 a2_3) = \{(q, a2_1)\} \quad \delta(q, a2, a2_1 a2_1) = \{(q, a2_1)\} \quad \delta(q, a2, a2_1 b0_2) = \{(q, a2_1)\}$$
$$\delta(q, a2, b0_2 a2_3) = \{(q, a2_1)\} \qquad \delta(q, b0, \varepsilon) = \{(q, b0_2), (q, b0_4), (q, b0_5)\}$$
$$\delta(q, \dashv, \bot\, b0_2) = \{(f, \varepsilon)\} \qquad \delta(q, \dashv, \bot\, a2_3) = \{(f, \varepsilon)\} \qquad \delta(q, \dashv, \bot\, a2_1) = \{(f, \varepsilon)\}$$

The RTE from Figure 2a describes, for instance, the tree from Figure 1. This tree in its postfix notation (with $\dashv$ symbol appended) is accepted by the automaton.

▶ **Theorem 11.** *Algorithm 2 creates PDA A such that $L(A) = post(L(E))$.*

Proof. The proof consists of two parts: $post(L(E)) \subseteq L(A)$ and $L(A) \subseteq post(L(E))$.

(a) A RTE.   **(b)** The PDA for the RTE from Figure 4a.
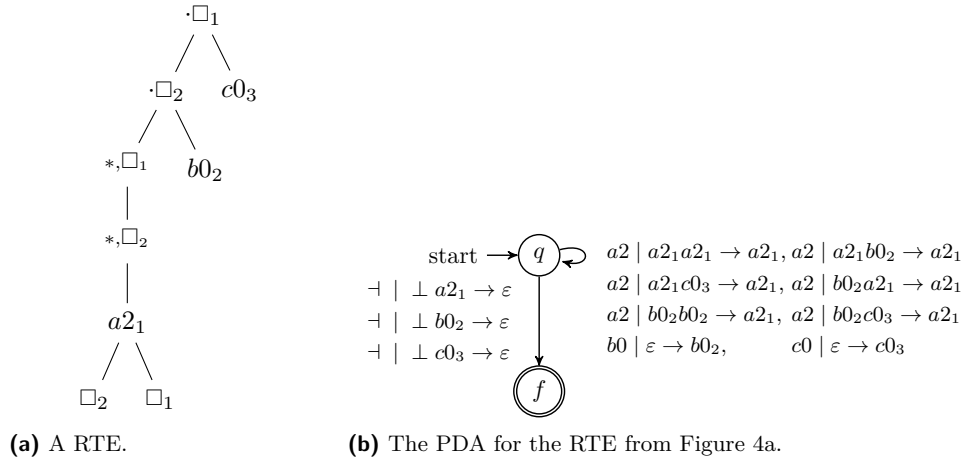
■ **Figure 4** A RTE and its equivalent PDA.

Case $post(L(E)) \subseteq L(A)$: Proof comes directly from the proof of the Follow and First functions. The functions analyse all possible combinations of parent-children relations. The relations are used in the transition function of the PDA. When an input tree (except $\dashv$ symbol) is read, the automaton can continue only if the pushdown store content equals to the string $\perp f$ ($f \in$ First(E') set) to ensure that whole tree was read.

Case $L(A) \subseteq post(L(E))$: If there is a word from $L(A)$ that is not in $post(L(E))$ then either the computation of First or Follow functions were wrong or the transitions created from Follow sets would allow the automaton to accept something more. The functions First and Follow are proved to be correct.   ◀

▶ **Theorem 12.** *Algorithm 2 creates a real-time height-deterministic PDA.*

Proof. Transitions PDA always pop arity(a) symbols from the pushdown store and push one symbol when reading symbol a. Reading symbol $\dashv$ pops two symbols and pushes none. The pushdown store height is predetermined and same for all nondeterministic computations of the PDA on any string. This fulfills the conditions of height-determinism. The PDA never reads ε, therefore it is also real-time [13, 15].   ◀

The PDA has two properties worth mentioning: The function ReplaceConstants from Algorithm 1 has an exponential output with the number of node's children that are from $\mathcal{K}$ and the size of $subMap[\square]$ for given node. As every element from the Follow set results in one transition, the PDA has an exponential amount of transitions. Also, Theorem 12 shows that the PDA is determinisable because the PDA is real-time height-deterministic [13, 15].

▶ **Example 13.** RTE E' (Figure 4a) has the following properties: First$(E') = \{a2_1, b0_2, c0_3\}$, Follow$(E', a2_1) = \{(a2_1, a2_1), (a2_1, b0_2), (a2_1, c0_3), (b0_2, a2_1), (b0_2, b0_2), (b0_2, c0_3)\}$ and Follow$(E', b0_2) =$ Follow$(E', c0_3) = \emptyset$. The equivalent PDA is illustrated in Figure 4b.

4   Reducing the Number of Transitions

The PDA created by Algorithm 2 has an exponential number of transitions. The transition function δ enumerated all possible tuples of children for every node in the tree.

The idea behind the improvement is to make a better use of the pushdown store. New pushdown store symbols representing all possible symbols that can appear in the place of a

---

Algorithm 3: Improved PDA accepting linearised trees described by a RTE E.

---

input : RTE E.

output : PDA A such that $L(A) = post(L(E))$

Create PDA with following properties:

- Set of states is equal to $\{q, f\}$,
- input alphabet is $\mathcal{F} \cup \{\dashv\}$,
- pushdown store alphabet consists of all sets that appear in substitution mapping in $\square \in \mathcal{K}$ nodes and singletons consisting of indexed occurrences of symbols from $\mathcal{F}$,
- transitions (δ) are created by these rules:
  1. for all symbols $a_i \in \mathcal{F}$ add transition $\delta(q, \sigma(a_i), \varphi(\text{Follow}(E, a_i))) = \{(q, \{a_i\})\}$,
  2. for all nodes $\square_i$ labelled with a $\square \in \mathcal{K}$, for all symbols $a_i \in subMap_{\square_i}[\square]$ add
     $\delta(q, \sigma(a_i), \varphi(\text{Follow}(E, a_i))) = \{(q, subMap_{\square_i}[\square])\}$,
  3. for all symbols $a_i \in \text{First}(E')$ add transition $\delta(q, \dashv, \bot \{a_i\}) = \{(f, \varepsilon)\}$.

Resulting PDA is $A = (\{q, f\}, \mathcal{F} \cup \{\dashv\}, \{\{a_i\} \mid a_i \in \mathcal{F}\} \cup \{subMap_{\square_i}[\square] \mid \text{for all}$ nodes $\square_i$ labelled with a $\square \in \mathcal{K}\}, \delta, \bot, q, \{f\})$. Automaton accepts by the final state. The top of the pushdown store is on the right.

---

symbol $\square \in \mathcal{K}$ are introduced. These symbols effectively represent the complete substitution mapping. For every occurrence of the symbol $\square \in \mathcal{K}$ the substitution mapping set for this occurrence is to be added as a new pushdown store symbol.

The only difference in the analysis of the RTE is the Follow algorithm. On line 24, the computation is altered by removing the computation of Cartesian product and returning the list *lst* instead. This excludes the need for computing the Cartesian product. Furthermore, every symbol of $\mathcal{F}$ alphabet is now followed by exactly one tuple.

The ideas from previous paragraphs are applied in the Algorithm 3. The algorithm constructs an improved PDA which has an asymptotically lower amount of transitions.

▶ **Definition 14.** Let $subMap_{\triangle}[\square]$ return the substitution mapping for symbol $\square \in \mathcal{K}$ inside the $\triangle$ node of the syntax tree.

▶ **Theorem 15.** *Algorithm 3 creates a real-time height-deterministic PDA.*

Proof. Similar to the proof of Theorem 12. ◀

The automaton created by Algorithm 3 is determinisable.

▶ **Theorem 16.** *Algorithm 3 creates PDA equivalent to RTE E in $O(|E|^2)$ time and the number of transitions of the PDA is $O(\|\mathcal{F}_E\| \|\mathcal{K}_E\|)$.*

Proof. Overall time complexity can be determined from the efficient implementation of the algorithm. Computing and saving the First set takes $O(|E| \|\mathcal{F}_E\|)$ time. The substitution mapping can be computed in one traversal over the syntax tree of RTE. It is saved as a mapping from every occurrence of a node from $\mathcal{K}_E$ alphabet to the set of elements from $\mathcal{F}_E$. The Follow elements can be computed in the same traversal. This takes $O(|E|^2)$ time. Rules of type 1 and 3 are created in $O(\|\mathcal{F}_E\|)$ time from the Follow mapping and First set, respectively. While creating type 2 rules, for every $\mathcal{F}_E$ node it is required to iterate over the saved substitution mapping. Therefore, creating type 2 rules takes $O(\|\mathcal{F}_E\| \|\mathcal{K}_E\|)$ time.

The overall time complexity is $O(|E| \|\mathcal{F}_E\| + |E| |E| + \|\mathcal{F}_E\| \|\mathcal{K}_E\|) = O(|E|^2)$ as $\|\mathcal{K}_E\| \leq |E|$ and $\|\mathcal{F}_E\| \leq |E|$. The number of transitions is $O(\|\mathcal{F}_E\| \|\mathcal{K}_E\|)$ because there are $O(\|\mathcal{F}_E\|)$ transitions of types 1 and 3, and $O(\|\mathcal{F}_E\| \|\mathcal{K}_E\|)$ transitions of type 2. ◀
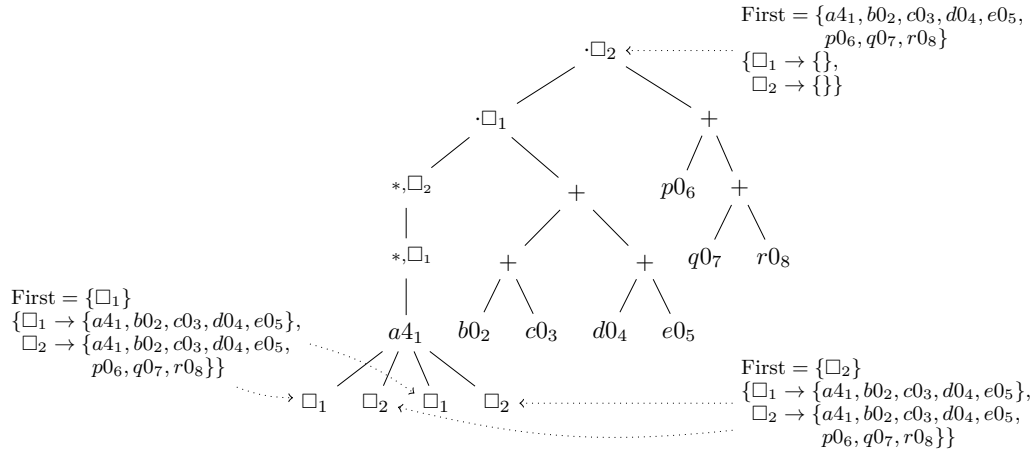
Figure 5 Sample RTE with First set and substitution mapping for important nodes.
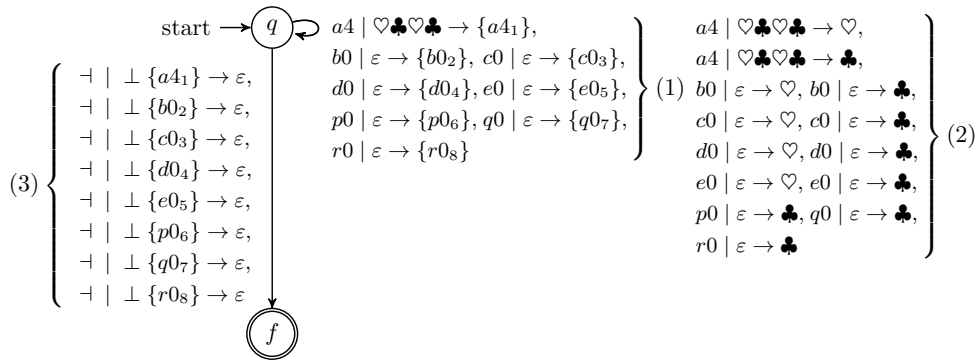
Figure 6 PDA equivalent to a RTE from Figure 5 with transitions grouped by type. For readability, symbol $\heartsuit$ stands for $\{a4_1, b0_2, c0_3, d0_4, e0_5\}$ and symbol $\clubsuit$ stands for $\{a4_1, b0_2, c0_3, d0_4, e0_5, p0_6, q0_7, r0_8\}$.

▶ **Example 17.** RTE E' from Figure 5 converted to equivalent PDA (Figure 6). First$(E') = \{a4_1, b0_2, c0_3, d0_4, e0_5, p0_6, q0_7, r0_8\}$. Follow$(E', a4_1) = (\{a4_1, b0_2, c0_3, d0_4, e0_5\}, \{a4_1, b0_2, c0_3, d0_4, e0_5, p0_6, q0_7, r0_8\}, \{a4_1, b0_2, c0_3, d0_4, e0_5\}, \{a4_1, b0_2, c0_3, d0_4, e0_5, p0_6, q0_7, r0_8\})$. Follow of other symbols (leaves) is $\emptyset$. Note that if the Follow was computed by Algorithm 1 then $|\text{Follow}(E', a4_1)| = 1600$.

5 Conclusion and Future Work

A new algorithm for the conversion of a RTE to a PDA has been described. The resulted PDA accepts all trees from the language described by the RTE in their linear postfix notation. Presented PDA belongs to the class of real-time height-deterministic PDAs, therefore it can always be determinised [15].

The presented algorithm creates the PDA in quadratic time w.r.t. to the size of input RTE's syntax tree, i.e. in $O(|E|^2)$ time. The number of transitions in the PDA is $O(\|\mathcal{F}_E\| \|\mathcal{K}_E\|)$.

There is also a number of interesting open problems. As the processing of the RTE is similar to processing the regular expression for the Glushkov's algorithm, we hope to explore more similarities with this algorithm. Although searching thoroughly, we have not found

an algorithm for the reverse conversion (from a finite tree automaton or a PDA to a RTE). Finally, we would like to explore the tree pattern matching problem where the definition of a set of tree patterns is represented by RTEs.

References

1 Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling. 1: Parsing.* Prentice-Hall, 1972.

2 Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *36th Symposium on Theory of Computing*, pages 202–211, 2004. `doi:10.1145/1007352.1007390`.

3 Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

4 Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964. `doi:10.1145/321239.321249`.

5 Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Christof Löding, Sofia Tison, and Marc Tommasi. Tree automata techniques and applications, 2007. Available on: `http://www.grappa.univ-lille3.fr/tata`.

6 Tomáš Flouri, Jan Janoušek, and Bořivoj Melichar. Subtree matching by pushdown automata. *Computer Science and Information Systems*, 7(2):331–357, 2010.

7 Victor Mikhailovich Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5), 1961. `doi:10.1070/RM1961v016n05ABEH004112`.

8 John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, 2nd edition, 2003.

9 Jan Janoušek and Bořivoj Melichar. On regular tree languages and deterministic pushdown automata. *Acta Informatica*, 46(7):533–547, 2009. `doi:10.1007/s00236-009-0104-9`.

10 Viraj Kumar, Parthasarathy Madhusudan, and Mahesh Viswanathan. Visibly pushdown automata for streaming XML. In *16th International Conference on World Wide Web*, pages 1053–1062, 2007. `doi:10.1145/1242572.1242714`.

11 Dietrich Kuske and Ingmar Meinecke. Construction of tree automata from regular expressions. In *12th International Conference on Developments in Language Theory*, pages 491–503, 2008. `doi:10.1007/978-3-540-85780-8_39`.

12 Éric Laugerotte, Nadia Ouali Sebti, and Djelloul Ziadi. From regular tree expression to position tree automaton. In *7th International Conference on Language and Automata Theory and Applications (LATA)*, pages 395–406, 2013. `doi:10.1007/978-3-642-37064-9_35`.

13 Dirk Nowotka and Jiří Srba. Height-deterministic pushdown automata. In *32nd International Symposium Mathematical Foundations of Computer Science (MFCS)*, pages 125–134, 2007. `doi:10.1007/978-3-540-74456-6_13`.

14 Radomír Polách, Jan Janoušek, and Bořivoj Melichar. Regular tree expressions and deterministic pushdown automata. In *7th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 70–77, 2011.

15 Radomír Polách, Jan Trávníček, Jan Janoušek, and Bořivoj Melichar. Efficient determinization of visibly and height-deterministic pushdown automata. *Computer Languages, Systems & Structures*, 46:91–105, 2016. `doi:10.1016/j.cl.2016.07.005`.

16 Grzegorz Rozenberg and Arto Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar.* Springer-Verlag, 1997.

17 Tom Sebastian and Joachim Niehren. Projection for nested word automata speeds up xpath evaluation on XML streams. In *42nd International Conference on Current Trends in Theory and Practice of Computer Science*, pages 602–614, 2016. `doi:10.1007/978-3-662-49192-8_49`.

18 Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968. `doi:10.1145/363347.363387`.

Context-Oriented Algorithmic Design

Bruno Ferreira
Instituto Superior Técnico/INESC-ID, Lisbon, Portugal
bruno.b.ferreira@tecnico.ulisboa.pt
 https://orcid.org/0000-0001-8227-2648

António Menezes Leitão
Instituto Superior Técnico/INESC-ID, Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt
 https://orcid.org/0000-0001-7216-4934

── Abstract ──

Currently, algorithmic approaches are being introduced in several areas of expertise, namely Architecture. Algorithmic Design (AD) is an approach for architecture that takes advantage of algorithms to produce complex designs, to simplify the exploration of variations, or to mechanize tasks, including those related to analysis and optimization of designs. However, architects might need different models of the same design for different kinds of analysis, which tempts them to extend the same code base for different purposes, typically making the code brittle and hard to understand. In this paper, we propose to extend AD with Context-Oriented Programming (COP), a programming paradigm based on context that dynamically changes the behavior of the code. To this end, we propose a COP library and we explore its combination with an AD tool. Finally, we implement two case studies with our context-oriented approach, and discuss their advantages and disadvantages when compared to the traditional AD approach.

1 Introduction

Nowadays, Computer Science is being introduced in several areas of expertise, leading to new approaches in areas such as Architecture. Algorithmic Design (AD) is one of such approaches, and can be defined as the production of Computer-Aided Design (CAD) and Building Information Modeling (BIM) models through algorithms [21]. This approach can be used to produce complex models of buildings that could not be created with traditional means, and its parametric nature allows an easier exploration of variations.

Due to these advantages, AD started to be introduced in CAD and BIM applications, which lead to the development of tools that support AD programs, such as Grasshopper. However, with the complexity of the models came the necessity of analyzing the produced solutions with analysis tools. For this task, the geometrical models are no longer sufficient, as analysis software usually requires special analytical models, that are different from geometrical models and can not be obtained with import/export mechanisms due to errors. This leads to the production of several models, which have to be kept and developed in parallel, involving different development lines that are hard to manage and to keep synchronized. This complex workflow proves that current solutions are not sufficient [29].

Some tools like Rosetta [20] address these issues by offering portable AD programs between different CAD applications and, more recently, BIM applications [6] and analysis tools [18]. Nevertheless, Rosetta does not offer a unifying description capable of producing both the geometrical and the analytical models with the same AD program, which can lead to the cluttering of the current program in an effort to reduce the number of files to maintain.

To solve these problems, we propose the use of COP to develop a computational model capable of adapting itself to the required context, which in this case is defined by the requirements of modeling applications and analysis software, allowing the production of different models with a change of context.

1.1 Context-Oriented Programming

COP was first introduced as a new programming approach that takes context into account [7]. According to a more recent depiction of this approach, COP aims to give users ways to deal with context in an explicit way in their programs, making it accessible to manipulate through software with features that are usually lacking in mainstream programming languages [12].

With this approach, users can express different behaviors in terms of the context of the system. This context is composed of the actors of the system, which can determine how the system is used, the environment in which the system is, which can restrict or influence its functionality, and the system itself, whose changes might lead to different responses.

Although there are different implementations of COP, which will be presented later in this paper, according to [12] necessary properties must be addressed by all of them:

- behavioral variation: implementations of behavior for each context;
- layers: a way to group related context-dependent variations;
- activation and deactivation of layers: a way to dynamically enable or disable layers, based on the current context;
- context: information that is accessible to the program and can be used to determine behavioral variations;
- scoping: the scope in which layers are active or inactive and can be controlled.

With these features, layers can be activated or deactivated dynamically in arbitrary places in the code, resulting in behaviors that fit the different contexts the program goes through during its execution. If analyzed in terms of multi-dimensional dispatch [27], it is possible to say that COP has four-dimensional dispatch, since it considers the message, the receiver, the sender, and the context to determine which methods or partial method definitions are included or excluded from message dispatch. These method definitions are used to implement behavioral variations in layers, which can be expressed differently in the different COP implementations. In some, the adopted approach is known as *class-in-layer*, in which layers are defined outside the lexical scope of modules [1], in a manner similar to aspects from Aspect-Oriented Programming (AOP) [17]. In others, a *layer-in-class* approach is used, having the layer declarations within the lexical scope of the modules.

In addition to the base concepts, each implementations has its own operators and strategies that might expand the capabilities of COP. The advantages and disadvantages of each of these implementations will be discussed later in the paper.

1.2 Objectives

The main objectives of this paper are: (1) present and compare the different implementations for COP that have been proposed by the research community, and (2) present two simple case studies that show how COP can be applied in AD. The case studies consist of previously

developed AD programs that we re-implemented with our proposed solution and then used to produce different models according to different context. Finally, the results of our solution are compared to the ones obtained previously.

In the next section, we start by presenting the related work, as well as a comparison between the different implementations of COP.

2 Related Work

In this section, we introduce several paradigms that served as basis for COP, namely AOP and Subject-Oriented Programming (SOP), as well as the more relevant implementations of COP. We end this section with a comparison between these implementations.

2.1 Aspect-Oriented Programming

Most programming paradigms, such as Object-Oriented Programming (OOP), offer some way to modularize the concerns necessary to implement a complex system. However, it is common to encounter some concerns that do not fit the overall decomposition of the system, being scattered across several modules. These concerns are known as *crosscutting concerns*.

AOP was created to deal with these *crosscutting concerns*, introducing ways to specify them in a modularized manner, called *aspects*. *Aspects* can be implemented with proper isolation and composition, making the code easier to maintain and reuse [17]. Using AOP, it is possible to coordinate the *crosscutting concerns* with normal concerns, in well defined points of the program, known as *join points*.

In addition to these concepts, AOP implementations, such as AspectJ, introduce addional ones, such as *pointcuts*, which are collections of *join points*, and *advices*, which are implementations of behavior attached to *pointcuts* [16]. An *aspect* is then a module that implements a *crosscutting concern*, comprised of *pointcuts* and *advices*. AspectJ also offers *cflow* constructs, that allow the expression of control-flow-dependent behavior variations, making it possible to conditionally compose behavior with *if pointcuts* [16].

There are similarities between AOP and COP: both make it possible to define behavior that depends on a condition, which can, in itself, be considered a context. However, while AOP aims to modularize *crosscutting concerns*, this is not mandatory in COP, since the use of *layer-in-class* approaches scatters the code across several modules. In addition, COP allows the activation and deactivation of layers in arbitrary pieces of code, while AOP triggers *pointcuts* at very specific *join points* that occur in the rest of the program [12]. This makes COP more flexible in dealing with behavioral variation.

2.2 Subject-Oriented Programming

SOP is a programming paradigm that introduces the concept of *subject* to facilitate the development of cooperative applications. These are defined by the combination of *subjects*, which describe the state and behaviors of objects that are relevant to them [10].

The *Subjects*' goal is to introduce a perception of an object, such as it is seen by a given application. *Subjects* do so by adding classes, state, and behavior, according to the needs of that application. By doing this, each application can use a shared object through the operations defined for its *subject*, not needing to know the details of the object described by other *subjects* [10].

Subjects can also be combined in groups called *compositions*, which define a composition rule, explaining how classes and methods from different *subjects* can be combined. These can then be used through *subject-activation*, which provides an executable instance of the *subject*, including all the data that can be manipulated by it.

Due to the introduction of all these concepts, SOP offers what is known as *Subjective Dispatch* [27]. This extends the dispatch introduced by OOP, by adding the sender dimension, in addition to the message and the receiver. This was later expanded by COP, which introduces a dimension for context, as mentioned previously.

It is possible to see that, similarly to the other analyzed paradigms, SOP also supports behavior variations in the form of *subjects*. However, if we consider that each *subject* might have different contexts of execution, we need an extra dimension for dispatch, which is what COP offers.

2.3   COP Implementations

COP was proposed as an approach that allows the user to explore behavioral variations based on context. Although this approach introduces concepts such as layers and contexts, each of the implementations address the concepts differently, sometimes due to the support of the host language in which the COP constructs are implemented. In this section we present the different implementations available.

2.3.1   ContextL

ContextL was one of the first programming language extensions to introduce support for COP. It implements the features discussed previously by taking advantage of the Common-Lisp Object System (CLOS) [4].

The first feature to be considered is the implementation of layers, which are essential to implement the remaining features available in ContextL [5]. These layers can be activated dynamically throughout the code, since ContextL uses an approach called Dynamically Scoped Activation (DSA), where layers are explicitly activated and a part of the program is executed under that activation. The layer is active while the contained code is executing, becoming inactive when the control flow returns from the layer activation.

Regarding the activation of multiple layers, it is important to note that the approach introduced in ContextL, as well as in other implementations that support DSA, follows a stack-like discipline. Also, in ContextL this activation only affects the current thread.

By taking advantage of layers, it is then possible to define classes in specific layers, so that the classes can have several degrees of detail in different layers, introducing behavior that will only be executed when specific layers are activated. The class behavior can also be defined with *layered generic functions*. These take advantage of the generic functions from CLOS, and are instances of a generic function class named *layered-function* [5].

In addition, ContextL supports contextual variations in the definition of class slots as well. Slots can be declared as *:layered*, which makes the slot accessible through layered-functions. This introduces slots that are only relevant in specific contexts.

By looking at the constructs implemented in ContextL, it is possible to conclude that behavioral variations can be implemented in specific classes or outside of them. This means that ContextL supports both *layer-in-class* and *class-in-layer* approaches. The former allows the definition of partial methods to access private elements of enclosing classes, something that the latter does not support, since *class-in-layer* specification cannot break encapsulation [1].

Finally, it should be noted that ContextL follows a library implementation strategy: it does not implement a source-to-source compiler, and all the constructs that support the COP features are integrated in CLOS by using the Metaobject Protocol [15].

2.3.2   PyContext and ContextPy

PyContext was the first implementation of COP for the Python programming language. Although it includes most of the COP constructs in a similar manner to other implementations, it also introduces new mechanisms for layer activation, as well as to deal with variables.

Explicit layer activation is an appropriate mechanism for several problems but, sometimes, this activation might violate modularity. Since the behavioral variation may occur due to a state change that can happen at any time during the program execution, the user needs to insert verifications in several parts of the program, increasing the amount of scattered code. To deal with this, PyContext introduces *implicit layer activation*. Each layer has a method *active*, which determines if a layer is active or not. This, in combination with a function *layers.register_implicit*, allows the framework to determine which layers are active during a method call, in order to produce the correct method combination [30].

Regarding variables, PyContext offers contextual variables, which can be used with a *with* statement in order to maintain their value in the dynamic scope of the construct. These variables are called *dynamic variables*. These are globally accessible, and their value is dynamically determined when entering the scope of a *with* construct [30]. In conjunction with specific getters and setters, it is possible to get the value of the variable, change it in a specific context, and then have it restored when exiting the scope of that context. It is important to note that this feature is thread-local.

As for the other features, PyContext does not modify the Python Virtual Machine, being implemented as a library. Layers are implemented using meta-programming, and layer activation mechanisms take advantage of Python's context handlers. As for the partial definition of methods and classes, PyContext follows a *class-in-layer* approach.

More recently, ContextPy was developed as another implementation of COP for the Python language. This implementation follows a more traditional approach to COP, offering Dynamically Scoped Layer activation, using the *with* statement, which follows a stack-like approach for method composition. For partial definitions, ContextPy follows a *layer-in-class* approach, taking advantage of decorators to annotate base methods, as well as the definitions that replace those methods when a specific layer is active [13]. Finally, similarly to PyContext, ContextPy is offered as a library that can be easily included in a Python project.

2.3.3   ContextJ

ContextJ is an implementation of COP for the Java programming language, and one of the first implementations of this approach for statically typed programming languages.

ContextJ is a sorce-to-source compiler solution that introduces all the concepts of COP in Java by extending the language with the *layer*, *with*, *without*, *proceed*, *before* and *after* terminal symbols [2]. Layers are included in the language as a non-instantiable type, and their definitions follows a *layer-in-class* approach. Each layer is composed of an identifier and a list of partial method definitions, whose signature must correspond to one of the methods of the class that defines the layer. Also, to use the defined layers, users must include a layer import declaration on their program, in order to make the layer type visible.

As for partial method definitions, they override the default method definition and can be combined, depending on the active layers. The *before* and *after* modifiers can also be used in partial method definitions, in order to include behavior that must be executed before and after the method execution. In addition, the *proceed* method can be used to execute the next partial definition that corresponds to the next active layer, allowing the combination of behavioral variations [2].

Regarding layer activation, ContextJ supports dynamically scoped layer activation by using a *with* block. Layers are only active during the scope of the block, and the activation is thread-local. *With* blocks can be nested, and the active layer list is traversed according to a stack approach. This, in combination with the *proceed* function, allows the user to compose complex behavior variations. In addition, it is possible to use the *without* block to deactivate a layer during its scope, in order to obtain a composition without the partial method definitions of that specific layer.

Finally, ContextJ also offers a reflection Application Programming Interface (API) for COP constructs. It includes classes for *Layer*, *Composition*, and *PartialMethod*, along with methods that support runtime inspection and manipulation of these concepts.

2.3.4   Other COP Implementations

The implementations described in the previous sections present the major strategies and features that are currently used with COP. Nevertheless, there are more implementations for other languages, which we briefly describe in this section.

Besides ContextJ, there are other implementations of COP for Java, namely JCop [3] and EventCJ [14], which use join-point events to switch layers, cj [25], a subset of ContextJ that runs on an ad hoc Java virtual machine, and JavaCtx [22], a library that introduces COP semantics by weaving aspects.

There are also COP implementations for languages such as Ruby, Lua, and Smalltalk, namely ContextR [26], ContextLua [31], and ContextS [11] respectively. ContextR introduces reflection mechanisms to query layers, while ContextLua was conceived to introduce COP in games. ContextS follows the more traditional COP implementations, such as ContextL.

In addition, some implementations, such as ContextErlang, introduce COP in different paradigms, like the actor model. ContextErlang also introduces different ways to combine layers, namely per-agent variation activation and composition [24].

Regarding layer combination and activation, there are also implementations that offer strategies that differ from dynamic activation. One example is ContextJS [19] that offers a solution based on open implementation, in which layer composition strategies are encapsulated in objects. These strategies can add new scoping mechanisms, disable layers, or introduce a new layer composition behavior that works better with a domain-specific problem [19].

More recently, Ambience [9], Subjective-C [8], and Lambic [28] were developed. Ambience uses the amOS language and context objects to implement behavioral variations, with the context dispatch made through multi-methods. Subjective-C introduces a Domain Specific Language (DSL) that supports the definition of constraints and the activation of behaviors for each context. Finally, Lambic is a COP implementation for Common Lisp that uses predicate dispatching to produce different behavioral variations.

In the next section we present a comparison between all these implementations, as well as the advantages and disadvantages of using each one.

2.4   Comparison

Table 1 shows a comparison between the analyzed COP implementations.

As it is possible to see, most of the analyzed implementations are libraries, with source-to-source compilers being mostly used in statically typed programming languages. The library implementation has advantages when trying to add COP in an already existing project, since it does not change the language and uses the available constructs. On the other hand, source-to-source compilers, such as ContextJ, introduce new syntax that simplifies the COP mechanics.

■ **Table 1** Comparison between the COP implementations. DSA stands for Dynamically Scoped Activation, LIC for *layer-in-class*, and CIL for *class-in-layer*. Lambic uses predicate dispatching instead of layers, so the last two columns do not apply. Adapted from [23].

	Base Language	Implementation	Layer Activation	Modularization
ContextL	Common Lisp	Library	DSA	LIC, CIL
ContextErlang	Erlang	Library	Per-agent	Erlang Modules
ContextJS	JavaScript	Library	Open Implementation	LIC, CIL
PyContext	Python	Library	DSA, Implicit Layer Activation	CIL
ContextPy	Python	Library	DSA	LIC
ContextJ	Java	Source-to-Source Compiler	DSA	LIC
JCop	Java	Source-to-Source and Aspect Compiler	DSA, declarative layer composition, conditional composition	LIC
EventCJ	Java	Source-to-Source and Aspect Compiler	DSA	LIC
JavaCtx	Java	Library and Aspect Compiler	DSA	LIC
ContextR	Ruby	Library	DSA	LIC
ContextLua	Lua	Library	DSA	CIL
ContextS	Smalltalk	Library	DSA, indefinite activation	CIL
Ambience	AmOS	Library	DSA, global activation	CIL
Lambic	Common Lisp	Library	-	-
Subjective-C	Objective-C	Preprocessor	Global Activation	LIC

As for layer activation, the most common strategy is DSA. However, to increase flexibility, some solutions introduce indefinite activation, global activation, per agent activation or, in the case of ContextJS, an open implementation, allowing users to implement an activation mechanism that best fits the problem they are solving. Although DSA is appropriate for most problems, other strategies might be best suited for multi-threaded applications or problems whose contexts depend on conditions that cannot be captured with the default layer activation approach.

Finally, regarding modularization, it is possible to see that most implementations use the *class-in-layer* or the *layer-in-class* approach. The former allows users to create modules with all the concerns regarding a specific context, while the latter places all the behaviors on the class affected by the contexts. Hence, *class-in-layer* reduces code scattering, while *layer-in-class* simplifies program comprehension. There are implementations that support both approaches, such as ContextL, but usually the supported approach is restricted by the language's features. Nevertheless, there are cases, such as ContextPy and PyContext, that operate on the same programming language but follow different principles regarding the COP concepts.

All these implementation support the COP paradigm, although they offer different variations of the relevant concepts. Choosing the most appropriate implementation requires a careful examination of their distinct features, and how they help in fulfilling the requirements of the problem at hand.

3 Context-Oriented Algorithmic Design

In this section, we propose to combine COP with AD, introducing what we call Context-Oriented Algorithmic Design. Since it is common for architects to produce several different models for the same project, depending on the intended use (e.g., for analysis or rendering), we define these different purposes as contexts. By doing this, it is possible to explicitly say which type of model is going to be produced.

In addition, we introduce definitions for the design primitives using COP as well. For each primitive, we can define different behavioral variations, depending on the model we

want to produce. For example, since some analysis models require surfaces instead of solids, a primitive function like *Wall* would produce a box in a 3D context and a simple surface in an analysis context.

Finally, since COP allows the combination of layers, we can take advantage of that to combine concepts, such as Level of Detail (LOD) with the remaining ones. This allows more flexibility while exploring variations, since it does not only support the exploration in several contexts, but also the variation of LOD inside the same context, as architects might want lesser detail in certain phases to obtain quicker results.

3.1   Implementation

To test our solution we created a working prototype, taking advantage of Khepri, an existing implementation of AD, and ContextPy to introduce the COP concepts. Khepri is a portable AD tool, similar to Rosetta [20], that allows the generation of models in different modeling back-ends, such as Rhinoceros or Revit, and offers a wide range of modeling primitives for the supported applications. This tool offers a native implementation in Python, a popular programming language among architects, and also a language with a COP implementation, making it easier to extend.

As for the choice of the COP implementation to use, we decided for ContextPy. This implementation is a library, making it easier to include in existing projects and tools, such as Khepri. Also, since AD programs are usually single-threaded, and we can easily indicate the scope to be affected by the context, Dynamic Layer Activation is a good approach to solve the problem. Finally, the code should be as easy to understand as possible, so an implementation with a *layer-in-class* approach would fit our needs, since all the variations are included in the module they modify. All of this is supported by ContextPy.

Having these two tools, we implemented a new library, that extends Khepri, and introduces design elements with contextual awareness. Since the behavioral variations produce the results on the selected modeling tool, this new layer uses Khepri's primitive functions to produce the results. The functions to use depend on the context. For instance, the program uses those that produce surfaces in analysis contexts, and those that produce solids on 3D contexts. For each new modeling element of our library, we define a class with basic behavior and a variation for each possible context, which are defined as layers in the library as well.

Listing 1 shows a simplified definition of the `Slab` object. This definition has a default behavior, and variations for a 3D, 2D, and analysis contexts, which are identified by the layers in the decorators. In each of these methods, Khepri functions are used, in order to produce the results in the modeling tools.

In the next section, we introduce two case studies used to evaluate our approach.

4   Evaluation

For the evaluation of our COP library we started by using an algorithmic model of a shopping mall originally used for evacuation simulations. The model was produced with an algorithmic solution, which we modified to use our library.

We chose this case study because the original implementation required a plan view of the model, which had to be produced in addition to the usual 3D view. To take advantage of the same algorithm, the original developers provided two versions of the shop function which produces each of the shops available in the mall. One is a 2D version that creates lines, and the other is a 3D version that creates solids. In order to switch between them, a couple lines of code were commented and some variables were changed as well. Listing 2

Listing 1 Definition of a simplified `Slab` object in ContextPy.

```python
class Slab:
    def __init__(self, path, thickness):
        self.path = path
        self.thickness = thickness

    @around(a3DLayer)
    def generate(self):
        return extrusion(surface_from(self.path), self.thickness)

    @around(a2DLayer)
    def generate(self):
        return self.path

    @around(anAnalysisLayer)
    def generate(self):
        return surface_from(self.path)
```

Listing 2 Simplified version of the code with the definition of both shops and the selection of one.

```python
def shop2d(p, v, l, w):
    ...
    rectangle(...)
    line(...)
    ...

def shop3d(p, v, l, w):
    ...
    cuboid(...)
    ...

#shop = shop2d
shop = shop3d
```

shows a simplified definition of both 2D and 3D versions of the shop function, as well as the commented line of code needed to activate the 2D version, and the active line of code that selects the 3D version.

This approach has several disadvantages, namely the need to change the code when it is necessary to change the type of model, and having to comment and uncomment several lines of code when we want to change from 2D to 3D. Both of these tasks are error prone, since developers might forget to do some of them.

Our COP-based solution eliminates the aforementioned problems. Using our library, we re-implemented parts of the algorithm, namely the shop function. Since we have different implementations for the elements, such as walls, available in different contexts, we do not require two versions of the same function. Listing 3 shows a simplified definition of the COP version of the shop function, using the `Wall` and `Door` elements of our library.

■ **Listing 3** Simplified version of the COP version of the `shop` function.

```
def shop(p, v, l, w):
    ...
    w1 = Wall(p1, p2, wall_thickness, wall_height)
    w2 = Wall(p2, p3, wall_thickness, wall_height)
    w3 = Wall(p3, p4, wall_thickness, wall_height)
    w4 = Wall(p4, p1, wall_thickness, wall_height)
    d1 = Door(w4, p5, p6, door_height)
```

■ **Listing 4** Activation of the desired layer.

```
with activelayer(a3DLayer):
    mall(xy(0,0), 100000, 12000, 25000, 7000, 7000, 4)
```
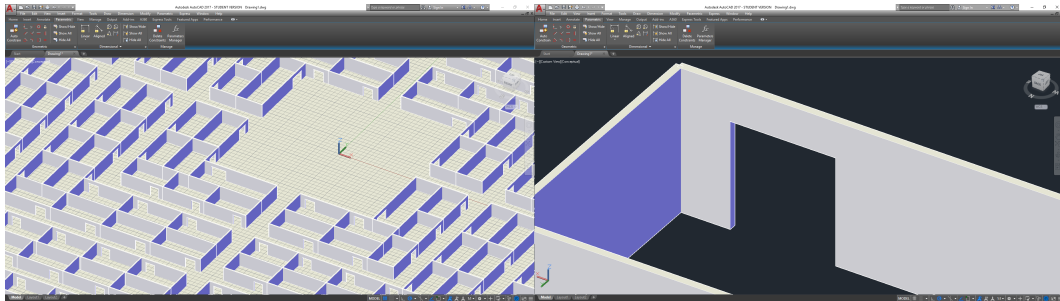

■ **Figure 1** 3D model of the shopping mall, produced with COP in AutoCAD. On the right it is possible to see that the walls are 3D elements.

To shift between contexts, we eliminated the commented lines of code and introduced a `with activelayer` construct, which receives the layer corresponding to the model we want to produce, and the expression that generates the entire shopping mall, as seen in listing 4.

Since we wanted to produce a plan view and a 3D model, and those correspond to layers that we support in the library, we could generate both of them by introducing `a3Dlayer` or `a2Dlayer` as arguments of the `with` construct. The results can be seen in figures 1 and 2. In addition, since we support a layer that produces only surfaces for analysis purposes, namely radiation analysis, we were able to produce another model simply by changing the context. By using `anAnalysisLayer` as argument for the `with` construct, we produced a model for analysis (visible in Figure 3) without any changes to the algorithm.

With our solution, we were able to reduce the code that produces the models and introduce a more flexible way to both change the context and produce different views of the model. This did not required any additional functions, except the `with` construct, as seen in listing 4. In addition, by simply introducing new contexts, our algorithms are capable of producing new models without any changes.

We also re-implemented another case study, which generated a private house. Due to energy requirements, the generated house had to be analyzed regarding solar radiation. Since this analysis only required part of the model and the substitution of all solid elements with surfaces, the original code had several functions that produced each of the required variations, similarly to the previous example.
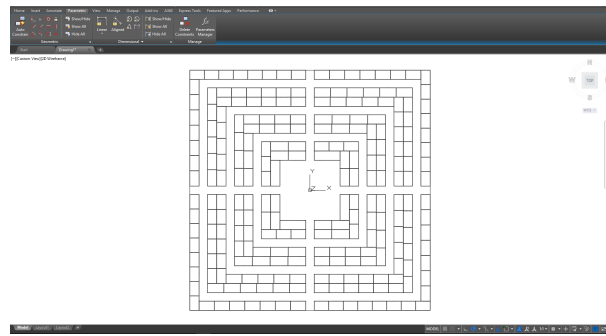
Figure 2 2D model of the shopping mall, produced with COP in AutoCAD.
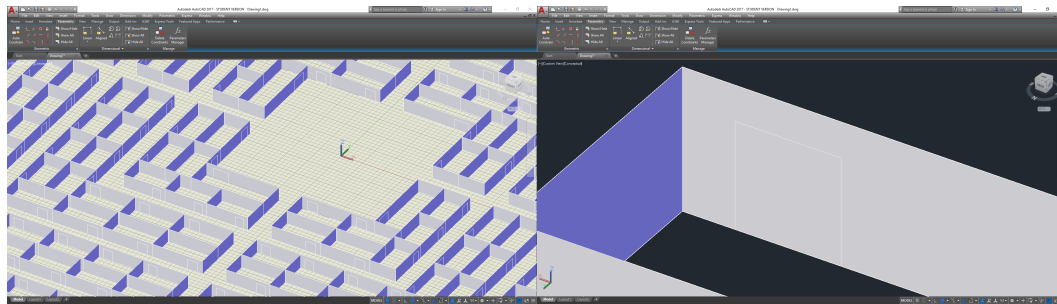
Figure 3 Model of the mall produced for analysis, in AutoCAD. All the walls were replaced with surfaces, as it is possible to see in more detail on the right.

To simplify the algorithmic description of the house, we introduced two new COP layers in the program: (1) the `floorAnalysisLayer`, which only generates the ground floor and replaces elements with surfaces; and (2) the `fullModelLayer`, which produces the complete model in 3D. Both of these layers are used in combination with the layers described in the previous example, in order to define a contextual-dependant definition of the house function. When the house function is used with `fullModelLayer`, the `a3DLayer` is activated and all the elements of the house are produced in 3D, as seen in figure 4. If the `floorAnalysisLayer` is used instead, `anAnalysisLayer` is activated and all the elements are produced as surfaces. In addition, the layer only generates the necessary elements for analysis, as seen in figure 5.

This case study illustrates another advantage of our proposed approach, which is the ability to change layers dynamically during the program execution. This feature offers more flexibility to developers, allowing the production of simplified models for analysis purposes, and more complex ones for 3D visualization. With COP this can be achieved by simply defining and changing layers, instead of using multiple functions with complex conditional statements.

5   Conclusions and Future Work

Currently, algorithmic approaches are used in Architecture to create complex models of buildings that would otherwise be difficult to produce. Moreover, AD also simplifies and automates tasks that were error-prone and time-consuming, and allows an easier exploration of variations. Nevertheless, when architects need to produce different views of the same model, the algorithmic representations must be adapted, creating different versions that increase maintenance efforts.
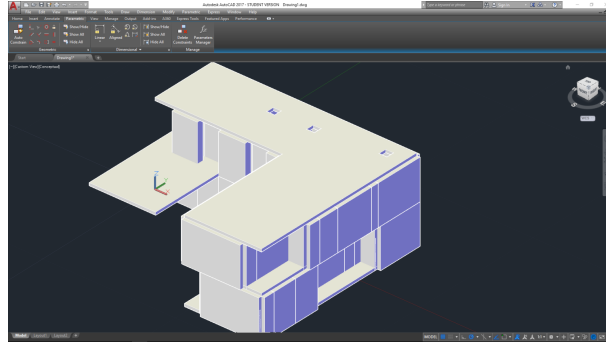
Figure 4 3D model of a house, produced with COP in AutoCAD.
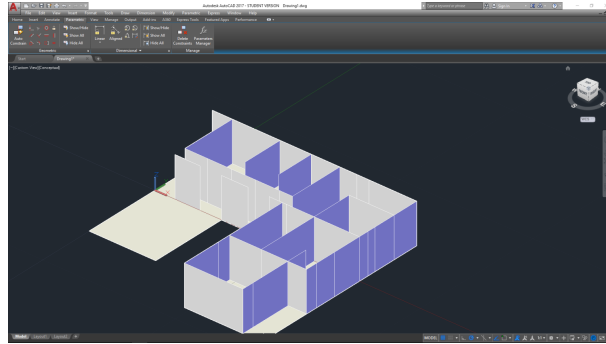
Figure 5 Simplified 3D model of the house for analysis, produced with COP in AutoCAD.

In this paper, we explore the combination of AD with COP, a paradigm that dynamically changes the code's behavior depending on the active context. There are multiple implementations of COP for different programming languages, namely ContextL, ContextJ, and ContextPy, among others, all of which have different features and advantages.

In our solution, we took advantage of ContextPy, a library implementation for the Python programming language, that uses DSA, and a *layer-in-class* approach. All these features fit the needs of AD problems, and the use of Python simplifies the introduction of COP in existing tools, such as Khepri.

To test our solution, we used our COP library in existing AD programs that produced models for different types of analysis. The programs included multiple definitions of the same functions to produce different views of the model, which were activated by commenting and uncommenting code. Both programs were re-implemented with COP, which eliminated the multiple definitions and the commented code. This allowed the production of the models for several different contexts without additional changes in the program.

With these examples, we can conclude that COP can be combined with AD and it can be useful when exploring different views of the models, which require different behaviors from the same program.

As future work, we will continue to expand our library with more building elements. We will also introduce more contexts, giving users more layers for the production of different kinds of models.

In addition, we will explore the combination of layers. One idea is to explore a LOD layer in combination with the others, in order to produce simpler models in an exploration phase, and more complex ones in later stages of development.

Finally, we want to research the methodology that users should follow to use our approach, as well as the features that should be included in a development environment to simplify the use of this approach by non-expert users, such as architects.

───── **References** ─────

1   Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, pages 6:1–6:6, 2009. `doi:10.1145/1562112.1562118`.

2   Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented programming with Java. *Information and Media Technologies*, 6(2):399–419, 2011.

3   Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Software Composition*, volume 6144, pages 50–65, 2010.

4   Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *ACM-SIGPLAN Notices*, 23, 1988.

5   Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextL. In *Symposium on Dynamic languages*, pages 1–10, 2005.

6   Sofia Feist, Guilherme Barreto, Bruno Ferreira, and António Leitão. Portable generative design for building information modelling. In *Living Systems and Micro-Utopias: Towards Continuous Designing - 21st International Conference of the Association for Computer-Aided Architectural Design Research in Asia*, pages 147–156, 2016.

7   Michael L. Gassanenko. Context-oriented programming. In *EuroForth'98 Conference*, 1998.

8   Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-C: Bringing context to mobile platform programming. In *International Conference on Software Language Engineering*, pages 246–265, 2010.

9   Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object system. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008. `doi:10.3217/jucs-014-20-3307`.

10   William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Notices*, 28(10):411–428, 1993. `doi:10.1145/167962.165932`.

11   Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407. Springer, 2008. `doi:10.1007/978-3-540-88643-3_9`.

12   Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

13   Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic contract layers. In *ACM Symposium on Applied Computing*, pages 2169–2175, 2010.

14   Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Designing event-based context transition in context-oriented programming. In *2nd International Workshop on Context-Oriented Programming*, pages 2:1–2:6, 2010. `doi:10.1145/1930021.1930023`.

15   Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

16   Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327–354, 2001.

17   Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

18   Ant]onio Leitão, Renata Castelo Branco, and Carmo Cardoso. Algorithmic-based analysis - design and analysis in a multi back-end generative tool. In *Protocols, Flows, and Glitches: 22nd CAADRIA Conference*, pages 137–147, 2017.

19   Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, 2011.

20   José Lopes and António Leitão. Portable generative design for CAD applications. In *31st Conference of the Association for Computed Aided Design in Architecture*, pages 196–203, 2011.

21   Jon McCormack, Alan Dorin, and Troy Innocent. Generative design: a paradigm for design research. *Proceedings of Futureground, Design Research Society*, 2004.

22   Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. JavaCtx: seamless toolchain integration for context-oriented programming. In *3rd International Workshop on Context-Oriented Programming*, pages 4:1–4:6, 2011. `doi:10.1145/2068736.2068740`.

23   Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012. `doi:10.1016/j.jss.2012.03.024`.

24   Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. ContextErlang: introducing context-oriented programming in the actor model. In *11th International Conference on Aspect-oriented Software Development*, pages 191–202, 2012. `doi:10.1145/2162049.2162072`.

25   Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *ACM Symposium on Applied Computing*, pages 1944–1951, 2009.

26   Gregor Schmidt. ContextR & ContextWiki. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

27   Randall B Smith and David Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996.

28   Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions. In *Software Composition*, volume 6144, pages 66–81, 2010.

29   Ramon van der Heijden, Evan Levelle, and Martin Riese. Parametric building information generation for design and construction. In *Computational Ecologies: Design in the Anthropocene - 35th Annual Conference of the Association for Computer Aided Design in Architecture*, pages 417–429, 2015.

30   Martin Von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM, 2007.

31   Benjamin Hosain Wasty, Amir Semmo, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. ContextLua: dynamic behavioral variations in computer games. In *2nd International Workshop on Context-Oriented Programming*, pages 5:1–5:6. ACM, 2010. `doi:10.1145/1930021.1930026`.

Abcl: Abc music notation with rich chord support

José João Almeida

Departamento de Informática / Centro Algoritmi
Universidade do Minho, Campus de Gualtar, Braga, Portugal
jj@di.uminho.pt
 https://orcid.org/0000-0002-0722-2031

— Abstract —

It is well known the relevance of accompany chords but there is a lack of tools capable of automatically generating sound from them.

In this paper we describe a domain specific language (Abcl) aimed to be a prototyping environment for new experimental music operators. Currently Abcl: (1) adds support for accompany chords (chordmode, instruments, chord-lines); (2) adds clearer support for percussion (drums, drum-machine) (3) adds a support for variables and functions.

Abcl tool is a syntactic-preprocessor that produces Abc. The DSLToolkit, used to create Abcl, is also briefly presented and discussed in the paper.

1 Introduction

Texts with lyrics and chords are often used as an easy way of give musicians the minimal information necessary to accompany a music. Adding (guitar) chords to lyrics is a popular way of helping in the process of learning a music and learning how to play it, how to provide the musical accompaniment for singer or a musical ensemble. Most of the times, documents with chords and lyrics don't provide all the necessary information to be possible to automatically generate sound.

Consider the example in Listing 1. It should be enough for a musician if she already knows the music, but to automatically generate sound, we need to define:
- the measure (3/4);
- the general velocity (1/4=100);
- the duration of each chord;
- what instrument should be used to play the chords and the bass;
- how should the set of nodes of the chord be played during each bar.

This word was developed in the context of the Abc music notation community and we intended to discuss a set of extensions to better support accompany chords and drums. The approach taken was to develop a notation (Abcl DSL) and build a syntactic preprocessor to extend Abc. This way we built an experimental prototype that we can use in our music projects, where syntax can easily changed and discussed. When we obtain a more stable version, we plan to submit a proposal for the next version of abc notation standard.

In this rest of this section we will briefly introduce Abc music notation, and the experimental accompany chord primitives already existent in some Abc processing tools. In

■ **Listing 1** A 20th century *folia* by J.Berthier.

```
T:  Laudate  Dominum
C:  Jacques  Berthier

La[Am]udate  do[E]minum,
La[Am]udate  do[G]minum,
O[C]mnes,  ge[G]ntes,
A[Am]le [F]lu [Dm]ia [E]

La[Am]udate  do[E]minum,
La[Am]udate  do[G]minum,
O[C]mnes,  ge[G]ntes,
A[Am]le [Dm]lu [E]ia [Am]
```

■ **Listing 2** Example of the Abc notation for the score presented in Figure 1.

```
X:101
T:Verbum  caro  factum  est
C:Anonimous,  16th  century
M:3/4
L:1/8
K:G
V:1  clef=treble−1  name="Soprano"  sname="S."
G4 G2 | G4 F2 |A4 A2 | B4 z2|: B3 A GF| E2 D2 EF| G4 F2 | G6 !fine!:|
w: Ver− bum|ca− ro|fac− tum|est |Por − que *|to − dos *|hos sal −|veis
```

Verbum caro factum est (part 1, soprano)

■ **Figure 1** Music score for the Abc in Listing 2.

Section 2 we will present some of the features of Abcl tool and in Section 3 we will discuss the development of the syntactic preprocessor.

1.1 Abc music notation

Abc [13, 7, 4] is used as the base notation throughout all of this paper. The extract presented in Listing 2 illustrates the use of Abc notation, and Figure 1, its corresponding score and MIDI.

Music languages are linguistically different from programming languages. In general music description languages tend to be a merge of several sublanguages like: information fields and metadata, lyrics, tunes (notes and duration), annotations, accompany chords symbols, typesetting geometry. Each sublanguage has a different linguistic flavor.

Naturally, the power of Abc is strongly connected with its processing tools [12, 1, 3, 10] and related projects [5, 2].

Although Abc standard has just a very simple support for chords, the tool Abc2midi[1] has a set of extensions that cover some relevant chord accompany properties:

Figure 2 Music score output for Abcl code from Listing 3.

- `%%MIDI chordprog n`, to define the instrument used for the accompany;
- `%%MIDI bassprog n`, to define the instrument for the bass notes;
- `%%MIDI chorvol n`, to set the volume of the chord notes;
- `%%MIDI bassvol n`, to set the volume of the bass notes;
- `%%MIDI chordname ....`, to define e chord variant;
- `%%MIDI gchordon`, to turn on chord sounds;
- `%%MIDI gchordoff`, to turn off chord sounds;
- `%%MIDI gchord ...`, to define the arpeggiate notes for a bar;
- `%%MIDI gchordbars n`, to express that gchord arpeggiate notes are to be extended for n bars.

The use of these MIDI primitives is cryptic and hard to read and master.

1.2 Accompany Chords

Accompany chords play a very important role in partial music. They are a popular way of starting to learn music but can also be a support for musical improvisation, a way to prepare band rehearsal, to create simple karaoke, and a starting point to several relevant music activities. Several projects discuss [11], teach and *guess* [6] chords.

1.3 In this paper

In this paper we describe a domain specific language [9, 8] – Abcl – that is a syntactic preprocessor that: (1) adds clearer support for accompany chords (chordmode, instruments, chord-lines) (2) adds clearer support for percussion (drums, drum-machine) and (3) adds a reduced support for variables and functions. Abcl internally is written in Perl+DisLex.

As usual, we will call Abcl to the language and `abcl` to the compiler that converts Abcl to Abc.

2 Abcl by example

Example 1

Consider the Abcl example from Listing 3. Executing "`abcl ex.abcl > ex.abc`" we get the output presented in Figure 2.

■ **Listing 3** Abcl example.

```
chordmode 3gui =b2c2ih guitar
chordmode 3gui1=c3 guitar
chordmode 3gui2=ccc guitar

X: 1
T: Laudate Dominum
C: Jacques Berthier
M: 3/4
L: 1/4
Q: 100
K: Am
ch:
|: \3gui { Am|E|Am|G|C|G |1
 \3gui2 Am F Dm | \3gui1 E :|2
 \3gui2 Am Dm E | \3gui1 Am }→\v1
|: \3gui1 \v1 |
W:Laudate' dominu'm,
W:Laudate' dominu'm,
W:Omne's, gente's,
W:Alelu'ia
```

Garota de Ipanema

Antonio Carlos Jobim

■ **Figure 3** Music score generated for "Garota de Ipanema".

In this example, we can see the use *chordmode* instructions to define the arpeggios to be used. We can also see and ear the use of different solutions to build different spaces for improvisation.

The notation { *music* }→\id is used to store *music* in a variable \id, to be reused whenever useful. Abcl also provides other ways of defining variables (see Appendix A) and functions.

Although not much relevant, the generates Abc (file `ex.abc`) is presented in Listing 4.

Example 2

In the following example we discuss some challenges related to Bossa Nova *unpredictable* rich rhythms. In this example we define a set of chords for "Garota de Ipanema". The generated accompany midi (completely unacceptable!) is using the default values of arpeggio, as presented in Figure 3.

In order to improve it a "Bossa Nova beginner's guide" was consulted. Figure 4 is an exercise for students that are learning how to accompany this type of songs. Please notice the always changing rhythm. Please note that this example is not the best choice for "Garota

```
X:  1
T:  Laudate  Dominum
C:  Jacques  Berthier
M:  3/4
Q:  100
K:  Am
 z3 | :
%%MIDI gchordbars 1
%%MIDI gchord b2c2ih
%%MIDI chordprog 24
%%MIDI bassprog 24
  "Am" z3 |
  "E" z3 |
  "Am" z3 |
  "G" z3 |
  "C" z3 |
  "G" z3 | 1
%%MIDI gchordbars 1
%%MIDI gchord ccc
 ... and more 70 similar lines
```


Figure 4 Example of an exercise of "Bossa Nova".

de Ipanem".

In Listing 5 we used two sub-patterns: "`bn1 = first 2 bars ; bn4 = first 4 bars`". Clearly these arpeggios introduce a relevant change in the MIDI, and can be a starting point to discuss the use of guitar/piano in Bossa Nova.

3 Abcl preprocessor tool

As we said before, our project deals with enriching a musical language with new functionality and clearer syntax. We want to be able to prototype experimental operators and syntax. Music tends to be the merge of multi sublanguage with different linguistic flavor. This raised some DSL-building challenges.

In the first experiences, we tried the set of lexical preprocessor. These tools cover preparsing textual substitution (macro), file inclusion and conditionals. Although useful the most popular open-source preprocessor like Gpp (GNU preprocessor) or CPP (C preprocessor) were almost impossible to use without introduce deep changes in the language. Generic lexical preprocessors like m4 also prove to be difficult for the current task.

In addition to lexical-preprocessors functionality, for the current project, we needed:

■ **Listing 5** Sub-pattern usage.

```
chordmode bn1= fczc−bzcz|bzcc−fccz  guitar
chordmode bn2= c  guitar
chordmode bn3= gzcz−zczc  guitar
chordmode bn4= fczc−bzcz|bzcc−fccz|fczc−bzcz|bzcc−fczc  piano

X: 1
T: Garota  de  Ipanema
C: Antonio  Carlos  Jobim
M: 4/4
L: 1/4
K: F
ch: \bn1 Fmaj7|  |G7|  |Gm7|Gb7 |1 Fmaj7|\bn2 Gb7 :|2 Fmaj7|  ||
 \bn3 Gb7|  |B7|  |F#m7|  |D7|  |Gm7|  |Eb7|  |Am7|D7|Gm7|C7 ||
 \bn4 Fmaj7|  |G7|  |Gm7|Gb7|Fmaj7|  ||
```

- State conditions (like the ones presented in Flex) in order to deal with sublanguage heterogeneous syntax.
- Regular-expressions tools: in order textually rewrite new syntax.
- reflexive capabilities (runtime definition of functions).

In this context we choose to build a DSL prototyping toolkit, with a Flex-inspired language processor (DisLex), aimed to support syntactic preprocessors.

DisLex: a Flex-inspired language processor

DisLex is a Flex-flavored lexical analyzer for Perl. It is part of `Parse::DSLUtils`, a Perl module aimed to help in the construction of DSL. In complement, `Parse::DSLUtils` also covers `Parse::Yapp` simplification, and templates functionality.

Following some relevant features of DisLex tool:

- By default, input is slurped to a variable (`$yyfile`).
- Regular-expression based using `\G` and `pos($yyfile)` to keep current position in a efficient way.
- (`RegExp, Perl-action`) rules are the basic building blocks. Perl's regular-expressions offers a very rich group capture functionality that proved to be very effective. In flex group capture is not available.
- Full Unicode expressions available
- Greedy and non-greedy regular-expressions operators available.
- No support for chooser-longest-match disambiguation rule.
- State conditions: to help in implementation of automata, using:
  - `BEGIN state` to change states
  - `REC state` and `DONE` to change and came back (similar to Flex `yy_push_state(state)`, `yy_pop_state()` functions)
- It includes a large set of predefined regular-expressions, covering some non-regular patterns like:
  - curly-bracket blocks, XML elements, Latex environments

When used as a lexical analyzer, DisLex typically, uses rules like

```
(\d+) return("INT",$1))
```

It provides directives (syntactic sugar) to skip white spaces and comments:

```
%white [\ \t]+
%comments #.+
```

Some less common functionality:
- Return many – sometimes is easier to return several tokens (using a queue of symbols to be returned)
  ```
  \+= { returnmany(['=','='],["INT",1],['+','+']);}
  ```
- a predefined `yylexdebug(func,file)` to help testing and debugging lexical analyzer's behavior.

A set of predefined functions is provided to cover some simplified cpp-like functionality (includes, defines).

4 Conclusions

Although in an initial stage, from our experience, Abcl tool proves to be useful for:
- providing support for modeling arpeggios for specific styles of music;
- practice improvisation supported by neutral chord bases;
- experimentation on accompany solutions.

The use DisLex and Parse::DSLUtils were crucial to obtain a working prototype in a very short time.

We are currently working with Abcl with multi-voice chords, and multi-voice drums and we already have interesting examples of the use of chords and percussion using Abcl DSL language.

References

1 James Allwright and Seymour Shlien. abc2midi: Abc to midi translator. `http://abc.sourceforge.net/abcMIDI/`. Tool.
2 José João Almeida, Nuno Ramos Carvalho, and José Nuno Oliveira. Wiki::Score a collaborative environment for music transcription and publishing. *Information, Services and Use (ISU)*, 31(3-4/2011):177–187, 2012. `doi:10.3233/ISU-2012-0647`.
3 Bruno M. Azevedo and José João Almeida. Abc with a unix flavor. In *2nd Symposium on Languages, Applications and Technologies (SLATE)*, volume 29, pages 203–218, 2013. `doi:10.4230/OASIcs.SLATE.2013.203`.
4 Abc Comunity. Abc musical notation – standard version 2.2, 2013. Standard. URL: `http://abcnotation.com/wiki/abc:standard:v2.2/`.
5 Michael Scott Cuthbert and Ben Houge. Music21 - a toolkit for computer-aided musicology. `http://web.mit.edu/music21/`. Toolkit's Homepage.
6 W. Bas de Haas, José Pedro Magalhães, and Frans Wiering. Improving audio chord transcription by exploiting harmonic and metric knowledge. In *13th International Society for Music Information Retrieval Conference (ISMIR)*, 2012.
7 Guido Gonzato. Making music with abc2 - a pratical guide, 2018. URL: `http://abcplus.sourceforge.net/abcplus_en.html`.
8 Tomaž Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: a systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

9   Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda Pereira, Matej Čre-
        pinšek, Daniela da Cruz, and Pedro Rangel Henriques. Comparing general-purpose and
        domain-specific languages: An empirical study. *Computer Science and Information Sys-*
        tems, 7(2):247–264, 2010. `doi:10.2298/CSIS1002247K`.
10 Nils Liberg et al. EasyABC abc editor, 2016. URL: `http://easyabc.sourceforge.net/`.
11 José Pedro Magalhães and Hendrik Vincent Koops. Functional generation of harmony
        and melody. In *2nd ACM SIGPLAN International Workshop on Functional Art, Music,*
        Modeling and Design, pages 11–21, 2014. `doi:10.1145/2633638.2633645`.
12 Jean-François Moine. abcm2ps - abc to postscript/eps/svg translator. `http://moinejf.`
        `free.fr/`. Tool.
13 Chris Walshaw. Abc notation. `http://abcnotation.com/`. Musical Notation.

A   Cannon

Example of a very simple use of tune variables: the structure of a 2 voice cannon (Frere
Jacques).

```
\p1={CDEC  |  CDEC}
\p2={EFG2  |  EFG2  |  G/2A/2 G/2F/2E C  |  G/2A/2 G/2F/2E C}
\p3={DG,C2  |  DG,C2}

X:1
T:  Frere  Jacques  (2  voice)
M:  4/4
L:  1/4
K:  C
[V:1]  \p1  |:  \p2  |  \p3  |1 \p1  :|2  Z2  |]
[V:2]  Z2  |:  \p1  |  \p2  |1 \p3  :|2  \p3  |]
```

Frere Jacques (2 voice)

Asura: A Game-Based Assessment Environment for Mooshak

José Carlos Paiva

CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal
up201200272@fc.up.pt
https://orcid.org/0000-0003-0394-0527

José Paulo Leal

CRACS & INESC-Porto LA, Faculty of Sciences, University of Porto, Portugal
zp@dcc.fc.up.pt
https://orcid.org/0000-0002-8409-0300

Abstract

Learning to program is hard. Students need to remain motivated to keep practicing and to overcome their difficulties. Several approaches have been proposed to foster students' motivation. As most people enjoy playing games of some kind and play on a regular basis, the use of games is one of the most widely spread approaches. However, taking full advantage of games to teach specific concepts of programming requires much effort. This paper presents Asura, a game-based assessment environment built on top of Mooshak that challenges students to code Software Agents (SAs) to play a game, allowing them to test the SAs against each others' SAs and watch a movie of the test. Once the challenge development stage ends, teachers are able to organize game-like tournaments among SAs. One of the key features of Asura is that it provides a means to reduce the required effort of building game-based challenges up to that of creating traditional programming exercises.

1 Introduction

In the past recent years, the demand for programmers in the job market has grown rapidly, raising the popularity of programming courses among future undergraduate students who seek for a rewarding career [16]. However, learning to program is hard. Introductory programming courses are considered difficult by many students [4] and often linked to high dropout and failure rates [1]. Many educators consider the lack of abstraction and problem-solving skills as the main sources of difficulties for novice programmers [7]. Nevertheless, these difficulties

can be mitigated if students have enough motivation to keep practicing and struggling to overcome them [10].

Consequently, several approaches to foster the engagement of students in programming activities, such as problem-based learning, storytelling, simulations, and competition-based learning, have been proposed and evaluated. Yet, one of the most successful and widespread approaches is gamification, which consists of using game elements and mechanics to engage users in non-game contexts. The most common methods of gamification just give rewards (e.g., badges, experience points, or gifts) to students when they succeed or complete a task. However, rewards are generally weak motivators and its use is controversial, since many argue that they can harm intrinsic motivation and arise long-term educational issues [11]. Other game aspects such as graphical feedback, competitive challenges, goals, collaboration, or abstraction of the physical world, are, typically, much more attractive than rewards. In this sense, some proposals replace the learning activity completely by a game, as is the case of CodeRally[1] and RoboCode [6] which have demonstrated the success of this approach. Unfortunately, the implementation of complete games to teach a specific concept is very costly in terms of time and money.

This paper presents Asura, a game-based assessment environment built on top of Mooshak [9], a system for managing programming contests on the web. Specifically, it integrates into the computer science languages learning environment of Mooshak 2.0, named Enki [13], providing game-based programming challenges to any Enki course. Each challenge asks the student to code a Software Agent (SA) to play a game, requiring the student to understand the rules and goals of the game. After the "break the ice phase", students are challenged to improve their SAs insomuch that they defeat every opponent. While performing this task, students can take advantage of facing any of the existing opponents in a private match, and watch how the match unfolds in a game-like movie. Once the time to solve the challenge ends, educators can organize tournaments, similar to those found on traditional games and sports, among SAs. The tournament is displayed in an interactive GUI, allowing the viewers to control which games they want to see, navigate through stages, and check the ranking. Furthermore, one of the key goals of Asura is to enable teachers to build games with a similar complexity to that of creating an ICPC-like problem. These games can be very simple, such as a number guessing game, or more complex than CodeRally or Robocode. For that, Asura provides a set of helpers including a Java framework and a Command-Line Interface (CLI) tool to support the authoring of challenges.

The remainder of this paper is organized as follows. Section 2 reviews the systems and tools in which Asura lies on as well as environments with characteristics common to Asura. Section 3 describes Asura, its architecture and components. Section 4 provides the guidelines of the experiment that will be conducted to validate Asura. Finally, Section 5 summarizes the contributions of this paper, the expected results of the validation, and the next steps of this work.

2 State-of-the-Art

Asura is a game-based assessment environment designed to integrate into an already existing computer science languages learning environment of Mooshak 2.0, named Enki. The environment aims to offer a way to motivate students to program and overcome their difficulties through practice, requiring from teachers an effort similar to that of creating an ICPC-like

---

[1] `https://www.ibm.com/developerworks/mydeveloperworks/blogs/code-rally`

problem. It engages students by challenging them to code an SA to play a game, supporting them with graphical game-like feedback to visualize how the SA performs against other SAs. The final goal of an Asura challenge is to win a tournament, like those found on traditional games and sports, among all submitted SAs.

There are already tools in the literature providing some of these features separately. For instance, competition is not a new paradigm in programming learning [2, 8]. Students are increasingly facing programming contests after leaving universities as a part of the recruitment process for top technology companies. So, providing a learning experience with focus on competition may help them in the future. One of the first types of systems to foster competition among programming learners were automatic judges, such as DOMJudge[2], PKU JudgeOnline[3], and Mooshak [9]. Even if these systems were developed for international and regional competitions, teachers of undergraduate programming courses found them useful also as teaching assistant tools [5, 3] to promote competitive programming learning environments and give instant feedback on laboratory classes and exams. In particular, the increasing interest shown on using Mooshak for learning has motivated the development of several extensions, such as quiz evaluation and exam policies.

Recently, Mooshak was completely reimplemented in Java with Graphic User Interfaces (GUIs) using the Google Web Toolkit (GWT). This reimplementation was labeled as Mooshak 2.0[4]. Beyond the changes in the code-base, Mooshak 2.0 gives special attention to computer science learning. For instance, it now includes a specialized computer science languages learning environment – Enki [13] – , which not only supports exercises using typical programming languages, but also diagramming exercises, quizzes, among others. Enki blends assessment and learning, integrating with several external tools, such as a gamification service – Odin [12] – to support the creation of leaderboards, reward students for their successes, among others, and an educational resources sequencing service – Seqins [14] – to offer different learning paths according to the skills of each student.

Currently, there are many other web-based learning platforms that focus on competition to motivate learners, such as HackerRank[5], in which competition is based on leaderboards that consider the number of solved problems and time spent solving them, and CodeFights[6], in which a player "fights" against other player or a bot developed by a company, to complete a set of challenges before the opponent.

Also, game-based approaches for teaching programming have already been applied in several cases. Rajaravivarma [15] proposes a course with a set of challenges, in which the students have to program their own games. Sui et al. [17] presents a browser-based environment that combines gamification and peer-to-peer interaction. This environment challenges the students to write SAs to play simple board games, allowing them to test these SAs against SAs developed by other peers. The environment also provides a debugger interface, which allows the student to check the state of the SA step-by-step.

There are also a few online platforms using games as the central source of engagement, such as CodinGame[7] and Leek Wars[8]. CodinGame proposes several puzzles for learners to practice their coding skills. Most of them require the user to develop an SA to control the

---

[2] https://www.domjudge.org/
[3] http://poj.org/
[4] https://mooshak2.dcc.fc.up.pt/
[5] https://hackerrank.com/
[6] https://codefights.com/
[7] https://codingame.com
[8] https://leekwars.com
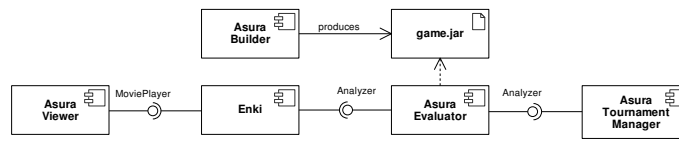
■ **Figure 1** Diagram of components of the architecture of Asura.

behavior of a character in a game environment, and provide a 2D game-like graphical feedback. Leek Wars is a serious game where the player controls a character (a leek) through coding. The goal of the game is to beat other players' leeks during fights, awarding points to climb to the top of the ranking. The result of the fight is displayed as a 2D frame-by-frame movie.

3   Asura

Asura is an environment for game-based assessment in programming learning. Its main features include the graphical game-like feedback, the tournament-based assessment, and the framework and tools that it offers regarding the simplification of the process of building Asura challenges. This environment leverages the SAs evaluation on Mooshak 2.0's program analysis, extending it to support multiple submissions running in the same evaluation environment. This allows students to code their SAs in any programming language supported in Mooshak. On the client side, the Graphic User Interface (GUI) is embedded into Enki's GUI, allowing any course created in Enki to provide Asura challenges out-of-the-box.

The architecture of Asura, presented in Figure 1, is composed of four components: Asura Builder, Asura Viewer, Asura Tournament Manager and Asura Evaluator. The Builder, Viewer, and Tournament Manager are completely new components, whereas the Evaluator is a component that extends Mooshak 2.0 program analysis to support game assessment. In this architecture, Enki makes the bridge between the server and the client, requesting evaluation to the Evaluator and loading the feedback into the Viewer. Each of the next subsections describes a component of Asura, including its role and some implementation details.

3.1   Evaluator

The evaluator engine of Mooshak grades a submission by following a set of rules while generating a report of the evaluation for further validation from a human judge. This evaluation follows a black-box approach. The process consists of two types of analysis: static, which checks for integrity of the source code of the program and produces an executable program; and dynamic, that involves the execution of the program with each test case loaded with the problem.

The Evaluator component of Asura inherits the static analysis of the Mooshak's evaluator engine. The only difference is that the compile command line can include a language-specific player wrapper, present in the `game.jar` file, for complex games. However, the dynamic analysis is completely reimplemented. Instead of test cases with input and output text files, Asura Evaluator receives as input a list of paths of the selected opponents' submissions that generate the input and consume the output of each other. Since these submissions are already compiled (when necessary), the component just initializes a process for each of them. After that, it organizes matches containing the current submission and a distinct set of the selected opponents' submissions. The length of this set depends on the minimum and maximum number of players per match, which are specified by the game manager. At this point, the evaluation proceeds on an instance of the specific game manager, which is instantiated from

the `game.jar`, as well as the game state object. The game manager receives the list of player processes indexed by the player ID and starts the game. The execution of the game is completely controlled by the game manager, which is responsible for keeping the SA's informed about the state of the game, querying the SAs for their state update at the right time, ensuring that the game rules are not broken, managing the state of the game, and classifying and grading submissions. If an SA fails to comply with the rules, the match ends and the SA receives a "Wrong Answer".

During the game, any updates to the state object can passed on to the movie builder. The state object provides the following methods that can be used by the manager to control its lifecycle: `prepare(movieBuilder, players)` which initializes the state and sets up the metadata of the movie, `execute(movieBuilder, playerId, playerUpdate)` that updates the game state with the action of a certain player, `endRound(movieBuilder)` which ends a round of actions, and `finalize(movieBuilder)` which finalizes the game, adding the submission results in the movie, among other things.

Finally, the status obtained from the matches containing the observations, mark, classification and feedback are compiled into a single status which is added to the submission report, and sent to the client.

3.2 Builder

The Builder component is a Java framework for building Asura challenges, providing a game movie builder, a general game manager, several utilities to exchange complex state objects between the manager and the SAs, and general wrappers for players in many different programming languages. The framework is accompanied by a Command-Line Interface (CLI) tool to easily generate Asura challenges and install specific features, such as support for a particular programming language, a default turn-based game manager, among others.

Much of the necessary effort for building video games is spent on graphics. To simplify this task, Asura introduces the concept of game movie. A game movie consists of a set of frames, each of them containing a set of sprites together with information about their location and applied transformations, and metadata information, such as `title`, `background`, `width`, `height`, `fps`, and `players` (i.e., player names indexed by their ID). Furthermore, the concept of game movie is formally defined in a JSON schema[9]. To allow the manager to easily build the JSON data, the game movie builder provides several methods, such as: setters for each metadata field, `addFrame()` which adds a frame to the movie, `addItem(sprite, x, y, rotate, scale)` which adds a sprite to the current frame in position (x, y) with the given rotation and scale, `addMessage(playerId, message)` which adds a message to the player, setters for observations and classification, and `saveFrame()`/`restoreFrame()` which allow to add a frame state to a stack to restore it later.

The abstract game manager provided by the Builder defines the "contract" of the managers, and provides the necessary utilities for dealing with input/output streams of the processes. Specialized managers must implement the method `manage(state, players)`, determining the order to play, and managing the state of the game accordingly. Some of these specialized managers, such as a turn-based game manager, are already developed and can be easily used in a challenge.

The exchange of state updates between the SAs and the manager is done through JSON. Depending on the programming language, this can be a hurdle and make it very complex

---

[9] `https://mooshak2.dcc.fc.up.pt/asura/static/match.schema.json`
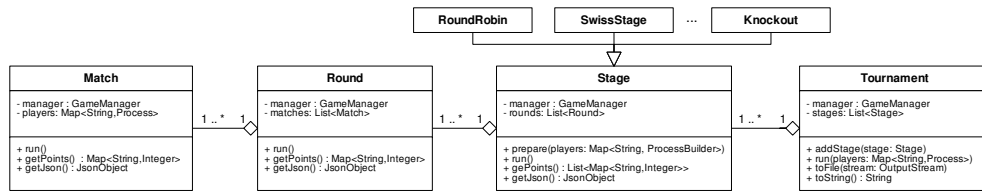
Figure 2 UML class diagram of the Tournament Manager.

for SA's to process it. For that, there are wrappers for players providing several methods to process JSON. Moreover, each game can define its own wrappers providing methods specific to the game, which can be used by SAs.

The documentation for the very first release of Asura Builder is available online[10] and has already been followed by some peers who volunteered to test the system.

3.3 Tournament Manager

The Tournament Manager is the component responsible for managing and running tournaments. A tournament can have any number of stages, each with its own type (pools or knockout) and format (e.g., round-robin, swiss type, knockout, double knockout, among others). Stages are populated with a set of players, those who qualified in the previous stage, which will compose the matches of every round of the current stage. The assignment of players to matches is a task of the class implementing the specific tournament format. These tournaments can be organized in the administrator GUI of Mooshak 2.0, through a wizard developed specifically for this task. This wizard allows to select players individually, add/remove stages, and set some properties of the tournament, such as the number of players per match, the number of qualified players in each stage, and the points awarded to a win/draw/loss.

Figure 2 presents the UML class diagram of the implementation of the Tournament Manager. This implementation contains a class for the tournament as well as for each of its phases: stages, rounds, and matches. These classes have very similar methods, such as `run()` which executes the phase, and `getPoints()` which returns either a list of players' points, in group stages, or the players' points. Each match of the tournament is executed in the Evaluator component.

3.4 Viewer

Asura Viewer is a GWT widget with two modes: match and tournament. Figure 3 presents both modes side-by-side. In the tournament mode, it expects a JSON file following the Tournament JSON Schema[11]. This data contains a reference to each match's movie, organized by stages and rounds, as well as partial and complete rankings of each phase. The tournament mode widget consists of an interactive GUI, allowing students to navigate through the stages of a tournament, visualize specific matches or the whole course of a player, and check the rankings of each stage.

The match mode displays the game movie JSON produced during the evaluation phase. The GUI mimics that of a media player, containing a slider, a play/stop button, buttons to

---
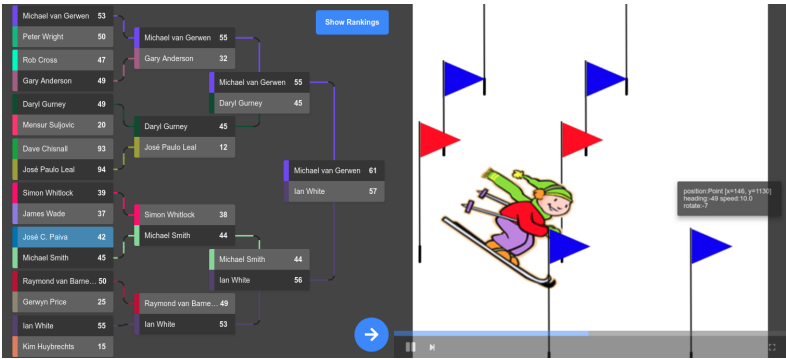
Figure 3 Asura Viewer display modes. Tournament mode (on the left) displays a knockout stage of a tournament. Match mode (on the right) presents the graphical feedback provided to a slalom skier.
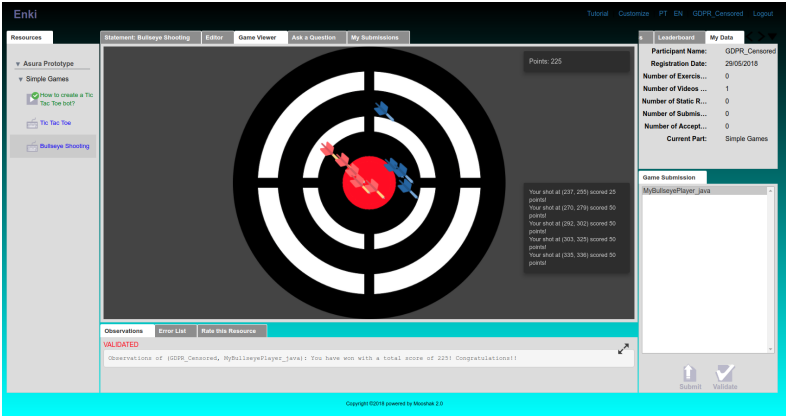
Figure 4 Asura Viewer integrated in the GUI of Enki after the validation of the SA.

navigate through the current play-list, a full-screen button, a box to display messages, and a canvas where the movie is drawn. The movie is completely resizeable keeping the original aspect ratio.

This component can be embedded in any environment, provided that the JSON data passed to it adheres to the defined JSON schemas for tournaments and matches. Figure 4 presents a screenshot of the Asura Viewer integrated in the GUI of Enki, after a player validated its SA against an existing SA.

4 Experiment Guidelines

An experiment with Asura will be conducted in the laboratory classes of an undergraduate Object-Oriented Programming (OOP) course at *Escola Superior de Media Artes e Design* (ESMAD) – a school of the Polytechnic Institute of Porto. The purpose of this course is to introduce Javascript and some OOP concepts integrated into its ECMAScript 6 version to students with little to none programming background. The course is composed of two distinct laboratory classes which will be labeled hereafter as Control Group (CG) and Experimental Group (EG), respectively. The homogeneity of the groups will be checked using students' Grade Point Average (GPA). Both groups will have access to an online course in Enki containing a set of programming activities to assimilate knowledge obtained

during expository classes. However, CG will have to solve traditional programming exercises, whereas EG will have to develop players for Asura games. These activities will be created by a group of three teachers, pertaining to the same institution as the students, with a single requirement that both sets of activities have identical difficulty. The course will be available online during two weeks, in which students are free to access in and after class.

At the end of the experiment, students and teachers will be asked to fill in online questionnaires about the usefulness of the learning environment. Teachers will have specific questions regarding the difficulty of developing problems for Asura. Students will have questions to assess the level of engagement obtained while using the environment. The results obtained from the online questionnaires as well as usage data collected during the preparation and execution of the course will be analyzed in order to draw conclusions about the effectiveness of Asura. This usage data includes a number of variables, such as the number of solved exercises, the number of submissions per problem, the time spent per exercise, and the number of different strategies used per problem (identified by a threshold on the number of different characters).

5   Conclusions and Future Work

This paper presents Asura, an environment that aims to provide engaging game-based activities with graphical game-like feedback as well as to facilitate the creation of those activities by teachers. From the students' perspective, the main idea of Asura is to challenge students to code an SA that plays a game, taking advantage of the graphical feedback on its performance against other SAs. Once the SA development period ends, teachers can organize tournament amongst SAs. With regard to teachers, Asura provides a Java framework that supports the creation of game-based challenges, particularly in the process of building the game movie.

Asura is a work in progress. It is currently in the final development stage, just lacking the implementation of a few types of tournament stages and some minor improvements in the Builder. The design of each component of Asura, including the way of integrating them with the existing work, is already done. Several game-based challenges were already developed using the framework provided by the Builder.

The next phase encompasses the execution of the experiment described in Section 4. It is expected that students in EG will spend considerably more time in activities, trying different strategies to beat up their colleagues, which would indicate greater engagement. However, some students may be negatively affected by losing and show their displeasure in the questionnaire's responses. Students in the CG will not spend more time in the environment than the necessary amount to solve all problems once. Also, some exercises may not be solved by the end of the experiment. In respect to teachers, they will notice a small increase in difficulty while creating the games, since they need to be familiar with Java and to understand the framework beforehand. The amount of time that they spend doing the graphics is also unpredictable since it highly depends on the quality that they want to reach. Nevertheless, this should not result in a significant difference in terms of time, when comparing with the time of setting up an ICPC-like problem, if they already know Java.

One of the major points of improvement already identified is in the Asura Builder. Many teachers are not familiar with Java, but the framework requires its use. This will certainly prevent or make it very difficult for these teachers to work with Asura. For this reason, the Builder component will be extended to support language-agnostic creation of games.

References

1 Jens Bennedsen and Michael E. Caspersen. Failure rates in introductory programming. *SIGCSE Bulletin*, 39(2):32–36, 2007. `doi:10.1145/1272848.1272879`.

2 Juan C. Burguillo. Using game theory and competition-based learning to stimulate student motivation and performance. *Computers & Education*, 55(2):566–575, 2010. `doi:10.1016/j.compedu.2010.02.018`.

3 Ginés Gárcia-Mateos and José Luis Fernández-Alemán. A course on algorithms and data structures using on-line judging. *SIGCSE Bulletin*, 41(3):45–49, 2009. `doi:10.1145/1562877.1562897`.

4 Anabela Gomes and António José Mendes. Learning to program-difficulties and solutions. In *International Conference on Engineering Education (ICEE)*, 2007.

5 Pedro Guerreiro and Katerina Georgouli. Enhancing elementary programming courses using e-learning with a competitive attitude. *International Journal of Internet Education*, 10(1):27–42, 2008.

6 Ken Hartness. Robocode: Using games to teach artificial intelligence. *Journal of Computing Sciences in Colleges*, 19(4):287–291, 2004.

7 Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *SIGCSE Bulletin*, 37(3):14–18, 2005. `doi:10.1145/1151954.1067453`.

8 Ramon Lawrence. Teaching data structures using competitive games. *IEEE Transactions on Education*, 47(4):459–466, 2004. `doi:10.1109/te.2004.825053`.

9 José Paulo Leal and Fernando Silva. Mooshak: A web-based multi-site programming contest system. *Software: Practice and Experience*, 33(6):567–581, 2003. `doi:10.1002/spe.522`.

10 Scheila Wesley Martins, António José Mendes, and António Dias Figueiredo. A strategy to improve student's motivation levels in programming courses. In *Frontiers in Education Conference*, pages F4F–1–F4F–7, 2010. `doi:10.1109/FIE.2010.5673366`.

11 Wilbert J. McKeachie. The rewards of teaching. *New Directions for Teaching and Learning*, 1982(10):7–13, 1982. `doi:10.1002/tl.37219821003`.

12 José Carlos Paiva, José Paulo Leal, and Ricardo Queirós. Odin: A service for gamification of learning activities. In *International Symposium on Languages, Applications and Technologies*, pages 194–204, 2015. `doi:10.1007/978-3-319-27653-3_19`.

13 José Carlos Paiva, José Paulo Leal, and Ricardo Alexandre Queirós. Enki: A pedagogical services aggregator for learning programming languages. In *Conference on Innovation and Technology in Computer Science Education*, pages 332–337, 2016. `doi:10.1145/2899415.2899441`.

14 Ricardo Queirós, Paulo José Leal, and José Campos. Sequencing educational resources with Seqins. *Computer Science and Information Systems*, 11(4):1479–1497, 2014. `doi:10.2298/csis131005074q`.

15 Rathika Rajaravivarma. A games-based approach for teaching the introductory programming course. *SIGCSE Bulletin*, 37(4):98–102, 2005. `doi:10.1145/1113847.1113886`.

16 Anthony Robins, Janet Rountree, and Nathan Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172, 2003. `doi:10.1076/csed.13.2.137.14200`.

17 Li Sui, Jens Dietrich, Eva Heinrich, and Manfred Meyer. A web-based environment for introductory programming based on a bi-directional layered notional machine. In *Conference on Innovation and Technology in Computer Science Education*, pages 364–364, 2016. `doi:10.1145/2899415.2925487`.

CaVa$^{\text{DSL}}$: Virtual Learning Spaces Formal Specification

Ricardo Giuliani Martini

Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
rgm@algoritmi.uminho.pt
 https://orcid.org/0000-0001-5217-6110

Pedro Rangel Henriques

Centro Algoritmi / Departamento. de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
prh@di.uminho.pt
 https://orcid.org/0000-0002-3208-0207

Abstract

In the context of Cultural Heritage, *memory institutions* build exhibition rooms to expose their assets and disseminate knowledge through these learning spaces. CaVa project aims at facilitating the process of learning spaces construction on the web to implement virtual museums. This paper presents CaVa$^{\text{DSL}}$, an external Domain-Specific Language (DSL) designed to specify virtual Learning Spaces enabling their automatic generation. To introduce CaVa$^{\text{DSL}}$ language, the paper runs a case study, *Museu Virtual Interativo da Fotografia* (MVIF), presenting the specification for the museum's exhibiting rooms and their final layout. The process that analyzes and transforms the formal specification into the virtual Learning Spaces is briefly described to present the core engine of CaVa platform.

1 Introduction

To preserve objects that belong to the assets of *memory institutions*, like museums or archives, and to disseminate the enclosed knowledge (the so-called *cultural heritage*), people is resorting more and more to the digital format and computer-based systems. In particular, Internet is an appealing vehicle for imparting knowledge; the related technology has changed the manner of reading, thinking, writing, and learning [1]. According to [6], the rapid and continuing advances in information and communication technologies are changing the ways people share, use, develop and process information. In this context a new concept emerged years ago and is gaining popularity: *virtual museum* [3]; CaVa project, that is discussed along this paper, was born to automatically create those museums. Associated with it, we decided to call *virtual Learning Space* (vLS) the website containing the information related with the museum belongings (its digital objects) that will be seen and explored by the visitor so that he can

learn through it. These new web Learning Spaces are the *exhibition rooms* of the traditional (physical) museums [5].

This paper presents an approach to automatically build these virtual Learning Spaces formally specified in a specially tailored language (CaVa^{DSL}) that resorts to a domain ontology. That Domain Specific Language was designed aiming at being easily used by museum Curators and other non-programmers, experts in Cultural Heritage areas – so, as we will illustrate with a running-example along the paper, CaVa^{DSL} has a light syntax and is focussed on the process of planning and mounting museum exhibits. The ontology schema[1], at the core of CaVa system and underlying approach, is defined with two purposes: first, to give formal semantics to the digital object repository, using an high-level and abstract schema independent of the final use; second, to describe the information that must be displayed in the envisaged virtual Learning Space.

Our main goal in this paper is to discussed as a properly designed language – based on an appropriate vocabulary (defined for a specific knowledge domain) – can effectively facilite the mission of people working in *memory institutions*. From specifications written according to our ontological approach in CaVa^{DSL} language, the system we developed, CaVa [2, 4, 5], is able to automate the creation of web-based virtual exhibition rooms (vLS) to display cultural, material or immaterial, objects.

This paper is organized as follows. The next Section 2 introduces our case study: *Museu Virtual Interativo da Fotografia* (MVIF for short). Section 3, the main one, introduces CaVa^{DSL} language, and Subsection 3.1 continues its presentation outlining the specification of MVIF virtual LS with CaVa^{DSL}. In order to process the formal specification written in CaVa^{DSL} and generate the MVIF exhibition rooms, Section 4 describes CaVa^{gen}, a set of web-application generators. Section 5 presents the result of rendering the files generated by CaVa^{gen}, the desired MVIF. Finally, Section 6 concludes and gives the directions for future work.

2   MVIF at a glance

MVIF is a virtual museum developed by Ricardo Ravanello. The museum exposes the techniques, equipments, ideas and characters of the history of photography [8]. In the work underlying this paper, the idea is not to reproduce the museum faithfully [2], but just to use MVIF specification and implementation to illustrate our proposal presenting a concrete example of a CaVa^{DSL} museum description. To store the MVIF assets, a relational database was built. Some tables of that database can be seen at the MVIF website[3] clicking on the left side menu "MVIF DB TABLES".

To reproduce the MVIF museum in CaVa, it is necessary to describe the context domain in which it is enclosed. For that and according to the CaVa ontological basis, we designed an ontology describing the MVIF concepts and relations; a part of it will be presented. Figure 1 shows the two main concepts related to the MVIF domain, `Technique` and `Period`, and four subclasses of the later. Notice that those two concepts abstract precisely the data enclosed in the two tables shown in the museum's link referred above). Four attributes (datatype properties) for the concept `Period` are also displayed.

---

[1] Ontology defined at the abstract level of concepts and relations, without instances.
[2] Actually the museum's web site available is the original one, by Ravanello, and not a web site generated by CaVa environment.
[3] Available at `http://www4.di.uminho.pt/~gepl/cava/MVIF/`
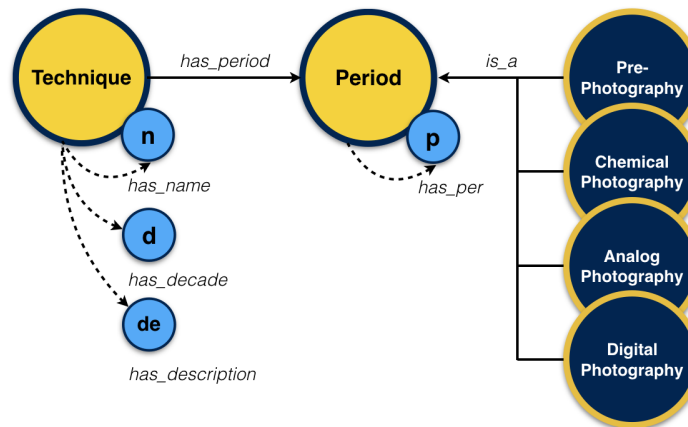
■ **Figure 1** A portion of the "Museu Virtual Interativo da Fotografia" ontology.

To relate the abstract concepts of the MVIF ontology with their instances stored in the relational database structure, it is necessary a *mapping* between the two – that mapping will create a pavement to enable querying the database using the abstract vocabulary established by the ontology. For this purpose, the Ontop[4] framework based on the Ontology-Based Data Access (OBDA) axioms was used. The mapping for this concrete case is available at the MVIF website on the left side menu "MVIF MAPPING".

3 CaVa^DSL Language

Aiming at an easy generation of virtual Learning Spaces for the use of the person in charge of archives or museums, an external DSL was designed. CaVa^DSL was developed to describe, in an abstract level, virtual exhibition rooms in the museum Curator's perspective, giving the Curator the possibility to specify the virtual LS based on a domain ontology vocabulary. CaVa^DSL focuses on the exhibition rooms, so the main component of the language is a list of exhibitions. The syntax of CaVa^DSL is similar to that of the JSON (JavaScript Object Notation) language, because it is based on the 'key-value pair' format and it is easy for humans to read and write. The structure of CaVa^DSL is split into four main blocks that specify: the main configuration, the header, the content, and the footer of the virtual Learning Space, as specified by the derivation rule `p0`.

```
[p0] cava:  mainConfig header content footer ;
```

The details of that language will be presented in the next subsection using MVIF as a running example.

3.1 MVIF specified in CaVa^DSL

This section presents the specification of the MVIF museum organized according to the four main blocks of CaVa^DSL.

The *mainconfig* defines the virtual Learning Space title and main description, as well as, other components (e.g., carousel of images) related to the entire LS (not only about a specific page, like an exhibition, for example); this configuration is written according to the production rule `p1` of Listing 1:

Listing 1 Production rule p1: mainConfig.

```
1  [p1] mainConfig:  'mainconfig' '[' learningSpaceTitle learningSpaceAbout?
       learningSpaceCarousel? ']' ;
```

Listing 2 Example applying the production rule p1: mainConfig.

```
1   mainconfig [
2       LS title: "Museu Virtual Interativo da Fotografia",
3       about [
4           p: "O museu virtual da fotografia se propõe a organizar as informações
           históricas dentro de temáticas
5           curatoriais apresentadas em cronologia.",
6           p: "Partimos de uma convicção: não  é possível compreender a importância, as
            possibilidades, o presente e o
7           futuro da fotografia sem compreende as ideias e os conceitos, os
           equipamentos e as tecnologias, as práticas e
8           os usos, os personagens e suas trajet órias pelos quais e por onde a
           fotografia passou para chegar onde
9           chegou nos dias de hoje.",
10      ]
11      carousel [
12          interval: 5,
13          images [
14              caption: "Author - Ricardo Ravanello", src: "imagem-capa-mvif.png",
          active,
15              caption: "Fornalha de um Hammam - Marrocos, 2015 (Ricardo Ravanello)",
          src: "imagem2-mvif.jpg",
16          ]
17      ]
18  ]
```

Listing 3 Production rule p2: menu.

```
1  [p2] header:  'menu' '[' (optionHeader)+ ']' ;
2       optionHeader:  brand | backgroundColor | fontColor | behaviourStat | items ;
```

Listing 2 is a part of the CaVa^{DSL} MVIF specification written according to the derivation rule p1.

The non-terminal grammar symbol *menu* defines the main menu of the virtual Learning Space that is composed of: brand, background and foreground colors; behaviour (fixed menu or moving following the page scroll); type of the menu links (simple or dropdown), containing the label and the link. Production rule p2 in Listing 3 formalizes this composition.

The part of the CaVa^{DSL} MVIF specification in Listing 4 is written according to the derivation rule p2.

Production rule p3 in Listing 5 states that the non-terminal symbol *exhibitions* (a part of the content block and the main component of CaVa^{DSL}) is a list of exhibition rooms. Each exhibition is composed of: title, short description and icon; additional info with a title and a description; behaviour (collapsed or expanded); exhibition type (permanent, temporary[5], future, or special); query operator – **all** (search for all occurrences of a determined ontology

---

[5] If the type is set up to 'temporary', it is possible to configure a notification to the visitor based on the exhibition expiration date.

Listing 4 Example applying the production rule p2: menu.

```
1  menu [
2      brand: "Museu Virtual Interativo da Fotografia",
3      background color: green,
4      foreground color: white,
5      behavior: fixed,
6      options [
7          label: "Exibições", dropdown [
8              dropdown label: "Permanentes", url: "permanentes",
9              dropdown label: "Temporárias", url: "temporarias",
10             dropdown label: "Especiais", url: "especiais",
11             dropdown label: "Futuras", url: "futuras",
12         ]
13         # "Temáticas" dropdown menu
14         label: "Sobre", url: "sobre_mvif", extension: php,
15     ]
16 ]
```

Listing 5 Production rule p3: exhibitions.

```
1  [p3] exhibitions:  'exhibitions' '[' (exhibition)+ ']' ;
2                     exhibition:  'exhibition' '[' (optionExhibition)+ ']' ;
3                     optionExhibition:  exhibitionTitle | exhibitionShortDescription |
       exhibitionIcon
4                         | exhibitionBehaviour | exhibitionAdditionalInfo | exhibitionType
       | exhibitionNotification | (queryOperators | sparql) ;
```

Listing 6 Production rule p4: queryOperators.

```
1  [p4] queryOperators:  all | one ;
2      all:  CONCEPT '->' 'all' '(' parametersAll ')' labelsOptions ;
3      parametersAll:  listName ',' mappingOrTriplesFileName ',' ontologyFileName ;
4      listName:  TEXT ;
5      mappingOrTriplesFileName:  TEXT ;
6      ontologyFileName:  TEXT ;
7      labelsOptions:  '[' (labelsExhibitionRoom)+ ']' ;
8      labelsExhibitionRoom:  elem (',' elem)* ;
9      elem:  TEXT ;
```

Listing 7 Production rule p5: sparql.

```
1  [p5] sparql:  'SPARQL' '[' sparqlStatement ']' '[' labelsOptions ']' ;
```

concept declared and returns the set of instances); or **one** (search for only one object (first instance) that corresponds to the conditional parameter and the ontology concept) – or SPARQL query (specified according to the production rule **p5**).

The `queryOperators` non-terminal symbol defines the operator that shall be used to query the database repository using the ontology vocabulary. The production rule **p4** in Listing 6 formalizes the alternatives and the components of such operators, crucial to extract the information to exhibit in each museum's room.

The rule **p5** in Listing 7 defines the non-terminal symbol `sparql` that offers another operator to write a query resorting directly to the SPARQL query language.

The non-terminal `sparqlStatement` corresponds to a declaration based on the SPARQL grammar.

Listing 8 Example applying the production rule p3 and p4: exhibitions and queryOperators.

```
1   exhibitions [
2        exhibition [
3            title: "Técnicas da Fotografia",
4            short description: "Dividimos a história da fotografia em três grandes
        períodos. Essa classificação se
5            justifica não apenas pela mudança dos suportes ou das técnicas, mas
        também pelo fato de que é possível
6            diferenciar drasticamente todo o sistema em torno da fotografia em cada
        um dos três momentos.",
7            icon: "camera-retro",
8            additional info [
9                title: "1672-2010",
10               description: "Período",
11           ]
12           behavior: expanded,
13           type: permanent,
14           Technique->all("Técnicas", "mvif.obda", "http://semanticweb.org/
15               rgm/2017/mvif/")[headerOfEachElement:"Técnica", "Década", "Descrição", "
        Período"],
16       ]
17  ]
```

Listing 9 Production rule p6: footer.

```
1   [p6] footer:  'footer' '[' (optionFooter)+ ']' ;
2        optionFooter:  footerImage | footerFormatDate | footerDeveloper |
        footerBehavior | footerStyle ;
```

To demonstrate how the production rules `p3` and `p4` are applied, an MVIF specification example is presented in Listing 8. Notice that the terminal symbol `CONCEPT` used in production rule `p4` (Listing 6) must denote a concept belonging to the ontology.

At last, the non-terminal symbol *footer* is used to specify an area at the bottom of the LS, containing: images and date; company or developer name; behaviour (fixed footer or moving according to the page scroll); style (simple footer with the data above mentioned or extended with an array of links with title, subtitle, URL, icon, etc.[6]). This is formalized by production rule `p6` in Listing 9.

The fourth part of the CaVa^DSL MVIF specification, written according to the derivation rule `p6`, is presented in Listing 10 to illustrate its use.

Notice that each block and component specification starts with the left bracket "[" and closes with the right bracket "]", always embracing pairs consisting of plain text or built-in terms.

4 CaVa^gen

CaVa^gen is a component of CaVa system as fully documented in [4, 5]. As can be seen in the block diagrams shown in the previously referred website at `http://www4.di.uminho.pt/~gepl/cava/MVIF/`, this CaVa component consist of four processors, namely CaVa^structure,

---

[6] The extended footer is good for addresses, social network links and other important information related to the virtual Learning Space.

Listing 10 Example applying the production rule p6: footer

```
1   footer [
2       images [
3           image: "cava_logo.png",
4           alignment: right,
5       ]
6       format date: "Y",
7       developer [
8           name: "Ricardo Martini",
9           alignment: left,
10      ]
11      behavior: fixed,
12      style: condensed,
13  ]
```

CaVa$^{\text{queries}}$ CaVa$^{\text{queriesTriple}}$ (the laternative to deal with triples datastorage, instead of mappings), and CaVa$^{\text{run}}$[7].

The first one is responsible for the generation of the static content of the virtual LS. Moreover, CaVa$^{\text{structure}}$ has the task of executing the CaVa$^{\text{queries}}$ (or CaVa$^{\text{queriesTriple}}$) processor when, at least, one query operator is set up in the CaVa$^{\text{DSL}}$ Specification. The second and third modules have the duty of assembling the ontology queries based on the query operator(s) specified in the CaVa$^{\text{DSL}}$ description. They shall handle the mapping or other intermediate file that links the ontology to the database (e.g., an OBDA mapping file, a Terse RDF Triple (turtle) file, etc). The last one, CaVa$^{\text{run}}$, is responsible for executing the queries mounted by CaVa$^{\text{queries}}$ (or CaVa$^{\text{queriesTriple}}$) processor and generates the queries results file to be used by the LS Scripts.

The purpose of CaVa$^{\text{gen}}$ processors is to get the LS Specification (CaVa$^{\text{DSL}}$) as input and transform it into several scripts possibly written in more than one web language (e.g., HTML, PHP, JS, template engines, CSS, etc.) and other kind of documents (e.g., state files[8]), that together make up multiple web pages, i.e., the complete virtual Learning Space.

CaVa$^{\text{gen}}$ processors have two objectives: (1) parsing the CaVa$^{\text{DSL}}$ specification of a virtual LS that was manually written by the enduser[9] according to CaVa$^{\text{grammar}}$; (2) generating/setting up the LS scripts component of CaVa$^{\text{render}}$, that comprises the configuration and script files necessary to be rendered by the web browser[10].

Figure 2 shows a simple CaVa$^{\text{gen}}$ workflow involving the three steps that shall be followed to achieve the main goal of this work: Specification of a virtual Learning Space in CaVa$^{\text{DSL}}$; Automatic generation and assembly of queries; Automatic generation of static and dynamic content of LS Scripts.

From step 1, if there is at least one query operator specified at any exhibition room of the CaVa$^{\text{DSL}}$ specification, step 2 is executed, followed by step 3; otherwise, after step 1, step 3 is executed (excluding step 2), generating only the static content of the LS. After executing these steps, the web browser (comprised in CaVa$^{\text{render}}$) will receive the necessary script files aiming at interpreting and rendering the final virtual Learning Space specified.

---

[7] Due to space constraints, we can't include here the block diagrams that depict the system architecture; this is way we direct the Reader to the project website and papers.

[8] State files contain data to be used by the processors of CaVa system in order to retrieve important information to proceed with the execution and generation of the virtual LS.

[9] the Museum Curator or another expert in cultura heritage information.

[10] To produce the desired the virtual Learning Space.
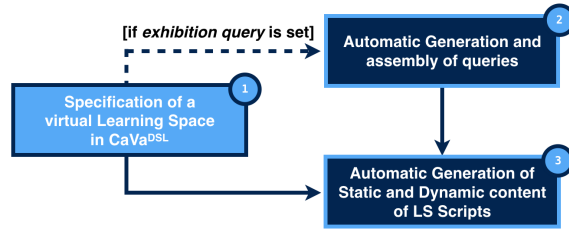
■ **Figure 2** CaVa workflow: from the CaVa^DSL Specification to the virtual LS automatic generation in three steps.

■ **Figure 3** Main configuration rendered.

5 Generating and Rendering MVIF virtual LS with CaVa^gen

Based on the process explained in Section 4, CaVa^gen analyzes the MVIF specification, written in CaVa^DSL (as discussed in Section 3), and transforms it into several scripts that together constitute the envisage exhibition rooms of that virtual museum. Figures 3 to 6 illustrate the effect of rendering, via an web browser, the generated MVIF program and data files. For the sake of space, only some generated web pages will be shown (namely mainconfig, menu, and exhibitions); more details are available from the website http://www4.di.uminho.pt/~gepl/cava/MVIF/.

Figure 3 shows the MVIF *entrance all* (actually the homepage) after rendering the files generated from the `main configuration` part of the CaVa^DSL specification. Notice in Figure 3 the main components (*LS title*, *about* and *carousel*) for the museum's web pages created according to that specification.

Based on the `menu` part of the MVIF specification (CaVa^DSL code shown in Listing 3), the museum's menus are rendered as illustrated in Figure 4. According to that specification, the bar on the top of the museum's window, exhibits the LS name (`Museu Virtual Interativo da Fotografia`), and three menu options: two buttons corresponding to dropdown lists (named `Exibições` and `Temáticas`), and a simple button named `Sobre`.

The exhibitions running in the museum are group in four type. As only one, of *permanent* type, was specified in our the running example, it will be the only available in the menu button `Exibições` under the option `Permanentes`. Figure 5 shows the exhibition room rendered using the files generated from the `exhibitions` part of the MVIF CaVa^DSL specification.

In Figure 5 it is possible to see the exhibition attributes specified as *visible*: `title`, `short description`, `icon` (camera-retro), `additional info` (1672-2010 Período) and `behavior` (expanded, denoted by the chevron-up icon in the top right corner).

Figure 4 MVIF Menu rendered.

Figure 5 Exhibitions list rendered.
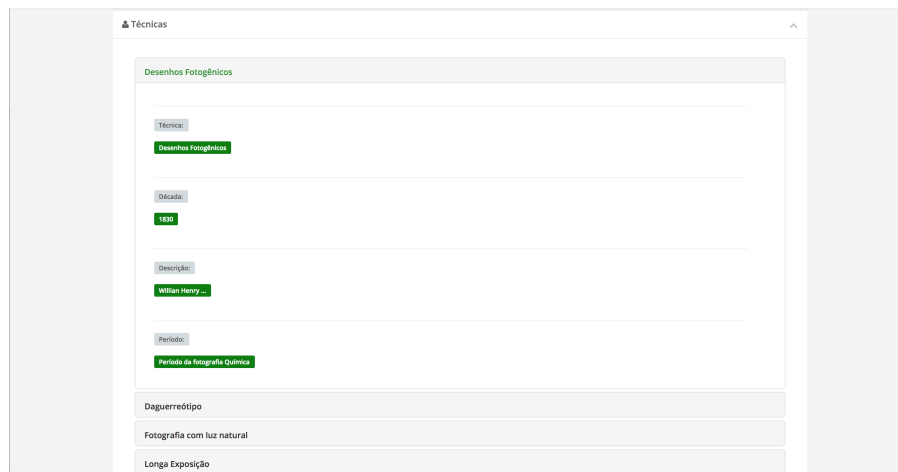
Figure 6 Permanent exhibition room rendered.

The final exhibition room rendered, containing the data retrieved from the knowledge repository by the query performed, can de seen in Figure 6. All the details of the webpage shown were rigorously defined in the MVIF specification listed previously (see Section 3.

6 Conclusion

In this paper CaVa$^{\text{DSL}}$ language was presented. This language was specifically designed to allow the curators of *memory institutions* to describe rigorously virtual Learning Spaces aiming at their automatic generation. CaVa$^{\text{gen}}$ [4], the engine developed to process CaVa$^{\text{DSL}}$ specifications, was also introduced briefly in the paper.

The language itself was specified by a typical context-free grammar written in ANTLR notation [7]. Its complex processor was generated automatically by the compilers-generator system ANTLR from a translation grammar defined by a set of Java Listeners written over the CFG referred above. So the CaVa$^{\text{DSL}}$ development process is not worthwhile for further discussion in the present context. The objective of the paper was, on one hand, to introduce the language design tuned for an easy usage in cultural environments, and on the other

hand, to emphasize the power of automatic generation of programs, describing a complex application domain.

Although not discussed in the paper, CaVa^DSL was applied to three different Cultural Heritage scenarios, specifying and creating virtual Learning Spaces complying with their needs as defined by the respective ontologies [3].

To proceed, we intend to extend CaVa^DSL to incorporate more powerful elements, like more sophisticated query operators, variables or functions, that will increase the language expressiveness and improve its usability. We also consider important to investigate the possibility to use YAML or other host language, instead of building it from the scratch; this approach could enable the reuse of some YAML features for free. Furthermore, we plan to work soon on the improvement of: the generated User Interface; the portability of the system; and the system performance. The most important and sensible future work direction is the design and conduction of experiments to assess the system's usability and collect end-user feedback. This is a crucial step to properly move CaVa project onwards.

References

1 Janna Quitney Anderson. *Challenges and Opportunities: The Future of the Internet.* Cambria Press, 2010.

2 Ricardo G. Martini, Cristiana Araújo, Pedro Rangel Henriques, and Maria João Varanda Pereira. CaVa: an example of the automatic generation of virtual learning spaces. In *Trends and Advances in Information Systems and Technologies*, volume 1, pages 633–643. Springer, 2018.

3 Ricardo Giuliani Martini. *Formal Description and Automatic Generation of Learning Spaces based on Ontologies.* PhD thesis, University of Minho, 2018. (to be discussed).

4 Ricardo Giuliani Martini and Pedro Rangel Henriques. Automatic generation of virtual learning spaces driven by CaVa^{DSL}: An experience report. *AMC SIGPLAN Notices*, 52(12):233–245, 2017. `doi:10.1145/3170492.3136046`.

5 Ricardo Giuliani Martini, Giovani Rubert Librelotto, and Pedro Rangel Henriques. Formal description and automatic generation of learning spaces based on ontologies. *Procedia Computer Science*, 96:235–244, 2016. `doi:10.1016/j.procs.2016.08.136`.

6 Melbourne declaration on educational goals for young australians, 2008. Ministerial Council for Education, Early Childhood Development and Youth Affairs. URL: `http://www.curriculum.edu.au/verve/_resources/National_Declaration_on_the_Educational_Goals_for_Young_Australians.pdf`.

7 Terence Parr. *The Definitive ANTLR 4 Reference.* Pragmatic Bookshelf, 2nd edition, 2013.

8 Ricardo Brisólla Ravanello. *Narrativa para bens culturais: tecnologias e aplicabilidades da fotografia digital expandida em museus virtuais.* PhD thesis, Universidade do Minho, 2018.

Non-LR(1) Precedence Cascade Grammars

José-Luis Sierra

Fac. Informática. Universidad Complutense de Madrid
C/ Prof. José García Santesmases 9. 28040 Madrid, Spain
jlsierra@ucm.es
https://orcid.org/0000-0002-0317-0510

Abstract

Precedence cascade is a well-known pattern for writing context-free grammars (CFGs) that model the syntax of expression languages. According to this method, precedence levels are represented by non-terminals, and operators' attributes are used to write syntax rules properly. In most cases, the resulting *precedence cascade grammar* (PCG) has neat properties that facilitate its implementation. In particular, many PCGs are LR(1) grammars, which serve as input for conventional bottom-up parser generators. However, for some cumbersome operator tables the method does not produce such neat grammars. This paper focuses on these cumbersome operator tables by identifying several conditions leading to non-LR(1) PCGs.

1 Introduction

Most computer languages include an *expression sub-language* as their most distinctive feature. This sub-language allows users to begin with a repertoire of primitive expressions and create more complex expressions by combining simpler ones. Such a combination is carried out by operators [13].

In this paper we will focus only in the most common classes of operators: binary infix, and unary prefix and postfix operators. In addition, we will adopt the conventions of the Prolog language to describe the attributes for these operators [5]:

- Each operator will have a *name* (e.g., $+, -, * \ldots$). It will be possible to *overload* this name, allowing different operator definitions to share such a name.
- Each operator will belong to a *precedence level*. Each precedence level will be represented by a positive natural number. Operators in lower precedence levels will take *priority* over (i.e., will bind tighter than) operators in higher ones[1]. In addition, when an operator is used to build an expression, this expression will take the precedence level for that operator. Precedence levels for basic expressions will be 0.

---

[1] That is, following Prolog conventions, in this paper *precedence* and *priority* of operators will be *contravariant* properties.

Name	Precedence	Type
$\otimes$	3	fy
$\oplus$	3	xfy
$\boxplus$	2	yfx
$\otimes$	2	xfx
$\otimes$	1	yf

$$
\begin{aligned}
E_3 &\rightarrow \otimes E_3 \mid E_2 \oplus E_3 \mid E_2 \\
E_2 &\rightarrow E_2 \boxplus E_1 \mid E_1 \otimes E_1 \mid E_1 \\
E_1 &\rightarrow E_1 \otimes \mid E_0 \\
E_0 &\rightarrow a \mid (E_3)
\end{aligned}
$$

(a) Operator table for a sample expression language.

(b) PCG for the descriptions in Table 1a; it is an LR(1) grammar.

Figure 1 An operator table and its associated PCG.

- Operators will constrain the precedence levels of their arguments to be: (i) lower than their own precedence level (denoted by x in the description of the operator's argument), or (ii) lower or equal than such a precedence level (which will be denoted by y).

The fixity and the arguments' allowed precedences will together form the operator's *syntactic type*. Following Prolog convention, this type will be one of the following forms: (i) for infix operators, yfx, xfy, xfx; (ii) for prefix operators, fy, fx; and (iii) for postfix operators, yf, xf. This way, yfx operators are left-associative, xfy right-associative, and xfx non-associative. In turn, fy and yf are associative, while xf and fx are non-associative unary (prefix and postfix) operators. All this information can be condensed into an *operator table* for the language. Table 1a gives an example of an operator table[2].

To model the syntax of this kind of expression languages, it is possible to use a *precedence cascade* pattern, which is described to a greater or lesser extent in any typical textbook on compiler construction (e.g., [3, 8]). In order to describe the pattern, we will introduce the following notation:

- By $\downarrow(i)$ we will denote the precedence level immediately smaller than i, or 0 if i is the smallest precedence level.
- By $\top$ we will denote the greatest precedence level.

The pattern itself is based on the following conventions (Figure 1b shows the CFG that results from applying these conventions to the Table 1a):

- Each precedence level i has a non-terminal E_i associated with it that represents expressions built with operators at that level.
- Each operator $\odot$ in level i has a rule associated with it that characterizes the syntax of the expressions formed with that operator. This rule depends on the operator's type: (i) $E_i \rightarrow E_i \odot E_{\downarrow(i)}$ if the type is yfx; (ii) $E_i \rightarrow E_{\downarrow(i)} \odot E_i$ if it is xfy; (iii) $E_i \rightarrow E_{\downarrow(i)} \odot E_{\downarrow(i)}$ if xfx; (iv) $E_i \rightarrow \odot E_i$ if fy; (v) $E_i \rightarrow \odot E_{\downarrow(i)}$ if fx; (vi) $E_i \rightarrow E_i \odot$ if yf; and (vii) $E_i \rightarrow E_{\downarrow(i)} \odot$ if the type is xf.
- There is an additional rule $E_i \rightarrow E_{\downarrow(i)}$ for each level i.
- Finally, there is a non-terminal symbol E_0 that models the basic (i.e., literals, variables, function calls, etc.) and parenthesized expressions. In the sequel we will abstract all the basic expressions with a single a symbol. Thus, there will be an additional pair of rules $E_0 \rightarrow a \mid (E_\top)$

---

[2] Notice that, according to this operator table, an expression like "$\otimes a \oplus a \oplus a \otimes a\otimes$" will mean "$\otimes(a \oplus (a \oplus (a \otimes (a\otimes))))$", while another one like "$a \oplus \otimes a$" will be ill-formed (it should be written "$a \oplus (\otimes a)$").

Name	Prec.	Type
$\odot$	2	yfx
$\odot$	1	xfx

$$E_2 \;\rightarrow\; E_2 \odot E_1 \;\mid\; E_1$$
$$E_1 \;\rightarrow\; E_0 \odot E_0 \;\mid\; E_0$$
$$E_0 \;\rightarrow\; a \;\mid\; (E_2)$$

(a) Operator table with multiple definitions of the infix operator $\odot$. **(b)** PCG resulting of the operator table presented in Table 2a. **(c)** Two different parse trees for "$a \odot a$".
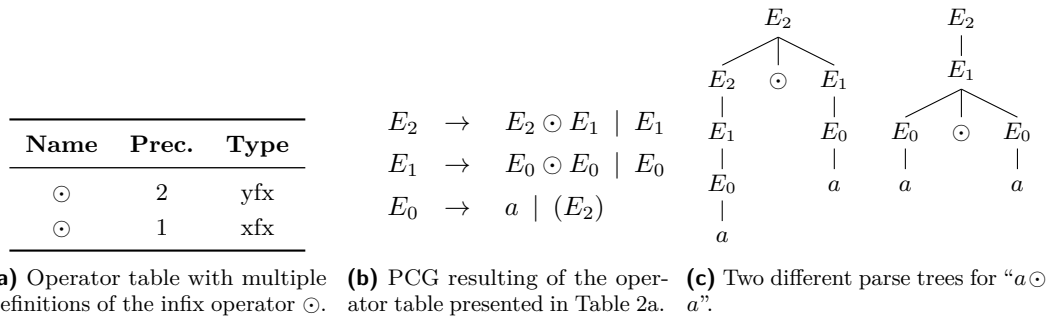
Figure 2 Example regarding multiple operator definitions with the same name and fixity.

We will refer to the CFGs produced by this pattern as *precedence cascade grammars* (PCGs). A well-known example of using this pattern for a real programming language is Jeff Lee's YACC grammar for ANSI C[3].

For most operator tables, the PCGs are LR(1) grammars [6] suitable for typical bottom-up, YACC-like, parser generators (this is the case, for instance, of the PCG in Figure 1b)[4]. However, there are also operator tables that lead to non-LR(1) grammars. Most of the time, this is due to contradictory operator definitions, which in turn produce ambiguous PCGs. Other times, such contradictions do not exist, but even so the resulting PCGs require more than one look-ahead symbol. In this paper we address these concerns by identifying common situations leading to non-LR(1) PCGs.

The rest of the paper is structured as follows. Section 2 describes the problems caused by multiple operator definitions with the same name and fixity. Section 3 addresses the problems caused by operators with opposite associativities at the same precedence level. Section 4 analyzes the concerns caused by the overloading of an operator in infix and postfix forms. Section 5 analyzes potential ambiguities caused by operators overloaded simultaneously in infix, prefix and postfix forms. Section 6 summarizes some work related to ours. Finally, Section 7 presents some conclusions and lines of future work.

2 Multiple operator definitions with the same name and fixity

Operator tables containing multiple operator definitions with the same name and fixity, but with different types and/or different precedence levels are intrinsically ambiguous, since any occurrence of the multiple-defined operator names can be explained indistinctly for either one or another definition. Therefore, the resulting PCG will be ambiguous.

Figure 2 illustrates the problems caused by this kind of tables. Notice that, since there are two definitions of the infix operator $\odot$, it is not possible to discern which version of $\odot$ is used. In consequence, the resulting PCG (Figure 2b) is ambiguous (and, therefore, non-LR(1)), as illustrated by the two different parse trees for the witness expression "$a \odot a$" in Figure 2c .

Finally, notice that the conditions reported in this section only affect multiple operator definitions with the same name and fixity. On the other hand, it is perfectly feasible to have multiple definitions with the same name, but with different fixities, and still obtain LR(1) PCGs (e.g., infix, prefix and postfix $\otimes$ in Figure 1).

---

[3] `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`
[4] These and other similar assertions on the LR(1) condition of particular CFGs can be verified, for instance, with the tools available online at `http://smlweb.cpsc.ucalgary.ca/`.

Name	Prec.	Type
$\odot$	1	yfx
$\boxdot$	1	xfy

(a) Operators precedence table.

$$E_1 \quad \rightarrow \quad E_1 \odot E_0 \ \mid \ E_0 \boxdot E_1 \ \mid \ E_0$$
$$E_0 \quad \rightarrow \quad a \ \mid \ (E_1)$$

(b) PCG for the operators described in Table 3a.
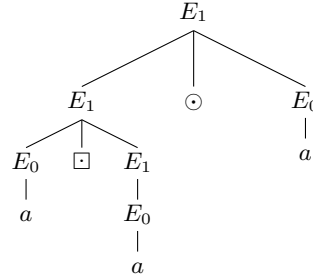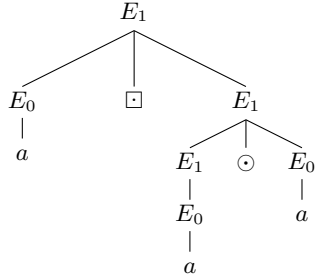
(c) Two different parse trees for "$a \boxdot a \odot a$".

▨ **Figure 3** Example regarding two operators with opposite associativies at the same precedence level.

3   Opposite associativities at the same precedence level

Another cause of non-LR(1) PCGs is the confluence, in the same precedence level, of two operators with *opposite* associativites, i.e., (i) one operator of type xfy with another one of type yfx or yf; (ii) a yfx operator with one of type fy; or (iii) a fy operator with a yf one. This confluence leads to ambiguity.

This situation is illustrated, for instance, by the operator Table 3a, which includes at the same precedence level a $\odot$ operator of type yfx and another one $\boxdot$ of type xfy. The resulting PCG is shown in Figure 3b. Thus, an expression like "$a \boxdot a \odot a$" will have two possible interpretations, depending on which of the two operators is applied first: "$(a \boxdot a) \odot a$" if $\boxdot$ is applied first, or "$a \boxdot (a \odot a)$" if it is $\odot$ that is applied first. As a result, the PCG in Figure 3b is ambiguous, as is proven in Figure 3c, which gives two different parse trees for "$a \boxdot a \odot a$". The other aforementioned unsuitable combinations due to opposite associatives can be illustrated in similar terms.

Finally notice that the existence of operators with different associativies at the same level only proves cumbersome for the aforementioned combinations. In this way, it is possible to find associative and non-associative operators at the same precedence level (e.g., $\boxplus$ and infix $\otimes$ in Figure 1), as well as several operators with the same associativity direction (e.g., $\oplus$ and prefix $\otimes$ in Figure 1), and still obtain LR(1) PCGs.

4   Overloading an operator with infix and postfix fixities

Definitions of an operator $\odot$ as an infix and a postfix one leads, in most of the situations, to non-LR(1) PCGs. Table 1 summarizes the different combinations and whether the resulting PCGs are LR(1) or not. These facts can be readily verified by providing the corresponding definitions, and generating and checking the resulting grammars[5].

---

[5] In particular, to check the LR(2) condition we used SLK (`http://www.slkpg.com/`), a parser generator that supports arbitrary look-ahead to resolve LR conflicts, and JikesPG (`http://jikes.sourceforge.net/`), another parser generator supporting arbitrary LALR(k) grammars.

■ **Table 1** Classes of PCGs for tables overloading a $\odot$ operator in infix (precedence level l_i) and postfix (precedence level l_p) forms (**LR(1)**: the resulting grammar is LR(1); **LR(2)**: the resulting grammar is LR(2), but not LR(1); **AMB**: the resulting grammar is ambiguous).

	$\tau_i = yfx$ $\tau_p = yf$	$\tau_i = yfx$ $\tau_p = xf$	$\tau_i = xfy$ $\tau_p = yf$	$\tau_i = xfy$ $\tau_p = xf$	$\tau_i = xfx$ $\tau_p = yf$	$\tau_i = xfx$ $\tau_p = xf$
$l_p > l_i$	**LR(2)**	**LR(1)**	**LR(2)**	**LR(2)**	**LR(2)**	**LR(2)**
$l_p = l_i$	**LR(1)**	**LR(2)**	**AMB**	**LR(1)**	**LR(2)**	**LR(1)**
$l_p < l_i$	**LR(2)**	**LR(2)**	**LR(1)**	**LR(2)**	**LR(1)**	**LR(2)**

Therefore, most of the combinations produce non-LR(1) PCGs. However, unlike previous scenarios, and with the exception of the case corresponding to the same precedence and yf and xfy types, which as indicated in the previous section leads to ambiguity, the resulting PCGs that are non-LR(1) are not ambiguous. On the contrary, they are LR(2) grammars.

Finally, remember that, as indicated in Table 1, there is also room for LR(1) PCGs for operator tables involving the infix and postfix forms of an operator. Indeed, an example is given in Figure 1, which overloads the $\otimes$ operator in infix and postfix forms.

5 Overloading an operator with infix, prefix and postfix fixities

The combinations of two operator definitions do not exhaust the conditions hindering LR(1) PCGs. Indeed, the overloading of an operator $\odot$ in infix, prefix and postfix forms can lead to ambiguity. The reason is that, in an expression like "$a \odot \odot a$", it is possible to consider: (i) the first occurrence of $\odot$ as a postfix operator and the second as an infix one; or (ii) the first as the infix operator and the second as a prefix one. In consequence, let l_i be the precedence level of the infix definition, let τ_i be its type, let l_{pre} be the precedence level of the prefix definition, and l_{post} that of the postfix one. Then, any of the following conditions lead to an ambiguous PCG [6]:

- $\tau_i = xfy$, $l_{post} < l_i$, $l_{pre} \leqslant l_i$.
- $\tau_i = yfx$, $l_{post} \leqslant l_i$, $l_{pre} < l_i$.
- $\tau_i = xfx$, $l_{post} < l_i$, $l_{pre} < l_i$.

Figure 4 illustrates one of these cumbersome combinations. The resulting PCG (Figure 4b) is ambiguous, as Figure 4c makes apparent. The ambiguity of the PCGs produced by the other cumbersome combinations can be illustrated in an analogous way.

Finally, notice that, by avoiding the cumbersome combinations described in this and the previous sections, it is possible to find tables with an operator overloaded in infix, prefix and postfix forms that lead to LR(1) PCGs. Again an example is given by the $\otimes$ operator in Figure 1.

6 Related work

As illustrated in this paper, ambiguity caused by cumbersome combinations of operator attributes is one of the main causes of non-LR(1) PCGs. In [7], starting from a characterization of the expression languages defined through precedence relations between operators, it is proved that, in the absence of operator overloading, ambiguity can be prevented by avoiding

---

[6] Any of these conditions make the ambiguous sentence "$a \odot \odot a$" a valid expression of the language.

Name	Prec.	Type
$\odot$	3	xfx
$\odot$	2	fx
$\odot$	1	xf

(a) Operator table.

$$E_3 \quad \rightarrow \quad E_2 \odot E_2 \mid E_2$$
$$E_2 \quad \rightarrow \quad \odot E_1 \mid E_1$$
$$E_1 \quad \rightarrow \quad E_0 \odot \mid E_0$$
$$E_0 \quad \rightarrow \quad a \mid (E_3)$$

(b) Resulting PCG.
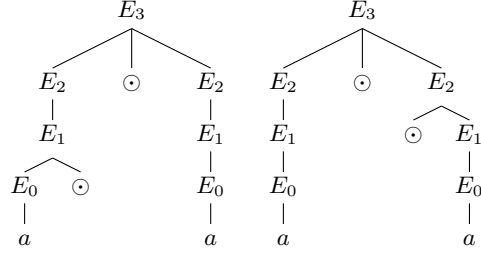
(c) Two parse trees for "$a \odot \odot a$"

Figure 4 Example regarding the overloading of an operator with infix, prefix and postfix fixities..

cycles in the operators' dependence graph. For the operators considered in our work these cycles only can arise between operators with the same precedence level and with opposite associativities. Therefore, this result is reflected in section 3. The work described in [1] proves that the syntax of expression languages without operator overloading in which each operator belongs to a different precedence level can be readily described with unambiguous CFGs. It is consistent with our analysis, since it leaves out all the cumbersome situations analyzed in the previous sections.

To a greater or lesser extent, languages with user-defined operators must cope with the aspects analyzed in this paper. Some representative examples of languages of this kind are Haskell, Scala, Sparrow and Prolog. For instance, Haskell [9] only provides support for user-defined infix operators with 10 precedence levels. No syntactic operator overloading is allowed. In addition, it is possible to find opposite associativies at the same precedence level, but expressions chaining left and right associative operators with the same precedence are rejected during parsing time. Scala [10] supports user-defined infix and postfix operators. It also supports a predefined set of prefix operators. Precedences are structured in two pre-established precedence classes (one class for prefix operators, another for infix ones), and each precedence class at a pre-established set of precedence levels. Actual precedence level and associativities are not literally declared but are derived from the operator's name. Associativity conflicts are managed as in Haskell. A similar approach is followed in Sparrow [14], although this language also allows user-defined prefix operators as well as the explicit declaration of precedences and associativies within each precedence class. Finally, as mentioned earlier, Prolog [5] supports definitions of operators analogous to those considered in this paper. In addition, the language includes some constraints on user-defined operators that, on one hand, avoid ambiguities and, on the other hand, facilitate parsing by limiting look-ahead. Specifically, it is not possible to define two operators with the same name and the same fixity (any attempt to do so redefines the operator instead of overloading it). It avoids the shortcomings described in section 2. Also, it is not possible to overload an operator as both an infix and a prefix one, which, on one hand avoids situations requiring more than one look-ahead symbol (like that described in section 4), and, on the other hand, avoids the potential ambiguities described in section 5. Finally, it solves the ambiguities derived from opposite associativities by making left-associative operators take priority over right-associative ones at the same precedence level. Therefore, as analyzed in this paper, all these languages exclude some perfectly valid combinations of operators from the point of view of unambiguity and limited (one symbol) look-ahead.

Finally, the implementation techniques for languages with user-defined operators are also relevant in the context of the current paper, since these techniques must also deal with valid and disallowed combinations of operator attributes. The work described in [11] describes how to use YACC to implement a parser for an expression language with arbitrary user-defined (not only prefix, postfix or infix) disfix operators. The approach does not support operator overloading. In addition, prefix operators always have a greater precedence level than infix ones, which in turn always belong to a greater level than the other dixfix operators, the ones with the highest priority. The work in [12] extends the approach based on ambiguous grammars and disambiguating rules in [2] by allowing the addition of new disambiguation rules during parsing time, which enables user-defined operators. It also provides some constraints to avoid cumbersome operator tables (that can lead to ambiguity or demand more than one look-ahead symbol), which are more restrictive than those posed by Prolog and those analyzed in the previous sections (for instance, one of the constraints requires all the overloaded operators to belong to the same precedence level). Finally, the work reported in [4] describes how to support dixfix operators by instantiating a PCG-like grammar scheme from precedence graphs that provide partial orderings on the precedence of the operators (instead of the total ordering provided by precedence levels). It also shows how, under the assumptions described in [7], the resulting CFG is unambiguous. Once again, none of these approaches accept all the legal combinations of operators identified in this paper.

7 Conclusions and Future work

In this paper we have explored the conditions under which the PCGs that model the syntax of expression languages become non-LR(1) CFGs. For this purpose, we have carried out a systematic analysis of combinations of two and three operator definitions, and we have characterized several problematic scenarios in grammatical terms. Most of them concern ambiguity in operator descriptions, which reflects ambiguous PCGs. Others expose the need for more than one look-ahead symbol. Although scenarios are avoided in most of the languages that support user-defined operators, from our analysis it is apparent that these design decisions could have been further refined (e.g., while Prolog prohibits overloading an operator in infix and prefix forms to limit the need for look-ahead, in this paper we have found that only certain combinations of this type of definitions lead to grammars which are not LR (1), but LR (2)). These findings also enable the direct analysis of operator tables, in order to diagnose potential problems and to explain such problems at the level of operator definitions (instead of, for instance, at the level of parsing conflicts in the generated PCGs).

While the set of combinations identified seems broad enough, the analysis performed has been fundamentally empirical. It is not possible, therefore, to affirm that an assertion like "*if an operator table does not contain any of the problematic combinations analyzed, then the resulting grammar will be* LR(1)" has been proved, but only that evidence in favor of it has been provided. It is necessary to carry out a more formal work oriented to proving this result or another similar to it, completing the catalog of problematic combinations if necessary. Another line of work is to consider where the resulting PCGs can be successfully transformed into appropriate CFGs for top-down parsing, as well as what the classes of these CGFs are (specially when these transformed CFGs are LL(1), or when they require more than one look-ahead symbol). Finally, we plan to run a similar analysis on the specifications based on ambiguous grammars plus disambiguating rules like those described by Aho et al [2].

─── **References** ───────────────────────

1 Annika Aasa. Precedences in specifications and implementations of programming languages. *Theoretical Computer Science*, 142(1):3–26, 1995. `doi:10.1016/0304-3975(95)90680-J`.

2 Alfred. V. Aho, Sethi Johnson, and Jeffrey D. Ullman. Deterministic parsing of ambiguous grammars. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 1–21, 1973. `doi:10.1145/512927.512928`.

3 Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

4 Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *20th International Conference on Implementation and Application of Functional Languages*, pages 80–99, 2011.

5 Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. Prolog syntax. In *Prolog: The Standard: Reference Manual*, pages 221–238. Springer, 1996. `doi:10.1007/978-3-642-61411-8_9`.

6 Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

7 Wafik Boulos Lotfallah. Characterizing unambiguous precedence systems in expressions without superfluous parentheses. *International Journal of Computer Mathematics*, 86(1):1–20, 2009. `doi:10.1080/00207160802166499`.

8 Kenneth C. Louden. *Compiler Construction: Principles and Practice*. PWS Publishing, 1997.

9 Simon Marlow, editor. *Haskell 2010 Language Report*. Haskell Community, 2010.

10 Martin Odersky. The Scala language specification, version 2.9. Technical report, Programming Methods Laboratory, EPFL, 2014.

11 Simon L. Peyton Jones. Parsing distfix operators. *Communications of the ACM*, 29(2):118–122, 1986. `doi:10.1145/5657.5659`.

12 Kjell Post, Allen Van Gelder, and James Kerr. Deterministic parsing of languages with dynamic operators. In *International Symposium on Logic Programming*, pages 456–472, 1993.

13 Ravi Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.

14 Lucian Radu Teodorescu, Vlad Dumitrel, and Rodica Potolea. Flexible operators in Sparrow. *International Journal of Research in Engineering and Technology*, 3(17):40–45, 2014.

ASAPP 2.0: Advancing the state-of-the-art of semantic textual similarity for Portuguese

Ana Alves
CISUC / ISEC, Polytechnic Institute of Coimbra, Portugal
ana@dei.uc.pt
 https://orcid.org/0000-0002-3692-338X

Hugo Gonçalo Oliveira
CISUC / Department of Informatics Engineering, University of Coimbra, Portugal
hroliv@dei.uc.pt
 https://orcid.org/0000-0002-5779-8645

Ricardo Rodrigues
CISUC / ESEC, Polytechnic Institute of Coimbra, Portugal
rmanuel@dei.uc.pt
 https://orcid.org/0000-0002-6262-7920

Rui Encarnação
CISUC, University of Coimbra, Portugal
race@dei.uc.pt
 https://orcid.org/0000-0002-5176-4137

─── **Abstract** ───

Semantic Textual Similarity (STS) aims at computing the proximity of meaning transmitted by two sentences. In 2016, the ASSIN shared task targeted STS in Portuguese and released training and test collections. This paper describes the development of ASAPP, a system that participated in ASSIN, but has been improved since then, and now achieves the best results in this task. ASAPP learns a STS function from a broad range of lexical, syntactic, semantic and distributional features. This paper describes the features used in the current version of ASAPP, and how they are exploited in a regression algorithm to achieve the best published results for ASSIN to date, in both European and Brazilian Portuguese.

1 Introduction

Computing the similarity of words or sentences in terms of their meaning is an active area of research in Natural Language Processing (NLP) and understanding (NLU). This is confirmed by related shared tasks, such as SemEval STS [2, 1, 8], which required the manual compilation of annotated data for benchmarking this specific task. Most successful approaches for English

learn a similarity function with ensembles of classifiers that combine different metrics, such as n-gram, word or chunk overlap, semantic relations, or distributional similarity [29, 32].

SemEval STS task targets English since 2012 and we can thus say that, for this language, STS is becoming mature. Spanish and Arabic were included in recent editions [1, 8], which have also targeted cross-lingual pairs. For other languages, STS is still in its early days. Until recently, there was not a public dataset for computing semantic similarity between Portuguese sentences. But, in 2016, a collection for STS in Portuguese was released in the scope of the ASSIN shared evaluation [12].

Following the participation of our ASAPP system [4] in ASSIN, we kept working towards the improvement of our results and advancing the state-of-the-art of Portuguese STS. This paper presents a post-evaluation approach to ASSIN, based on supervised machine learning.

The paper describes ASAPPV2.0 and focuses on the features currently extracted, many inspired by related work for English, but adapted for Portuguese. For some, we present the results achieved without supervision which, in some cases, were surprisingly high. Yet, the best results are obtained after learning a regression function, based in a varied set of lexical, syntactic, semantic and distributional features. In the end, we were able to improve not only the previous results of ASAPPv1.5 [14], but also outperform the best official results in ASSIN by two and four points, respectively in the European (PTPT) and Brazilian (PTBR) Portuguese collections.

The remainder of this paper starts by presenting some related work, in section 2, namely a brief overview of the best results for English STS, together with commonly used features, then focusing in Portuguese STS, mostly around the ASSIN task, its collections and best approaches. In section 3, all the exploited features are described and several are illustrated in examples, ending with some results obtained with different feature sets, but without supervision. In section 4, extracted features are exploited to learn a similarity function, this time with supervision, using not only different regression algorithms, but also using the training collections differently, towards the best results in the ASSIN task.

2   Related Work

The SemEval shared evaluations include STS tasks since 2012 [2]. Results are typically assessed by the Pearson correlation (hereafter ρ, between -1 and 1) and the Mean Squared Error (MSE) between values computed by the system and those based on the opinion of several human judges, for the same collection of pairs.

Most successful approaches are supervised. To learn a similarity function, they rely on an ensemble of classifiers and exploit different features, some of which as basic as token or n-gram overlap, but also similarity measures computed in WordNet [10], topics and deep semantic models (see, e.g., [29, 32]). For English, the best ρ has ranged from 0.618, in SemEval 2013, to 0.854 in SemEval 2017. For the adopted baseline – the cosine of the vectors that represent the words in each sentence of the pair – ρ has ranged from 0.311, in 2012, to 0.728, in 2017. For Spanish STS, the best system [32] in SemEval 2017 achieved $\rho = 0.856$, with similar features as the English version.

ASAP [3], which was the starting point of ASAPP, was originally developed for the *Evaluation of Compositional Distributional Semantic Models on Full Sentences through Semantic Relatedness* task [23], in SemEval 2014, but participated one year later in SemEval 2015 STS [5], though with modest results.

An earlier approach to Portuguese STS [27] exploited a knowledge base to identify related words in different sentences. The proposed measure was tested in natural language

Table 1 Examples from the training collections.

Collection	Id	Pair	Sim
PTPT	2675	**t:** *O Chelsea só conseguiu reagir no final da primeira parte.* **h:** *Não podemos aceitar outra primeira parte como essa.*	1.25
PTPT	315	**t:** *Todos que ficaram feridos e os mortos foram levados ao hospital.* **h:** *Além disso, mais de 180 pessoas ficaram feridas.*	3.00
PTBR	1282	**t:** *As multas previstas nos contratos podem atingir, juntas, 23 milhões de reais.* **h:** *Somadas, as multas previstas nos contratos podem chegar a R$ 23 milhões.*	5.00

descriptions of bugs in software engineering projects, which had their similarity annotated by two humans. But it was not until 2015 that a collection was publicly released for computing STS in Portuguese, with the goal of being used in the ASSIN shared task [12], which targeted Semantic Similarity and Textual Entailment in Portuguese. Training data comprised 3,000 sentence pairs for PTBR, and another 3,000 for PTPT. Test data comprised 2,000 PTBR pairs and 2,000 PTPT pairs. While recent editions of English STS have used text from varied sources, sentences in the ASSIN collections were obtained exclusively from Google News. Table 1 shows three pairs in the ASSIN training collections, including ids, sentences (t, for text, h, for hypothesis), and the average similarity given by four human judges that followed the same guidelines. Similarity values range from 1 (completely different sentences, on different subjects) to 5 (sentences mean essentially the same).

ASSIN had 6 participating teams, which submitted 14 runs for the STS task in PTBR and 17 in PTPT. Distinct systems achieved the best official results for PTPT and PTBR. For PTBR, the best run [17] achieved $\rho = 0.70$ with $MSE = 0.38$, obtained by computing the cosine similarity of a vector representation of each sentence, based on the sum of the TF-IDF scores and word2vec [25] vectors of each word. For PTPT, the best run [11] achieved $\rho = 0.73$ with $MSE = 0.61$, obtained after learning a similarity function with a Kernel Ridge Regression using several similarity metrics as input, computed between the two sentences of each pair, including overlap and set similarity measures on multiple text representations (e.g., lowercase, character trigrams). ASAPP [4], an adaptation of ASAP to Portuguese, also participated, with best runs achieving $\rho = 0.65$ and $MSE = 0.44$, for PTBR, and $\rho = 0.68$ and $MSE = 0.70$ for PTPT.

As the collections of ASSIN are available[1], work on Portuguese STS continued, even after the evaluation, using those collections as benchmarks. This included our previous work [14], where we report on gradual improvements as features and techniques are added, though without outperforming the best results. An important conclusion was that the best test results ($\rho = 0.711$ for PTPT, and $\rho = 0.697$ for PTBR) were obtained after training the model on both the PTPT and PTBR collections. But other recent works tackled Portuguese STS and relied on the ASSIN collection for evaluation [18, 7]. Hartmann et al. [18] tested a broad range of distributional similarity models of Portuguese (word embeddings) for different NLP tasks, including STS on the ASSIN collection. The best results obtained are quite low ($\rho = 0.60$ for PTBR, using Wang2vec skip-gram with 1,000 dimensions; $\rho = 0.55$ for PTPT, using word2vec CBOW with 600 and 1,000 dimensions), but their main goal was to compare the models, also developed by them. Their results suggest that relying on a single feature, even on large quantities of data, or on a small set of features of the same kind is not

---

[1] `http://nilc.icmc.usp.br/assin/`

enough to achieve high STS scores. Cavalcanti et al. [7] used a regression algorithm that exploited four features: the cosine similarity between the sentence vectors weighted with TF-IDF; word2vec similarity based on a three-layer sentence representation; word overlap; length of the shortest sentence divided by the length of the longest. They outperformed the best results for PTBR, with $\rho = 0.71$ and $MSE = 0.37$, and achieved $\rho = 0.70$ and $MSE = 0.57$, for PTPT.

3 Features for Semantic Textual Similarity in Portuguese

In order to compute their semantic similarity, several features are extracted from the ASSIN sentence pairs. A broad range of features was exploited, including lexical, syntactic, semantic and distributional features. Some were already used in previous versions of ASAPP [4, 14], but new features are new in ASAPPV2.0, namely the dependency-based and distributional features. Although these were later used to learn a model of STS, in the end of this section, we reveal a selection of unsupervised results, obtained with some subsets of related features.

Several features were extracted with the NLPPort [28] tools, developed in our group and freely available[2]. Those include TokPORT, a tokenizer; TagPORT, a part-of-speech tagger; ChkPORT, a syntactic chunker; LemPORT, a lemmatizer; and EntPORT, a named entity recognizer. In addition, PTStemmer[3] was used for obtaining the stems of each token. To acquire the semantic features, we resort to a set of Portuguese lexical knowledge bases (LKBs), enumerated in Section 3.3. Syntactic dependencies and distributional features were extracted with the spaCy toolkit[4].

3.1 Lexical Features

The following lexical features, related to words at the surface level, were exploited:

- Number of common tokens, after tokenization with TokPort.
- Number of negation words (*não*, *nada*, *nenhum*, *de modo algum*, ...) in each sentence of the pair and their absolute difference.
- Number of common lemmas, obtained with LemPORT.
- Number of common stems, obtained with PTStemmer.

Set similarity metrics were computed to devise their integration in the feature set. Those metrics included the Jaccard, Overlap, Dice coefficient, plus the Cosine Similarity, computed according to equations 1, 2, 3, 4, respectively, for the sets of the tokens, lemmas and stems, in each sentence of the pair (T and H).

$$Jaccard(T, H) = \frac{|T \cap H|}{|T \cup H|} \qquad (1) \qquad Dice(T, H) = \frac{|T \cap H|}{|T| + |H|} \qquad (3)$$

$$Overlap(T, H) = \frac{|T \cap H|}{|min(T, H)|} \qquad (2) \qquad Cos(T, H) = \frac{|T \cap H|}{\sqrt{|T|}\sqrt{|H|}} \qquad (4)$$

---

[2] NLPPort is available from `https://github.com/rikarudo/`
[3] PTStemmer is available from `https://code.google.com/archive/p/ptstemmer/`
[4] `https://spacy.io/`

(t) *Ricky Álvarez voltou hoje, dia 17 de Setembro, a ser associado à equipa do Futebol Clube do Porto.*
   NP: [Ricky Álvarez], [17 de Setembro], [a equipa], [o Futebol Clube do Porto]
   VP: [voltou], [dia]*, [ser associado]
   PP: [a], [a], [de]
   ADVP: [hoje]

(h) *Nas suas três temporadas na equipa de Milão, Ricky participou em 90 jogos e apontou 14 golos.*
   NP: [as suas três temporadas], [a equipa], [Milão], [Ricky], [90 jogos], [14 golos]
   VP: [participou], [apontou]
   PP: [em], [em], [de], [em]

Non-Zero Features	values		
	t	h	|#t - #h|
# NP	4	6	2
# VP	3	2	1
# PP	3	4	1
# ADVP	1	0	1

Figure 1 Extraction of noun, verbal, propositional and adverbial phrases and related features.

3.2 Syntactic Features

The set of syntactic features exploited included the number of noun, verb, prepositional and adverbial phrases in each sentence of the pair and their absolute difference. Figure 1 illustrates the chunk-based features, with their computation in a pair of sentences.

In ASAPPv2.0, syntactic dependencies are also exploited, namely the Jaccard coefficient between the dependencies in the first sentence of the pair and those in the second sentence. Syntactic dependencies were computed with spaCy's dependency parser. Each sentence is converted to a list of triples related to the arcs in the dependency tree, ignoring just the punctuation tokens. Each triple – $(token_1, token_2, DEPENDENCY)$ – contains two connected tokens (head and child) and the syntactic dependency name that labels the relation. The Jaccard similarity of the two lists is then computed to measure the similarity of the pair of sentences, as in equation 5. The computation of the previous feature is illustrated in Figure 2.

$$Jaccard\_Dep(T, H) = \frac{|Dep(T) \cap Dep(H)|}{|Dep(T) \cup Dep(H)|} \tag{5}$$

When computed with this feature, alone, semantic similarity is poor, but it captures some relations that are not covered by the other features used in previous versions of ASAPP. Namely, it aims at capturing the dissimilarity between sentences such as: {"The tiger killed the man.", "The main killed the tiger"}. Besides their strong overlapping and exact matches of noun phrases and verbal phrases, their outcome is significantly different.

3.3 Semantic Features

Given not only their importance for understanding the meaning of a sentence, but also their frequent presence in the ASSIN collection, named entities in the sentences of the pair were extracted and classified into one of nine types (abstraction, product, event, number, organization, person, place, thing and time). The number of named entities of each type in the sentences is used as features, plus their absolute difference, which makes a total of 27 features. Figure 3 illustrates the computation of those features.

(t) *Sebastian Vettel garantiu a pole-position para o Grande Prémio de Singapura de Fórmula 1.*

- ('Sebastian', 'Vettel', FLAT:NAME), ('garantiu', 'Sebastian', NSUBJ), ('garantiu', 'pole', OBJ),
- ('pole', 'a', DET), ('pole', 'position', APPOS), ('postion', 'Grande', NMOD)
- ('Grande', 'para', CASE), **('Grande', 'o', DET), ('Grande', 'Prémio', FLAT:NAME), ('Grande', 'Singapura', NMOD)**
- **('Singapura', 'de', CASE), ('Singapura', 'Fórmula', NMOD)**
- **('Fórmula', 'de', CASE), ('Fórmula', '1', FLAT:NAME)**

(h) *O Grande Prémio de Singapura de Fórmula 1 tem início marcado para as 13h00 de domingo.*

- **('Grande', 'o', DET), ('Grande', 'Prémio', FLAT:NAME), ('Grande', 'Singapura', NMOD)**
- **('Singapura', 'de', CASE), ('Singapura', 'Fórmula', NMOD)**
- **('Fórmula', 'de', CASE), ('Fórmula', '1', FLAT:NAME)**
- ('tem', 'Grande', NSUBJ), ('tem', 'início', OBJ), ('início', 'marcardo', ACL), ('marcardo', ' 13h00', OBL)
- ('13h00', 'para', CASE), ('13h00', 'as', DET), ('13h00', 'domingo', NMOD, ('doming', 'de', CASE)

$$Jaccard\_Dep(T, H) = \frac{7}{22} \approx 0.3182$$

Figure 2 Extraction of syntactic dependencies with the spaCy toolkit and its related feature.

Language is flexible in such a way that the same idea can be transmitted through different words, generally related by well-known semantic relations, such as synonymy or hypernymy. These relations are implicitly mentioned in dictionaries, and explicitly encoded in LKBs, such as WordNet [10]. We decided to use LKBs currently available for Portuguese, namely three wordnets – WordNet.Br [9] (which covers only verbs), OpenWordNet-PT (OWN.PT) [26] and PULO [30]; two synonymy-based thesauri – TeP [24] and OpenThesaurus.PT[5]; three lexical networks extracted from Portuguese dictionaries – PAPEL [15] and relations from Dicionário Aberto [31] and Wiktionary.PT[6]; and the semantic relations in a set of linguistic resources – Port4Nooj [6]. All of these LKBs cover synonymy relations (e.g., *realçar* synonym-of *sublinhar*) , all but OpenThesaurus.PT, WordNet.Br, and Port4Nooj cover antonymy (e.g., *tristeza* antonym-of *alegria*) , all but TeP and OT cover hypernymy relations (e.g., *mover* hypernym-of *tremer*) , in addition to relations of other types, covered only by some LKBs, such as part-of (e.g., *núcleo* part-of *átomo*), causation (e.g., *frio* causation-of *crestar*) , or purpose (e.g., *polir* purpose-of *lixa*) , among others.

The aforementioned LKBs have substantially different sizes and the creation of most involved some degree of automation, which means that they contain noise, including rarely used words and meanings, not so useful relations, and also actual errors. Therefore, we rely on redundancy to build more reliable and useful semantic networks [13], namely *Redun2* and *Redun3*, which include all the relation instances respectively in at least two or three LKBs. They were exploited in different ways (see Section 3.5), but the final model only considered the following features:

- Set similarity metrics considering semantic relations in *Redun3* LKB: after computing the overlap of the similarity of the stems, the metrics were adjusted as in equation 6.

---

[5] `http://paginas.fe.up.pt/~arocha/AED1/0607/trabalhos/thesaurus.txt`
[6] `http://pt.wiktionary.org`

(t) *Ricky Álvarez voltou hoje, dia 17 de Setembro, a ser associado à equipa do Futebol Clube do Porto.*
   Person: [Ricky Álvarez]
   Time: [17 de Setembro]
   Org: [Futebol Clube do Porto]
(h) *Nas suas três temporadas na equipa de Milão, Ricky participou em 90 jogos e apontou 14 golos.*
   Person: [Ricky]
   Place: [Milão]
   Numeric: [90], [14]
   Event: [jogos]

Non-Zero Features	values		
	t	h	|#t - #h|
# Person	1	1	0
# Time	1	0	1
# Organization	1	0	1
# Place	0	1	1
# Numeric	0	2	2
# Event	0	1	1

Figure 3 Extraction of named entities and related features.

There, γ was set according to equation 7 and Sim was computed according to equation 8. Constants were empirically set to $\alpha = 0.75$ and $\beta = 0.05$.

$$Jaccard^+(T, H) = \frac{|T \cap H| + \gamma}{|T \cup H|} \tag{6}$$

$$\gamma = \sum_{i=1}^{|T'|} \sum_{j=1}^{|H'|} Max(Sim(T'_i, H'_j)) \qquad (7) \ Sim(T'_i, H'_j) = \begin{cases} \alpha, & \text{if } dist(T'_i, H'_j) = 1 \\ \beta, & \text{if } dist(T'_i, H'_j) = 2 \\ 0, & \text{otherwise} \end{cases} \ (8)$$

- A feature for each of four relation groups considered (synonymy, hypernymy, antonymy and other) in each LKB, which would be the number of semantic relations of those types held, in the target LKB, between lemmas in one sentence of the pair and lemmas in the other, normalized after division by the sum of the number of open-class words (nouns, verbs, adjectives and adverbs) in sentences t and h. Although these features would clearly not be enough for computing similarity all alone, our belief is that they would be a useful complement to the other.

Figure 4 illustrates the computation of the Jaccard$^+$ feature with the *Redun3* network.

3.4 Distributional Features

In ASAPPv2.0, distributional features are also exploited, namely word and character n-gram distribution, and models of distributional similarity. For convenience reasons, these features were extracted with Python tools, in opposition to all the others, extracted with tools in Java.

Set similarity features already covered the similarity of n-grams of size 1. Yet, the new features considered n-grams of size 2 with additional restrictions. Character n-grams were also considered, as they are known for capturing features at different levels, and can be

Sentences:

(t) *Os Estados Unidos anunciaram oficialmente esta sexta-feira o abandono de um plano que visava treinar e equipar rebeldes na Síria.*

(h) *Os Estados Unidos anunciaram esta sexta-feira que interrompe o projeto de treino de rebeldes sírios.*

Relations in *Redun3* connecting words in *t* with words in *h*:

- *plano* synonym-of *projeto*
- *Síria* place-of *sírio*

- Synonymy (1): $\frac{1}{|openClassWords(t)+|openClassWords(h)|}$
- Other relations (1): $\frac{1}{|openClassWords(t)+|openClassWords(h)|}$

$$Jaccard^+(T,H) = \frac{lemmas(t) \cap lemmas(h) + \alpha + \beta}{lemmas(t) \cup lemmas(h)} = \frac{9 + 0.75 + 0.05}{23} \approx 0.39$$

Figure 4 Computation of Jaccard$^+$ feature with Reund3 as the semantic network.

extremely useful in morphologically-rich languages, such as Portuguese. Among other text classification tasks, character n-grams revealed to be successful in author attribution [21]. This adds to the simplified pre-processing steps, which require no specific tools or detailed linguistic knowledge.

The exploitation of n-grams resulted in three features – NG1, NG2, NG3 – obtained after computing the cosine similarity of two vectors containing the following information:

- **NG1**: vectors with the binary term-frequency (TF) in which the vocabulary corresponds to the set of n-grams of words, with $n \in \{1,2\}$, in lowercase, stemmed with the Portuguese RSLP stemmer available in the NTLK toolkit[7], after removing stopwords, and considering only n-grams that occur in more than one sentence (document_frequency > 1).

- **NG2**: vector with binary TF for character n-grams, with $n \in \{1,3\}$, within the limits of word boundaries, in lowercase, and considering only n-grams that occur in more than one sentence (document_frequency > 1) and a maximum of 50% of the sentences (max_document_frequency = $0.5 \times \#Dataset\_Sentences$).

- **NG3**: vectors with binary TFs for char n-grams, with $n \in \{1,3\}$, not considering word boundaries, in lowercase, and considering only n-grams that occur in more than one document (document_frequency > 1) and a maximum of 40% of documents (max_document_frequency = $0.4 \times \#Dataset\_Sentences$).

We also followed the current trend of using word embeddings, learned from a large corpus with a neural network, in semantic similarity tasks. For this purpose, we resorted to the NILC embeddings [18], which offer a wide variety of pre-trained embeddings, learned with different models in a large Portuguese corpus, and freely available[8].

---

[7] RSLP stemmer available from `http://www.nltk.org/_modules/nltk/stem/rslp.html`
[8] NILC embeddings available from `http://nilc.icmc.usp.br/embeddings`

More precisely, two different features were computed from the embeddings, both after the conversion of each sentence into a vector computed from the vectors of its tokens. The difference occurs on how this vector is created.

- In the first feature, the sentence vector is obtained from the sum of the token vectors;
- In the second feature, it is computed from the weighted sum of the token vectors, using the TF-IDF value of each token as the weight.

In both, the similarity of each pair of sentences is computed as the cosine similarity between their vectors.

The previous features were initially extracted using different embeddings. The results of using only these features were analysed (see Section 3.5), and only one model of embeddings was used in the final set of features.

3.5 Unsupervised Results

Before moving to a supervised approach, the correlation of some of the extracted features was computed when used alone to predict STS. This would give valuable hints on the relevance of each feature and, at the same time, set baselines. Three groups of features were tested: (i) set similarity combined with semantic networks; (ii) word embeddings; and (iii) n-grams. While the first group has been used since our first approach to ASSIN [4], the second and third were only recently added to our feature set.

3.5.1 Set similarity and LKBs

First, all the combinations of set similarity features were tested in the training collections, with different kinds of normalization (none, stemming and lemmatisation). The previous measures were then tested when combined with the semantic relations in each of the exploited LKBs, as described in Section 3.3. Table 2 shows a selection of the best results at this stage. The best results with Jaccard$^+$ and Cosine$^+$ were obtained with the *Redun3* LKB, and are presented here. Additional results can be found in our previous approach [14], where we also concluded that using all words, instead of just open-class, and not requiring a match of parts-of-speech would improve the correlation ρ. Another conclusion was that stemming would lead to better results than lemmatisation and that using the LKBs could lead only to minor improvements.

Based on those conclusions, the selection of approaches to use in the test collections was narrowed to two, Cosine$^+$ and Jaccard$^+$, on *Redun3*, computed after stemming. Their results are presented in table 3, together with a baseline that computes the cosine of the stems in both sentences of the pair. It is worth noticing that Cosine$^+$ would be the fifth and third best run in ASSIN, respectively for PTPT and PTBR, which corresponds to the fourth and second best system.

3.5.2 N-grams

All the three n-gram features were tested, first in the training, then on the test collection. Results, presented in table 4, are quite surprising, especially for the character n-grams (features NG2 and NG3). The power of these features, even when used alone, would result in technical ties with the second best run for PTPT and with the best run for PTBR, in the official evaluation. As mentioned earlier, character n-grams carry a mix of lexical, syntactic, and even author style content. Without any normalization, different forms of the same word are considered completely different tokens. This is often solved with stemming, which ends up

Table 2 Set similarity results in the training collections.

Normalization	Measure	PTPT		PTBR	
		ρ	MSE	ρ	MSE
None	Jaccard	0.661	1.220	0.587	1.168
None	Cosine	0.664	0.552	0.587	0.591
Stems	Jaccard	0.700	1.140	0.625	0.853
Stems	Cosine	**0.706**	**0.443**	**0.626**	**0.467**
Lemmas	Jaccard	0.695	1.131	0.610	0.921
Lemmas	Cosine	0.698	0.446	0.610	0.484
Stems	Jaccard$^+$	0.717	1.049	0.632	0.778
Stems	Cosine$^+$	**0.721**	**0.388**	**0.631**	**0.453**
Lemmas	Jaccard$^+$	0.709	1.116	0.621	0.843
Lemmas	Cosine$^+$	0.712	0.431	0.620	0.464

Table 3 Test results when semantic networks are exploited, plus the Cosine baseline.

Normalization	Measure	PTPT		PT-PBR	
		ρ	MSE	ρ	MSE
Stems	Jaccard$^+$	0.669	0.723	0.666	0.825
Stems	Cosine$^+$	**0.677**	**0.686**	**0.667**	0.454
(baseline) Stems	Cosine	0.656	0.658	0.653	**0.445**

Table 4 Results for n-gram features in the training and test collections.

| Features | Train | | | | Test | | | |
	PTPT		PTBR		PTPT		PTBR	
	ρ	MSE	ρ	MSE	ρ	MSE	ρ	MSE
NG1	0.664	0.470	0.580	0.537	0.600	0.748	0.600	0.514
NG2	**0.743**	**0.395**	0.685	0.429	**0.696**	0.608	**0.696**	**0.425**
NG3	**0.743**	0.454	**0.688**	**0.483**	0.699	**0.597**	0.695	0.488

considering them equal. So, character n-grams provide a more precise representation of word proximity, because the sets of n-grams for forms of the same word have much in common, but are not equal.

3.5.3   Word embeddings

Though there are many NILC embeddings, we compared only those with 300-sized vectors, a commonly used dimension. As mentioned earlier, two different features were extracted, one relying on the TF to compute the sentence vectors, and another relying on TF-IDF for the same purpose. Table 5 shows the results when using only these features, with the different tested embeddings.

From those results, we decided to use only the word2vec CBOW model, which got the best ρ in the training collection of PTPT, using TF, and the lowest MSE in all the other collections, with TF and TF-IDF. Another option would have been to select fastText SKIP-GRAM,

Table 5 Results for embedding features in the training collections.

Model	Weights	PTPT		PTBR	
		ρ	MSE	ρ	MSE
word2vec CBOW	TF	**0.592**	**0.614**	0.511	**0.755**
word2vec SKIP-GRAM	TF	0.551	1.379	0.491	2.195
fastText CBOW	TF	0.408	1.400	0.350	1.570
fastText SKIP-GRAM	TF	0.566	2.647	**0.521**	2.830
Wang2vec CBOW	TF	0.557	2.336	0.511	2.484
Wang2vec SKIP-GRAM	TF	0.559	2.625	0.508	2.784
GloVe	TF	0.507	2.076	0.444	2.299
word2vec CBOW	TF-IDF	0.609	**0.572**	0.550	**0.682**
word2vec SKIP-GRAM	TF-IDF	0.587	1.073	0.524	1.266
fastText CBOW	TF-IDF	0.451	1.649	0.402	1.772
fastText SKIP-GRAM	TF-IDF	**0.639**	2.393	**0.591**	2.526
Wang2vec CBOW	TF-IDF	0.626	1.951	0.578	2.054
Wang2vec SKIP-GRAM	TF-IDF	0.622	2.280	0.577	2.382
GloVe	TF-IDF	0.528	1.871	0.502	2.036

Table 6 Results for embedding features in the test collections.

Model	Weights	PTPT		PTBR	
		ρ	MSE	ρ	MSE
word2vec CBOW	TF	0.548	1.125	0.538	0.749
word2vec CBOW	TF-IDF	0.555	1.072	0.572	0.665

but the results of this model has always a very high MSE. Table 6 shows the results of the selected model in the test collections.

These results are not much different from those by Hartmann et al. [18], who used all the NILC embeddings in the test collections of ASSIN. It should also be noted that they are lower than all the other unsupervised results here, which shows that, when it comes to STS, this kind of embeddings alone are definitely not enough for achieving high results. Alternative ways for representing sentences as distributional vectors have to be devised in the future. Nevertheless, these two features were included in our feature set, where, combined with the others, should have a positive impact.

4 Learning a model for Portuguese STS

For improving the unsupervised results, the selected features were used together to learn a STS function from each training collection and, later, from both. Here, we describe the learning algorithms used, and report on the training and test results achieved, which beat the best performances in ASSIN to date, thus setting the state-of-the-art of Portuguese STS.

Table 7 Weka setup for the three learning algorithms used.

M5Rules
`weka.classifiers.rules.M5Rules -M 4.0`
RandomSubspace w/ M5
`weka.classifiers.meta.RandomSubSpace -P 0.5 -S 1 -num-slots 1 -I 10 -W`
`weka.classifiers.trees.M5P - -M 4.0`
Gaussian Process w/ RBF Kernel
`weka.classifiers.functions.GaussianProcesses -L 1.0 -N 0 -K`
`"weka.classifiers.functions.supportVector.RBFKernel -G 0.01 -C 250007"`

4.1 Regression Algorithms

Several regression algorithms, provided by the Weka [16] machine learning toolkit, were selected to learn a STS function. Table 7 presents the setup of the three best-performing algorithms, after an exhaustive set of runs. The used algorithms are:

- *M5Rules* [20] generates a decision list for regression problems using a separate-and-conquer strategy. In each iteration, it builds a model tree using the M5 algorithm and turns the "best" leaf into a rule.

- *Random Subspace* [19] is an ensemble learning algorithm that builds a decision tree classifier. It consists of random subspacing regression ensembles composed of multiple trees constructed systematically by pseudo-randomly selected subsets of components of the feature vector.

- Regression algorithm based on *Gaussian Processes* [22], with a Radial Basis Function (RBF) Kernel as the Gaussian function. This implementation is simplified in Weka: it does not apply hyper-parameter-tuning and uses normalization to the target class (similarity value), so the features simplify the choice of a noise level.

4.2 Training and Testing

Each of the selected regression algorithms was used for learning two STS models, for PTPT and for PTBR, based on the respective training collections. Table 8 shows the average training performance with the current set of features (v2) for the three regression algorithms, in a 10-fold cross validation, for PTPT and PTBR. When compared to our previous results (v1.5) [14], in the same table, there are improvements in training.

The learned models were then used for computing STS in the respective test collections. Table 9 shows the test results of the new models, again side-by-side with our previous results, and also with the systems that achieved the best official results, for PTBR and PTPT, in ASSIN, respectively Solo Queue [17] and L2F/INESC-ID [11]. Our current results are clearly better than our best unsupervised results and also than our previous results, which means that the new features had a positive impact. Furthermore, when compared to the best official ASSIN results, there are also improvements in ρ and MSE. More precisely, for PTPT, ρ is 0.02 points higher and MSE is 0.03 points lower than the best, and, for PTBR, ρ is 0.04 higher and MSE is 0.03 points lower. We can thus see these results as the new state-of-the-art of Portuguese STS.

▮ **Table 8** Performance when training in the PTPT and PTBR collections between previous (v1) and present (v2) ASAPP systems.

	Method	PTPT		PTBR	
		ρ	MSE	ρ	MSE
v1.5	M5Rules	0.742	0.472	0.657	0.518
	RandomSubspace	**0.756**	**0.457**	**0.662**	**0.515**
	GaussianProcess	0.739	0.479	0.658	0.520
v2	M5Rules	0.778	0.440	0.723	0.480
	RandomSubspace	**0.784**	**0.432**	**0.723**	**0.479**
	GaussianProcess	0.776	0.444	0.722	0.481

▮ **Table 9** Test results for models trained in the respective training collection compared with the state-of-the-art systems.

	Method	PTPT		PTBR	
		ρ	MSE	ρ	MSE
v1.5	M5Rules	0.703	0.714	0.678	0.411
	RandomSubspace	**0.709**	**0.698**	**0.686**	**0.403**
	GaussianProcess	0.694	0.725	0.683	0.406
v2	M5Rules	0.740	0.590	0.730	0.350
	RandomSubspace	**0.750**	**0.580**	**0.740**	**0.350**
	GaussianProcess	0.740	0.620	0.730	0.350
	Solo Queue [17]	0.700	0.660	**0.700**	**0.380**
	L2F/INESC-ID [11]	**0.730**	**0.610**	-	-

4.3 Training on both collections

Since they are just variants of the same language, instead of training independent models for PTPT and PTBR, we concatenated the training collections and learned new (variant-ignoring) models from the resulting larger collection, which comprised 6,000 pairs. Tables 10 and 11 show, respectively, the training performance of the same learning algorithms on a 10-fold cross-validation in the larger collection, and the results of the new models in each test collection. These are compared with the best official results in ASSIN.

Although with our previous feature set (v1.5) training with a single collection lead to improvements, with the current set, ρ was similar to the one obtained with a collection trained for each variant. Only MSE was lower. Still, this shows that a single model could be used for computing STS in the PTPT and PTBR collection.

5 Concluding Remarks

We have described the most recent developments on ASAPP. In addition to features used in our previous work, which already considered the presence of negations, token, lemma, stem, chunk and named entity overlap, plus semantic relations, new features were added: syntactic dependencies, word and character n-gram similarity, and distributional similarity.

■ **Table 10** Training performance in a collection with both PTPT and PTBR training pairs between previous and present ASAPP systems.

	Method	ρ	MSE
v1.5	M5Rules	0.705	0.493
	RandomSubspace	**0.713**	**0.486**
	GaussianProcess	0.701	0.493
v2	M5Rules	0.756	0.456
	RandomSubspace	**0.760**	**0.451**
	GaussianProcess	0.754	0.459

■ **Table 11** Test results for models trained with both PTPT and PTBR training pairs compared with the state-of-the-art systems.

	Method	PTPT		PTBR	
		ρ	MSE	ρ	MSE
v1.5	M5Rules	0.702	0.648	0.690	0.505
	RandomSubspace	**0.711**	**0.657**	**0.697**	**0.499**
	GaussianProcess	0.691	0.678	0.684	0.509
v2	M5Rules	0.740	0.540	0.730	0.350
	RandomSubspace	**0.750**	**0.540**	**0.740**	**0.340**
	GaussianProcess	0.740	0.560	0.730	0.350
	Solo Queue	0.700	0.660	**0.700**	**0.380**
	L2F/INESC-ID	**0.730**	**0.610**	-	-

Interesting results can be achieved with some of the previous features alone, where we highlight the good performance of character n-grams. Yet, the best results were obtained using all the previous features to learn a STS function from the training collections of ASSIN. Three different regression algorithms were tested for this purpose, and all outperformed the best official results of ASSIN – Pearson ρ of 0.75 and 0.74, MSE of 0.54 and 0.34, respectively for European and Brazilian Portuguese. This means that we can see the approach reported here as the current state-of-the-art of Portuguese STS. Moreover, we have confirmed that a single model, learned from training collections in both variants, obtains very similar results than two different models, each trained and tested on a variant-dependent collection.

Given the Pearson ρ of the human annotation of the ASSIN collections [12] – 0.74 – , trying to improve these results further is probably unrealistic, and possibly not very useful. Nevertheless, there is work to do, especially regarding an analysis of feature relevance, and the integration of all features in a single pipeline, which, until this point, was not our main goal. Although some experiments were reported with each feature alone, this analysis is harder when all the features are combined. For this purpose, the correlation between the features and the similarity scores could be computed to analyse feature relevance; a method such as Principal Component Analysis (PCA) could be applied for feature reduction; and, when possible, the STS functions obtained with the regression algorithms, and the included weights, should be analysed. Identifying the most relevant features should be especially useful for learning more about Portuguese STS and would help us on the integration of all feature extraction methods, hopefully only the most relevant, in a single pipeline.

It should also be stressed that the reported results were obtained in the ASSIN collection. As far as we know, there is currently no other collection with the same kind of annotations in Portuguese, at least freely available and with similar size. In the future, it would be important to test our approach in different collections of Portuguese sentences with STS scores.

─── **References** ───────────────────────────

1 Eneko Agirre, Carmen Banea, Daniel Cer, Mona Diab, Aitor Gonzalez-Agirre, Rada Mihalcea, German Rigau, and Janyce Wiebe. Semeval-2016 task 1: Semantic textual similarity, monolingual and cross-lingual evaluation. In *10th Intl. Workshop on Semantic Evaluation (SemEval)*, pages 497–511, 2016.

2 Eneko Agirre, Mona Diab, Daniel Cer, and Aitor Gonzalez-Agirre. Semeval-2012 task 6: A pilot on semantic textual similarity. In *6th Intl. Workshop on Semantic Evaluation*, pages 385–393, 2012.

3 Ana Alves, Adriana Ferrugento, Mariana Lourenço, and Filipe Rodrigues. ASAP: Automatic semantic alignment for phrases. In *8th Intl. Workshop on Semantic Evaluation (SemEval)*, pages 104–108, 2014.

4 Ana Alves, Ricardo Rodrigues, and Hugo Gonçalo Oliveira. Asapp: Alinhamento semântico automático de palavras aplicado ao português. *Linguamática*, 8(2):43–58, 2016.

5 Ana Alves, David Simões, Hugo Gonçalo Oliveira, and Adriana Ferrugento. ASAP-II: From the alignment of phrases to textual similarity. In *9th Intl. Workshop on Semantic Evaluation (SemEval 2015)*, pages 184–189, 2015.

6 Anabela Barreiro. Port4NooJ: an open source, ontology-driven portuguese linguistic system with applications in machine translation. In *Intl. NooJ Conference (NooJ'08)*, 2010.

7 Anderson Pinheiro Cavalcanti, Rafael Ferreira Leite de Mello, Máverick André Dionísio Ferreira, Vitor Belarmino Rolim, and João Vitor Soares Tenório. Statistical and semantic features to measure sentence similarity in Portuguese. In *Proceedings of 6th Brazilian Conference on Intelligent Systems*, pages 342–347, 2017.

8 Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. Semeval-2017 task 1: Semantic textual similarity multilingual and crosslingual focused evaluation. In *11th Intl. Workshop on Semantic Evaluation (SemEval)*, pages 1–14, 2017. `doi:10.18653/v1/S17-2001`.

9 Bento C. Dias-da-Silva. Wordnet.Br: An exercise of human language technology research. In *3rd Intl. WordNet Conf. (GWC)*, pages 301–303, 2006.

10 Christiane Fellbaum, editor. *WordNet: An Electronic Lexical Database (Language, Speech, and Communication)*. The MIT Press, 1998.

11 Pedro Fialho, Ricardo Marques, Bruno Martins, Luísa Coheur, and Paulo Quaresma. INESC-ID@ASSIN: Medição de similaridade semântica e reconhecimento de inferência textual. *Linguamática*, 8(2):33–42, 2016.

12 Erick Fonseca, Leandro Santos, Marcelo Criscuolo, and Sandra Aluísio. Visão geral da avaliação de similaridade semântica e inferência textual. *Linguamática*, 8(2):3–13, 2016.

13 Hugo Gonçalo Oliveira. Comparing and combining Portuguese lexical-semantic knowledge bases. In 6th *Symposium on Languages, Applications and Technologies (SLATE)*, pages 16:1–16:14, 2017.

14 Hugo Gonçalo Oliveira, Ana Oliveira Alves, and Ricardo Rodrigues. Gradually improving the computation of semantic textual similarity in Portuguese. In *18th EPIA Conference on Artificial Intelligence*, volume 10423, pages 841–854, 2017. `doi:10.1007/978-3-319-65340-2_68`.

15 Hugo Gonçalo Oliveira, Diana Santos, Paulo Gomes, and Nuno Seco. PAPEL: A dictionary-based lexical ontology for Portuguese. In *8th Intl. Conf. Computational Processing of the Portuguese Language (PROPOR)*, volume 5190, pages 31–40, 2008.

16   Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian Witten. The WEKA data mining software: An update. *SIGKDD Explorations*, 11(1):10–18, 2009. `doi:10.1145/1656274.1656278`.

17   Nathan Hartmann. Solo Queue at ASSIN: Combinando abordagens tradicionais e emergentes. *Linguamática*, 8(2):59–64, 2016.

18   Nathan S. Hartmann, Erick R. Fonseca, Christopher D. Shulby, Marcos V. Treviso, Jéssica S. Rodrigues, and Sandra M. Aluísio. Portuguese word embeddings: Evaluating on word analogies and natural language tasks. In *11th Brazilian Symposium in Information and Human Language Technology (STIL)*, 2017.

19   Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.

20   Geoffrey Holmes, Mark Hall, and Eibe Frank. Generating rule sets from model trees. In *12th Australian Joint Conf. on Artificial Intelligence*, pages 1–12, 1999.

21   Moshe Koppel, Jonathan Schler, and Shlomo Argamon. Authorship attribution in the wild. *Languages Resoursces Evaluation*, 45(1):83–94, 2011. `doi:10.1007/s10579-009-9111-2`.

22   David Mackay. Introduction to Gaussian Processes. In *Neural Networks and Machine Learning*. Springer, 1998.

23   Marco Marelli, Luisa Bentivogli, Marco Baroni, Raffaella Bernardi, Stefano Menini, and Roberto Zamparelli. Semeval-2014 task 1: Evaluation of compositional distributional semantic models on full sentences through semantic relatedness and textual entailment. In *8th Intl. Workshop on Semantic Evaluation (SemEval)*, pages 1–8, 2014.

24   Erick Maziero, Thiago Pardo, Ariani Felippo, and Bento Dias-da-Silva. A Base de Dados Lexical e a Interface Web do TeP 2.0 - Thesaurus Eletrônico para o Português do Brasil. In *VI Workshop em Tecnologia da Informação e da Linguagem Humana (TIL)*, pages 390–392, 2008.

25   Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *Workshop track of the Intl. Conf. on Learning Representations (ICLR)*, 2013.

26   Valeria Paiva, Alexandre Rademaker, and Gerard Melo. OpenWordNet-PT: An open Brazilian Wordnet for reasoning. In *24th Intl. Conf. on Computational Linguistics (COLING)*, 2012.

27   Vladia Pinheiro, Vasco Furtado, and Adriano Albuquerque. Semantic textual similarity of portuguese-language texts: An approach based on the semantic inferentialism model. In *11th Conf. on the Computational Processing of the Portuguese Language (PROPOR)*, pages 183–188, 2014. `doi:10.1007/978-3-319-09761-9_19`.

28   Ricardo Rodrigues, Hugo Gonçalo-Oliveira, and Paulo Gomes. NLPPort: A pipeline for portuguese nlp. In *7th Symposium on Languages, Applications and Technologies (SLATE)*, pages 18:1–18:9, 2018.

29   Barbara Rychalska, Katarzyna Pakulska, Krystyna Chodorowska, Wojciech Walczak, and Piotr Andruszkiewicz. Samsung Poland NLP team at SemEval-2016 task 1: Necessity for diversity; combining recursive autoencoders, wordnet and ensemble methods to measure semantic similarity. In *10th Intl. Workshop on Semantic Evaluation (SemEval)*, pages 602–608, 2016.

30   Alberto Simões and Xavier Guinovart. Bootstrapping a Portuguese wordnet from Galician, Spanish and English wordnets. In *Advances in Speech and Language Technologies for Iberian Languages*, volume 8854 of *LNCS*, pages 239–248, 2014.

31   Alberto Simões, Álvaro Sanromán, and José Almeida. Dicionário-Aberto: A source of resources for the Portuguese language processing. In *10th Intl. Conf. on the Computational Processing of the Portuguese Language (PROPOR)*, volume 7243, pages 121–127, 2012.

32   Junfeng Tian, Zhiheng Zhou, Man Lan, and Yuanbin Wu. Ecnu at semeval-2017 task 1: Leverage kernel-based traditional nlp features and neural networks to build a universal model for multilingual and cross-lingual semantic textual similarity. In *11th Intl. Workshop on Semantic Evaluation (SemEval)*, pages 191–197, 2017. `doi:10.18653/v1/S17-2028`.

Evaluation of Distributional Models with the Outlier Detection Task

Pablo Gamallo

Centro de Investigación en Tecnoloxías da Información (CiTIUS)
University of Santiago de Compostela, Galiza
pablo.gamallo@usc.es
 https://orcid.org/0000-0002-5819-2469

── Abstract ──

In this article, we define the outlier detection task and use it to compare neural-based word embeddings with transparent count-based distributional representations. Using the English Wikipedia as text source to train the models, we observed that embeddings outperform count-based representations when their contexts are made up of bag-of-words. However, there are no sharp differences between the two models if the word contexts are defined as syntactic dependencies. In general, syntax-based models tend to perform better than those based on bag-of-words for this specific task. Similar experiments were carried out for Portuguese with similar results. The test datasets we have created for outlier detection task in English and Portuguese are released.

1 Introduction

Intrinsic evaluations of distributional models are based on word similarity tasks. The most popular intrinsic evaluation is to calculate the correlation between the similarity scores obtained by a system using a word vector representation and a gold standard of human-assigned similarity scores. Recent critics to intrinsic evaluation claim that inter-annotator agreement at the word similarity task is considerably lower compared to other tasks such as document classification or textual entailment [3]. To overcome this problem, Camacho-Collados and Navigli [7] propose an alternative evaluation relying on the outlier detection task, which tests the capability of vector space models to create semantic clusters. More precisely, given a set of words, for instance *car*, *train*, *bus*, *apple*, *bike*, the goal of the task is to identify the word that does not belong to a semantically homogeneous group. In this case, the outlier is *apple*, which is not a vehicle. The main advantage of this task is to provide a clear gold standard with, at least, two properties: high inter-annotator agreement and easy method to increase the test size by adding new groups.

On the other hand, recent works comparing count-based word distributions with word embeddings (i.e., dense representations obtained with neural networks) to compute word similarity show mixed results. Some claim that embeddings outperform transparent and

explicit count-based models [2, 24], while others show that there are no significant differences between them [20, 23], in particular if the hyperparameters are configured and set in a similar way [22]. Other works report heterogeneous results since the performance of the two models varies according to the task to be evaluated [5, 9, 14, 19].

In this paper, we make use of the outlier detection task defined in Camacho-Collados and Navigli [7] to compare different types of embeddings and count-based word representations. In particular, we compare the use of bag-of-words and syntactic dependencies in both embeddings and count-based models. We observed that there are no clear differences if the two models rely on syntactic dependencies; yet embeddings seem to perform better than transparent models when the dimensions are made up of bag-of-words. In addition, we contribute to enlarge the test dataset by adding more groups of semantically homogeneous words and outliers (50% larger), and by manually translating the expanded dataset to Portuguese.

Next, we will describe the outlier datasets (Section 2), a count-based model with filtering (Section 3), and several experiments to compare different distributional models using the outlier datasets (Section 4). Conclusions will be addressed in Section 5.

2   The Datasets for Outlier Detection

For the outlier detection task, Camacho-Collados and Navigli [7] provided the *8-8-8* dataset[1], which consists of eight different topics each containing a cluster of eight words and eight outliers which do not belong to the given topic. For instance, one of the topics is "European football teams", which was defined with a set of eight nouns:

> FC Barcelona, Bayern Munich, Real Madrid, AC Milan, Juventus, Atletico Madrid, Chelsea, Borussia Dortmund

and a set of eight outliers:

> Miami Dolphins, McLaren, Los Angeles Lakers, Bundesliga, football, goal, couch, fridge

In order to expand the number of examples, two annotators were asked to create four new topics, and for each topic to provide a set of eight words belonging to the chosen topic, and a set of eight heterogeneous outliers. One of them proposed all the words in less than 15 minutes, and the other annotator just agreed with all the decisions taken by the first one. This 100% inter-annotator agreement is in contrast with the low inter-annotator levels achieved in the standard word similarity datasets, for instance in WordSim353 [10] the average pair-wise Spearman correlation among annotators is merely 0.61. The new expanded dataset, called *12-8-8*, contains 12 topics, each made up of 8+8 topic words and outliers. In addition, in order to simplify the comparison with systems that do not identify multiwords, we also changed the multi-words found in the *8-8-8* dataset by one-word terms denoting similar entities. For instance: the terms "Celtic" and "Betis" were used instead of "Atletico Madrid" and "Bayern Munich", all referring to football teams. The *12-8-8* dataset contains 50% more test examples than the original one. Finally, we also created a new dataset by translating *12-8-8* into Portuguese.

In Camacho-Collados and Navigli [7], the outlier detection task is defined on the basis of a generic concept of *compactness score*. Here, we propose to define a more specific *compactness score* by assuming that the similarity coefficient is symmetrical (e.g. Cosine). Intuitively,

---

[1] `http://lcl.uniroma1.it/outlier-detection/`

given a set of 9 words consisting of 8 words belonging to the same group and one outlier, the *compactness score* of each word of the set is the result of averaging the pair-wise similarity scores of the target word with the other members of the set.

Formally, given a set of words $W = w_1, w_2, ..., w_n, w_{n+1}$, where $w_1, w_2, ..., w_n$ belongs to the same cluster and w_{n+1} is the outlier, we define the compactness score $c(w)$ of a word $w \in W$, and assuming a symmetrical similarity coefficient *sim*, as follows:

$$c(w) = \frac{1}{n} \sum_{\substack{w_i \in W \\ w \neq w_i}} sim(w, w_i) \tag{1}$$

An outlier is correctly detected if the compactness score of the outlier word is lower than the scores of the cluster words. So, *accuracy* measures how many outliers were correctly detected by the system divided by the number of total detections: 12x8 in our *12-8-8* dataset. Camacho-Collados and Navigli [7] also define Outlier Position Percentage (OPP) which takes into account the position of the outlier in the list of $n + 1$ words ranked by the compactness score, which ranges from 0 to n (position 0 indicates the lowest overall score among all words in W, and position n indicates the highest overall score).

3 A Filtered-Based Distributional Model

The outlier datasets will be used to compare count-based distributional models with embeddings. The count-based model we propose is based on a filtering approach and dependency contexts.

As co-occurrence matrices representing context distribution are sparse, most entries of a sparse matrix are zeros that do not need to be stored explicitly. In fact, highly dispersed matrices are computationally easy to work with [9, 16]. A possible storage mode for a sparse matrix is a hash table where keys are word-context pairs with non-zero values [16]. To reduce the number of keys in a hash table representing word-context co-occurrences, we apply a technique to filter out contexts by relevance [6]. The compressing technique consists in computing an *informativeness* measure -e.g., loglikelihood [8]- between each word and their contexts. For each word, only the R (relevant) contexts with highest loglikelihood scores are kept in the hash table. R is a global, arbitrarily defined constant whose usual values range from 10 to 1000 [4, 26]. In short, we keep the R most relevant contexts for each target word. Context filtering allows us to dramatically reduce the context space and, unlike embeddings, makes the word model transparent, fully interpretable and easily readable by humans.

Syntactic-based word contexts can be derived from the dependency relations the words participate in (e.g. subject, direct object, modifier). To extract contexts from dependencies, we use the co-compositional methodology defined in Levy and Goldberg [21] and Gamallo [15]. Notice that syntax-based models are fully interpretable as each dimension is an explicit lexico-syntactic context.

4 Experiments and Evaluation

We performed three experiments. The first one using the original *8-8-8* dataset. The second one comparing more approaches against the expanded *12-8-8* dataset. And the third one comparing the best approaches of the previous experiments using the Portuguese *12-8-8* dataset.

Table 1 Outlier Position Percentage (OPP) and Accuracy of different word models on the 8-8-8 outlier detection dataset using Wikipedia.

System	Strategy	OPP	Accuracy
Dep500	count+syntax	**97.3**	**90.6**
CBOW	embed+bow	95.3	73.4
Skip-Gram	embed+bow	93.8	70.3
Glove	embed+bow	91.8	56.3
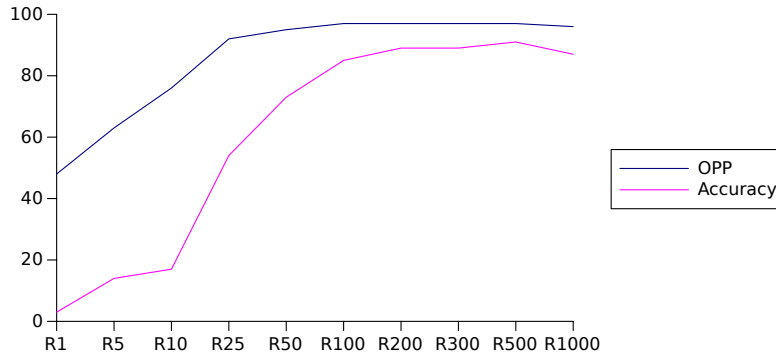
Figure 1 Accuracy and OPP of our count-based strategy across different filtering thresholds: from $R = 1$ to $R = 1000$.

4.1 The 8-8-8 Dataset

The goal of the experiment is to compare the basic count-based model defined in the previous section (3) with the results obtained by different versions of embeddings, which were reported in Camacho-Collados and Navigli [7].

Table 1 shows the results obtained by the count-based strategy we developed, *Dep500*, which is a count-based model with contexts represented as syntactic dependencies and a relevance filter $R = 500$. The contexts of the model were built by making use of a rule-based dependency parser, DepPattern [13]. The method outperforms the results obtained by three standard embedding models: the CBOW and Skip-Gram models of Word2Vec [24] and GloVe [28], which are based on bag-of-words contexts[2], and whose results were reported in Camacho-Collados and Navigli [7]. The dimensionality of the dense vectors was set to 300 for the three embedding models. Context-size 5 for CBOW and 10 for Skip-Gram and GloVe; hierarchical softmax for CBOW and negative sampling for Skip-Gram and GloVe. In all experiments, the corpus used to build the vector space was the 1.7B-tokens English Wikipedia (dump of November 2014).

The growth curve depicted in Figure 1 shows the evolution of accuracy and OPP over different R values. We can observe that the curve stabilizes at $R = 200$ and starts going down before $R = 1000$. It means that small count-based distributional models with relevant contexts perform better than large models made up of many noisy syntactic contexts.

---

[2] We use *bow* to refer to linear bag-of-word contexts, which must be distinguished from CBOW (continuous bag-of-words) [22]

4.2 The 12-8-8 Expanded Dataset

The main goal of the next experiments is to use the outlier detection task to compare the performance of different types of dependency parsers (rule-based and transition-based) to build both count-based distributions and neural embeddings. Additionally, we also compare the use of syntactic dependencies and bag-of-words in the same task. We require a dataset without multiwords since some of the tools we used for building distributions only tokenize unigrams. For this purpose, we defined the following six strategies:

Count₁ A count-based model with rule-based dependencies.

Count₂ A count-based model with transition-based dependencies.

Count₃ A count-based model with bag-of-words.

Emb₁ Embeddings with rule-based dependencies.

Emb₂ Embeddings with transition-based dependencies.

Emb₃ Embeddings with bag-of-words.

The three count-based models were built with the filter $R = 300$, whereas the dimensionality of the three embeddings was set to 300. The three embeddings were based on the continuous *skip-gram* neural embedding model [24], with negative-sampling parameter set at 15. The two bag-of-words models were generated using a window of size 10: 5 words to the left and 5 to the right of the target word. Both Emb_2 and Emb_3 are the models described in Levy and Goldberg [21], which are publicly available[3]. To create the dependency-based models, the corpus was parsed with a very specific configuration of the arc-eager transition-based dependency parser described in [17][4]. The performance of the parser for English is about 89% UAS (unlabeled attachment score) obtained on the CoNLL 2007 dataset. To build Emb_2, we made use of `word2vecf`[5], a modified version of word2vec , which is suited to build embeddings with syntactic dependencies [21]. Rule-based dependencies were obtained using DepPattern [13].

Even though the strategies are very different using very different software, we tried to use the same hyperparameters in order to minimize differences that are not due to the word models themselves. As Levy and Goldberg [23] suggest, much of the difference between vectorial models are due to certain system design choices and hyperparameter optimizations (e.g., subsampling frequent words, window size, etc.) rather than to the algorithms themselves. The authors revealed that seemingly minor variations in external parameters can have a large impact on the success of word representation methods.

Table 2 shows the results obtained on the *12-8-8* dataset by the six models built from the English Wikipedia. The four syntax-based methods (with rules or transitions, count-based or embeddings) give very similar scores. However, they tend to perform better than those based on bag-of-words (as in the previous experiment in Subsection 4.1). This is in accordance with a great number of previous works which evaluate and compare syntactic contexts (usually dependencies) with bag-of-words techniques [11, 12, 18, 21, 25, 27, 29]. All of them state that syntax-based methods outperform bag-of-words techniques, in particular when the objective is to compute semantic similarity between functional (or paradigmatic) equivalent words, such as detection of co-hyponym/hypernym word relations. By contrast, *bow*-based models tend to perform better in tasks oriented to identify semantic relatedness

---

[3] `https://levyomer.wordpress.com/2014/04/25/dependency-based-word-embeddings/`

[4] We are very grateful to the authors for sending us the English Wikipedia syntactically analyzed with their parser.

[5] `https://bitbucket.org/yoavgo/word2vecf`

▨ **Table 2** Outlier Position Percentage (OPP) and Accuracy of different word models on the 12-8-8 outlier detection dataset using Wikipedia.

System	Strategy	OPP	Accuracy
$Count_1$	rules	**94.92**	75.0
$Count_2$	transitions	93.48	71.87
$Count_3$	bow	86.71	60.41
Emb_1	rules	93.09	**76.04**
Emb_2	transitions	94.27	72.91
Emb_3	bow	93.88	69.79

▨ **Table 3** Outlier Position Percentage (OPP) and Accuracy of two distributional models on the 12-8-8 outlier detection dataset using Portuguese Wikipedia.

System	Strategy	OPP	Accuracy
$Count_1$	rules	**91.40**	**48.95**
Emb_1	rules	84.375	39.58

and analogies. We may conclude the following: First, the outlier detection task is suited to search for similarity and not for semantic relatedness [1], and second, the type of context (dependency-based or bag-of-words) is more determinant than the type of model (count-based or embeddings) for that task. Finally, embeddings clearly outperform count-based representations when the contexts are defined with bag-of-words (see score of Emb_3 against $Count_3$ in Table 2).

4.3 Portuguese 12-8-8 Dataset

The 12-8-8 Expanded Dataset was translated into Portuguese in order to make new tests in this language. The Portuguese experiments were carried out with the two best strategies, according to the previous experiments: count-based model with rule-based dependencies ($Count_1$) and embeddings with rule-based dependencies (Emb_1). As in the previous experiment, the count-based model was built with the filter $R = 300$, whereas the dimensionality of the embeddings was set to 300. The latter was implemented with *skip-gram* and negative-sampling parameter set at 15. Table 3 shows the results obtained on the Portuguese *12-8-8* dataset by the two models evaluated.

In these experiments, the count-based strategy clearly outperforms embeddings. This may be partially explained by the fact that the Portuguese Wikipedia is almost 10 times smaller than the English one, and neural networks require large corpus to make better predictions.

5 Conclusions

We have used the outlier detection task for intrinsic evaluation of distributional models in English and Portuguese. Unlike standard gold-standards for similarity tasks, the construction of datasets for outlier detection requires low human cost with very high inter-annotator agreement. Our very preliminary experiments show that the use of syntactic contexts in traditional count-based models and embeddings leads the two models to similar performance on the outlier detection task, even if count-based strategies seem to perform better with less training corpus.

In future work, we intend to develop outlier detection datasets for many other languages in order to make it possible multilingual word similarity evaluation. The software required to build the count-based models as well as the *12-8-8* datasets are publicly available[6].

References

1 Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Paşca, and Aitor Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 19–27, 2009.

2 Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 238–247, 2014.

3 Miroslav Batchkarov, Thomas Kober, Jeremy Reffin, Julie Weeds, and David Weir. A critique of word similarity as a method for evaluating distributional semantic models. In *Proceedings of the ACL Workshop on Evaluating Vector Space Representations for NLP*, pages 7–12, 2016.

4 Biemann, C., and Riedl M. Text: Now in 2D! A framework for lexical expansion with contextual similarity. *Journal of Language Modelling*, 1(1):55–95, 2013.

5 William Blacoe and Mirella Lapata. A comparison of vector-based representations for semantic composition. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 546–556, 2012.

6 Stefan Bordag. A comparison of co-occurrence and similarity measures as simulations of context. In *Computational Linguistics and Intelligent Text Processing (CICLing)*, pages 52–63, 2008.

7 José Camacho-Collados and Roberto Navigli. Find the word that does not belong: A framework for an intrinsic evaluation of word vector representations. In *Proceedings of the ACL Workshop on Evaluating Vector Space Representations for NLP*, pages 43–50, 2016.

8 Ted Dunning. Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1):61–74, 1993.

9 Manaal Faruqui and Chris Dyer. Non-distributional word vector representations. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, pages 464–469, 2015.

10 Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing search in context: the concept revisited. *ACM Transactions on Information Systems*, 20(1):116–131, 2002.

11 Pablo Gamallo. Comparing window and syntax based strategies for semantic extraction. In *Computational processing of the Portuguese language*, pages 41–50, 2008.

12 Pablo Gamallo. Comparing different properties involved in word similarity extraction. In *14th Portuguese Conference on Artificial Intelligence (EPIA'09)*, pages 634–645, 2009.

13 Pablo Gamallo. Dependency parsing with compression rules. In *Proceedings of the 14th International Workshop on Parsing Technology (IWPT)*, pages 107–117, 2015.

14 Pablo Gamallo. Comparing explicit and predictive distributional semantic models endowed with syntactic contexts. *Language Resources and Evaluation*, 51(3):727–743, 2017.

15 Pablo Gamallo, Alexandre Agustini, and Gabriel Lopes. Clustering syntactic positions with similar semantic requirements. *Computational Linguistics*, 31(1):107–146, 2005.

---

[6] `http://gramatica.usc.es/~gamallo/prototypes/Word2Model.tgz`

16   Pablo Gamallo and Stefan Bordag. Is singular value decomposition useful for word simalirity extraction. *Language Resources and Evaluation*, 45(2):95–119, 2011.

17   Yoav Goldberg and Joakim Nivre. A dynamic oracle for arc-eager dependency parsing. In *24th International Conference on Computational Linguistics Proceedings of the Conference (COLING)*, pages 959–976, 2012.

18   Gregory Grefenstette. Evaluation techniques for automatic semantic extraction: Comparing syntactic and window-based approaches. In *Workshop on Acquisition of Lexical Knowledge from Text (SIGLEX)*, pages 205–216, 1993.

19   Eric Huang, Richard Socher, and Christopher Manning. Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, pages 873–882, 2012.

20   Rémi Lebret and Ronan Collobert. Rehabilitation of count-based models for word vector representations. In *Computational Linguistics and Intelligent Text Processing (CICLing)*, volume 9041, pages 417–429, 2015.

21   Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 302–308, 2014.

22   Omer Levy and Yoav Goldberg. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning (CoNLL)*, pages 171–180, 2014.

23   Omer Levy, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225, 2015.

24   Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

25   Sebastian Padó and Mirella Lapata. Dependency-based construction of semantic space models. *Computational Linguistics*, 33(2):161–199, 2007.

26   Muntsa Padró, Marco Idiart, Aline Villavicencio, and Carlos Ramisch. Nothing like good old frequency: Studying context filters for distributional thesauri. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 419–424, 2014.

27   Yves Peirsman, Kris Heylen, and Dirk Speelman. Finding semantically related words in Dutch. Co-occurrences versus syntactic contexts. In *CoSMO Workshop*, pages 9–16, 2007.

28   Jeffrey Pennington, Richard Socher, and Christopher D. Manning. GloVe: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

29   Violeta Seretan and Eric Wehrli. Accurate collocation extraction using a multilingual parser. In *21st International Conference on Computational Linguistics*, pages 953–960, 2006.

Extending the Galician Wordnet Using a Multilingual Bible Through Lexical Alignment and Semantic Annotation

Alberto Simões

Applied Artificial Intelligence Lab (2Ai Lab)
Instituto Politécnico do Cávado e do Ave, Barcelos, Portugal
asimoes@ipca.pt
https://orcid.org/0000-0001-6961-2660

Xavier Gómez Guinovart[1]

Galician Language Technology and Applications (TALG Group)
Universidade de Vigo, Galiza, Spain
xgg@uvigo.gal
https://orcid.org/0000-0001-9961-6953

Abstract

In this paper we describe the methodology and evaluation of the expansion of Galnet – the Galician wordnet – using a multilingual Bible through lexical alignment and semantic annotation. For this experiment we used the Galician, Portuguese, Spanish, Catalan and English versions of the Bible. They were annotated with part-of-speech and WordNet sense using FreeLing. The resulting synsets were aligned, and new variants for the Galician language were extracted. After manual evaluation the approach presented a 96.8% accuracy.

1 Introduction

WordNet [15, 7] is the key lexical resource in many language applications. While contested by some researchers on the grounds of its fine-grained word senses, it is, indubitably, the most used and replicated lexical knowledge base. For many languages there is, at least, one project trying to build a similar resource.

The main problem is that most of these projects are unable to get enough funding to develop such a large resource by the hand of lexicographers and linguists. This is a bigger problem for under-resourced languages, like Galician. Therefore, these projects employ algorithms to obtain new variants and organise them in synsets, either using complete automatic processes or semi-automatic approaches, where the candidate variants extracted by the algorithms must be manually validated.

Our contribution is another method to obtain candidate variants, applied to the Galician wordnet (Galnet), using other languages wordnets (English, Catalan, Spanish and Portuguese) together with a sentence-aligned multilingual Bible.

To present it, this paper is organised as follows: Section 2 presents the two main resources used for this experiment: Galnet, the Galician wordnet, and the CLUVI multilingual Bible; Section 3 refers to other experiments on creating or enlarging lexical ontologies and their obtained results; in Section 4 the algorithm is explained with some examples; the results obtained by this experiment are then analysed in Section 5, where the different kinds of errors found are detailed; finally, Section 6 draws some conclusions on the methodology used and its applicability to other languages.

2   Language resources

This section presents the two main language resources used in this experiment. First, Galnet, the wordnet for Galician, which is the target for the inclusion of new variants obtained by this experiment. Then, the CLUVI parallel Bible, which will be used as a semantically tagged parallel corpus for the extraction of the new variant candidates.

2.1   WordNet and Galnet

WordNet is a lexical database of the English language, organised as a semantic network where the nodes are concepts represented as sets of synonyms and the links between nodes are semantic relations between lexical concepts. These nodes contain nouns, verbs, adjectives and adverbs grouped by synonymy. In WordNet terminology, a set of synonyms is called a *synset*. The term *variant* was coined during the EuroWordNet project [30] to refer to each (lemmatised) synonym in a synset, which is considered a lexical variant of the same concept. Thus, each synset represents a distinct lexicalised concept and includes all the synonymous variants of this concept. Additionally, each synset may contain a brief definition or gloss, which is common to every variant in the synset, and, in some cases, one or more examples of the use of the variants in context.

In the WordNet model of lexical representation, the synsets are linked by means of lexical-semantic relations. Some of the most frequent relations represented in WordNet are hypernymy/hyponymy and holonymy/meronymy for nouns; antonymy and quasi-synonymy (or semantic similarity) for adjectives; antonymy and derivation for adverbs; and entailment, hypernymy/hyponymy, cause and antonymy for verbs.

Galnet [9] is an open wordnet for Galician, aligned with an interlingual index (ILI) generated from the English WordNet 3.0, following the expand model [30] for the creation of new wordnets, where the variants associated with the Princeton WordNet synsets are translated using different strategies. Galnet can be searched via its own dedicated web interface[2] and can be downloaded in RDF and LMF formats[3].

Galnet is part of the Multilingual Central Repository (MCR)[4], a database that currently integrates wordnets from six different languages (English, Spanish, Catalan, Galician, Basque and Portuguese) using WordNet 3.0 as ILI [12]. Table 1 provides the number of synsets and variants for the different languages gathered in this repository, and their percentage of development with respect to the English WordNet.

---

[2] `http://sli.uvigo.gal/galnet/`
[3] `http://sli.uvigo.gal/download/SLI_Galnet/`
[4] `http://adimen.si.ehu.es/web/MCR/`

▬ **Table 1** Current coverage of wordnets in MCR.

	English (WordNet 3.0)		Galician (Galnet 3.0.26)	
	variants	**synsets**	**variants**	**synsets**
Nouns	146,312	82,115	46,972	31,356
Verbs	25,047	13,767	7,247	3,214
Adjectives	30,002	18,156	10,423	6,375
Adverbs	5,580	3,621	1,651	1,079
Total	206,941	117,659	66,293	42,024
%	100%	100%	32,0%	35%
	Spanish (MCR 2016)		**Portuguese (MCR 2016)**	
	variants	**synsets**	**variants**	**synsets**
Nouns	101.027	55.227	17.125	10.047
Verbs	20.953	9.541	8.360	3.786
Adjectives	20.938	12.373	6.330	3.581
Adverbs	3.583	1.854	789	528
Total	146.501	78.995	32.604	17.942
%	70,8%	67,1%	15,8%	15,2%
	Catalan (MCR 2016)		**Basque (MCR 2016)**	
	variants	**synsets**	**variants**	**synsets**
Nouns	73,810	46,917	40,420	26,710
Verbs	14,619	6,349	9,469	3,442
Adjectives	11,212	6,818	148	111
Adverbs	1,152	872	0	0
Total	100,793	60,956	50,037	30,263
%	48,7%	51,8%	24,2%	25,7%

2.2 The CLUVI multilingual Bible

The CLUVI Corpus[5] is an open collection of human-annotated sentence-level aligned parallel corpora, originally designed to cover specific areas of the contemporary Galician language in relation to other languages. With over 47 million words, the CLUVI collection currently comprises twenty-one parallel corpora in nine specialised registers or domains (fiction, computing, popular science, biblical texts, law, consumer information, economy, tourism, and film subtitling) and different language combinations with Galician, Spanish, English, French, Portuguese, Catalan, Italian, Basque, German, Latin and Chinese.

The CLUVI 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. When the term search is a lemma in a language, the result texts could include WordNet-based suggestions on its lexical equivalences in the translation languages using colour codes. In addition, the legal section of the CLUVI corpus, a subset of Galician-Spanish legal texts with 6.5 million words, can be freely downloaded with CC BY-NC-SA 3.0 license[6].

---

[5] `http://sli.uvigo.gal/CLUVI/`
[6] `http://hdl.handle.net/10230/20051`

At the moment, the CLUVI is the parallel corpus that contains the largest number and the most varied thematic range of translations from/to the Galician language. Galician texts present in the CLUVI collection sum up to about 12,000,000 words, which means a quarter of the total of the tokens in the corpus for all the languages and domains of translation. By way of comparison, other parallel corpora that currently facilitate access to translations to the Galician language are the collection of downloadable corpora from OPUS Project[7] [29] and from the Per-Fide Corpus[8] [3]. On the one hand, the OPUS collection provides Galician translated texts, mainly from English, taken from the web and automatically aligned at sentence level. Galician texts in OPUS total around 7,600,000 tokens, coming mainly from the localisation of the Linux operating system environment (Gnome, KDE4 and Ubuntu). On the other hand, the Per-Fide collection includes Portuguese-Galician software localisation parallel texts derived from the OPUS Project with about 400,000 tokens in Galician.

The multilingual Bible built for the CLUVI Corpus aligns the translations of the biblical texts in thirteen linguistic variants – Latin, Galician, Brazilian Portuguese, European Portuguese, Catalan, French, Italian, Spanish, English, German, Basque, Simplified Chinese and Traditional Chinese – , with a total of 31,279 translation units, generally equivalent to the Bible verses, and 7,481,611 words: 535,423 words in Latin, 656,998 words in Galician, 770,410 words in Brazilian Portuguese, 662,853 words in European Portuguese, 723,194 words in Catalan, 719,229 words in French, 674,795 words in Italian, 706,125 words in Spanish, 759,824 words in English, 701,279 words in German, 505,043 words in Basque, 31,308 words in Simplified Chinese and 35,130 words in Traditional Chinese. The CLUVI multilingual Bible collects the translations of all the books shared by the Western biblical canon (Protestant, Lutheran, Anglican and Roman Catholic traditions), including the Old Testament (Genesis, Exodus, Leviticus, Numbers, Deuteronomy, Joshua, Judges, Ruth, 1 and 2 Samuel, 1 and 2 Kings, 1 and 2 Chronicles, Ezra, Nehemiah, Esther, Job, Psalms, Proverbs, Ecclesiastes, Song of Songs, Isaiah, Jeremiah, Lamentations, Ezekiel, Daniel, Hosea, Joel, Amos, Obadiah, Jonah, Micah, Nahum, Habakkuk, Zephaniah, Haggai, Zechariah and Malachi) and the New Testament (Matthew, Mark, Luke, John, Acts, Romans, 1 and 2 Corinthians, Galatians, Ephesians, Philippians, Colossians, 1 and 2 Thessalonians, 1 and 2 Timothy, Titus, Philemon, Hebrews, James, 1 and 2 Peter, 1, 2 and 3 John, Jude and Revelation).

There have been some other projects focused on the creation of an aligned multilingual Bible corpus for linguistic research, such as [24] and [6]. However, up to now, the CLUVI multilingual Bible represents the only available parallel version of the Scriptures which includes a Galician translation.

3 Related work

The *expand model* mentioned above and followed by Galnet for the creation of the Galician wordnet has also been used in the development of the wordnets for Italian [18], Indonesian [21], Hungarian [14], Croatian [22] French – WOLF [25] and WoNeF [20] wordnets – and Kurdish [2]. The same approach has been taken in the MCR framework for the creation of the wordnets of Spanish [4], Catalan [5], Basque [19] and Portuguese [26].

In the expand model, one of the main methodologies used to extend a wordnet coverage from the variants associated with the Princeton WordNet synsets is the acquisition of their translations from parallel corpora. In fact, we have applied that methodology in a previous

---

[7] `http://opus.lingfil.uu.se/`
[8] `http://per-fide.di.uminho.pt/`

▪ **Table 2** Precision of parallel corpus-based expansion in [8] and number of obtained variants.

	Precision	New variants
SemCor	78.13%	2,053
Unesco	80.84%	2,150
Lega	77.42%	1,172
Eroski	80.28%	1,777
Tectra	82.74%	948

phase of the Galnet development [8], using the WN-Toolkit [16] – a set of Python programs for the creation or enlargement of wordnets – to expand the Galnet first distribution (released for download in 2012 as part of the MCR 3.0) from two different available parallel textual resources: the automatically translated (with Google Translate) English–Galician SemCor Corpus[9]; and the four sections of the CLUVI Corpus, namely, the Unesco Corpus of Spanish–Galician scientific-technical texts, the Lega Corpus of Galician–Spanish legal texts, the Eroski Corpus of Spanish–Galician consumer information texts and the Tectra Corpus of English–Galician literary texts. In all cases, only the English or Spanish part of the parallel corpora has been sense-tagged for the experiment, using FreeLing [17] for parsing with the UKB word sense disambiguator [1].

After the sense annotation, a word alignment algorithm is used in order to identify the candidate target variants in the texts. For that experiment, we use a very simple word alignment algorithm which is based on the most frequent translation and which is available as a part of the WN-Toolkit. That alignment algorithm calculates the most frequent translation found in the corpus for each synset taking into account that the parallel corpus must be tagged with WordNet synsets in the source part and the target corpus must be lemmatised and tagged with very simple tags (*n* for nouns, *v* for verbs, *a* for adjectives, *r* for adverbs, and any other letter for other parts of speech).

In Table 2 we can observe the precision and the number of new variants obtained in the experiment from each parallel corpus. The evaluation has been performed in an automatic way, comparing the obtained variants with the existing variants in the current distribution of Galnet. If the variant obtained for a given synset already exists in that same synset, the result is evaluated as correct. If there are no Galician variants for a given synset in the reference Galnet, this result is evaluated as incorrect. It should be noted that the automatically obtained precision values tend to be lower than real values. The reason is that sometimes we have one or more variants for a given synset in the reference Galnet, but the obtained variant is not present. If the obtained variant turns out to be correct, it will be evaluated as incorrect anyway.

In [16] the same methodology is applied to the automatic translation (via Google Translate) of the English SemCor to six languages (Catalan, Spanish, French, German, Italian and Portuguese). Table 3 shows the results of the automatic evaluation of the data yielded from that experiment for the expansion of the wordnets of these six languages.

Another line of research on automatic extension of ontologies is carried out for Portuguese in the framework of Onto.PT[10] [11], where new synsets are obtained from lexicographical, encyclopedic and textual resources using different similarity indexes for lexical acquisition [10].

---

[9] `http://www.gabormelli.com/RKB/SemCor_Corpus/`
[10] `http://ontopt.dei.uc.pt/`

■ **Table 3** Precision of parallel SemCor-based expansion in [16] and number of obtained variants.

	Precision	New variants
Catalan	87.63%	449
Spanish	88.93%	504
French	91.83%	142
German	70.26%	1,285
Italian	93.81%	66
Portuguese	84.14%	324

■ **Listing 1** Portuguese segment after being processed by FreeLing.

```
Javé javé NCMS000 1 −
concedeu conceder VMIS3S0 1 02327200−v:0.00125866/02199590−v:0.0012442
uma um DI0FS0 0.903495 −
grande grande AQ0CS00 0.998339 01382086−a:0.00262735/01472628−a:0.000829225
vitória vitória NCFS000 1 07473441−n:0.0155845
```

4   Methodology

The process of extraction of proposals for variants uses, as resources, the multilingual Bible, namely the alignments from Portuguese, English, Catalan and Spanish to Galician, as well as these languages' respective wordnets. Section 4.1 describes how the Bible was preprocessed, and Section 4.2 explains the variant proposals extraction algorithm.

4.1   Multilingual Bible Preprocessing

The multilingual Bible is available in a set of translation memory exchange (TMX) files, one for each Bible book. Thus, the first step is the concatenation of these files, in a single translation memory, and the projection of each language in a separate textual file, where each translation unit resides in its own line. Also, as a few translation units do not include translations for every other language, in these cases an empty line is created in the respective text file.

Each one of these files is then processed using FreeLing, where each word is annotated with its lemma and part of speech. For those words present in the wordnet for that language, FreeLing uses the UKB algorithm in order to associate the more probable sense (which corresponds to a WordNet synset) to that word. FreeLing does not add only one sense, but a set of different senses, adding a probability measure to each one. This process is done taking care of translation units boundaries, in order to keep them after the annotation process. Listing 1[11] presents an example of a Portuguese translation unit after being processed by FreeLing.

In Listing 1, each line corresponds to a word, with different fields (or columns) separated by spaces: original word, lemma, part-of-speech, tagger confidence and a list of references to the synsets containing that lemma. References are separated by a slash, and each reference comprises the synset ILI and the UKB algorithm confidence.

---

[11] The example was truncated to a maximum of two senses.

Listing 2 Portuguese segment based on lemmas.

```
javé conceder um grande vitória
```

$$
\textit{vitória}
\begin{cases}
\textit{vitoria} & : & 56.37~\% \\
\textit{salvación} & : & 7.78~\% \\
\textit{triunfo} & : & 4.66~\% \\
\textit{xustiza} & : & 2.56~\% \\
\textit{axuda} & : & 1.77~\%
\end{cases}
$$

Figure 1 Example of PTD entry for the word *vitória* for the language pair $\text{PT} \leftrightarrow \text{GL}$.

The lemma information from these files are then concatenated together, in order to reconstruct the original text files, but with each word replaced by each lemma (Listing 2).

These lemmatised files are then used by NATools [27] to compute probabilistic translation dictionaries (PTD) for the following language pairs: $\text{PT} \leftrightarrow \text{GL}$, $\text{EN} \leftrightarrow \text{GL}$, $\text{CA} \leftrightarrow \text{GL}$ and $\text{ES} \leftrightarrow \text{GL}$. A PTD is a dictionary that maps each word from the source language to a set of probable translations, together with their probability (see Figure 1 for an example entry). The main advantage of using this kind of dictionary is that the domain and range of the dictionary cover all the corpus words (even if, in some situations, proposing a bad translation).

4.2 Extraction of Variant Proposals

The extraction of variant proposals is done through the alignment of words in the Bible that were annotated by FreeLing with one or more WordNet senses. The algorithm is based on the following steps, which are executed for each translation unit of the Bible:

1. For each one of the source languages (PT, EN, CA and ES) the algorithm gets the current annotated translation unit, and finds semantically annotated words. For each one of these words, it saves in a dictionary the ILI for the three most probable senses. This step results in a dictionary that can be formally defined as the following map:

   $$
   \text{ILI} \mapsto \left(\mathcal{L} \mapsto \langle \text{W}_{\mathcal{L}} \cup \text{L}_{\mathcal{L}} \mapsto \mathbb{N} \rangle \right),
   $$

   where $\mathcal{L}$ stands for the set of source languages, and $\text{W}_{\mathcal{L}}$ and $\text{L}_{\mathcal{L}}$ for the set of words and lemmas, respectively, from language $\mathcal{L}$. These forms and lemmas are counted and their number of occurrences is saved.

   As an example, for the synset with ILI = 07473441-n, the top level dictionary would have the following entry, meaning that only one translation was found in each language and that this synset occurs two times in the whole Bible:

   $$
   \text{07473441-n} \rightarrow
   \begin{cases}
   \text{PT} & \rightarrow & \{\text{vitória} \rightarrow 2 \\
   \text{EN} & \rightarrow & \{\text{victory} \rightarrow 2 \\
   \text{ES} & \rightarrow & \{\text{victoria} \rightarrow 2 \\
   \text{CA} & \rightarrow & \{\text{victòria} \rightarrow 2
   \end{cases}
   $$

2. The dictionary created in the previous step is processed, one entry at a time (for each ILI): if that ILI has at least three source languages – that is, if the previous process found this ILI in, at least, three of the four processed languages – then it is considered. If not, it is discarded.

For each one of these forms and lemmas, the probabilistic translation dictionary is queried, and the probable translations retrieved. Those translations with a translation probability lower than 20% are discarded. For the other translations, another dictionary is created that maps each possible translation to the accumulated translation probability and an accumulated similarity measure (a simple Boolean measure that states if the source word and translation word are orthographically similar). This dictionary has the following structure:

$$\text{W}_{GL} \mapsto \langle \mathbb{R}, \mathbb{N} \rangle$$

where the first element of the pair is the accumulated translation probability, and the second is the accumulated similarity measure.

For the ILI presented before, this dictionary would contain:

$$\begin{cases} \text{triunfo} & \rightarrow & (0.2024, 0) \\ \text{vitoria} & \rightarrow & (2.4876, 3) \end{cases}$$

which means the Galician variant *vitoria* is similar to three variants from any other languages, and has a greater accumulated translation probability than the word *triunfo*. Finally, the target sentence, from the translation unit, is retrieved. Each word from the sentence whose part of speech matches the part of speech from the ILI (note that ILI includes a character, at the end of the code, indicating the PoS for those synset variants) and also matches one of the previously obtained translations will be considered a variant. If this variant is already part of the Galnet synset, then it is marked as a known variant (which will be used to validate the algorithm). All these variants are saved in a global dictionary, as variant candidates for that specific ILI. This dictionary also saves an accumulated similarity measure, and a counter on how many translation units suggested this specific candidate:

$$\text{ILI} \mapsto (\text{L}_{GL} \mapsto \langle \mathbb{N}, \mathbb{N} \rangle)$$

For the example ILI used above, the resulting dictionary includes:

$$07473441\text{-n} \rightarrow \begin{cases} \text{triunfo} & \rightarrow & (2, 0) \\ \text{vitoria} & \rightarrow & (14, 37) \end{cases}$$

meaning that the word *triunfo* was suggested performing the algorithm in two segments of the Bible, while the word *vitoria* was suggested by fourteen segments. Also, *triunfo* was never similar to any other language variant while *vitoria* was similar 37 times (see Section 4.2.1).

4.2.1   Similarity Measure

Regarding the similarity measure mentioned before, its main goal is to give preference to those variants that are orthographically similar to variants from other languages. This is specially useful given that three from the four source languages (PT, CA and ES) have some resemblances. For the pair PT $\leftrightarrow$ EN, the dictionary computed on some previous experiments for the Portuguese wordnet [28], based on rewriting rules, is used. For the other languages, the well known Levenshtein distance [13] is applied, with a distance of 1.

5   Evaluation and results

For the evaluation of this methodology we designed a protocol oriented to the manual analysis of the results by an expert lexicographer. The review of the candidates is done by checking their lexicographical adequacy to the concept represented in the WordNet knowledge base. In case of lack of adequacy, the proposal receives a code that indicates the reason for its exclusion. The codes established in the revision protocol are the following:

1. **MCR source:**
   wrong variant introduced by mistakes in the wordnets for other languages;
2. **Lemmatisation:**
   wrong variant introduced by a mistake in the lemmatisation process;
3. **Galician normative:**
   variant whose form is deprecated by the official Galician normative;
4. **False friends**:
   variant form is equal to a variant in other language, but with a different meaning;
5. **Levenshtein distance:**
   similar to the false friend class, but originated by the similarity measure.

5.1   Error analysis

In the following sections, we will describe and exemplify the phenomena grouped by each error code, and their influence on the overall evaluation of the results.

5.1.1   MCR source

The existing lexicographic errors found in the MCR – especially in Portuguese, Catalan or Spanish – can produce erroneous candidates for extraction. These candidates are not considered errors of the extraction methodology and, therefore, are not taken into account for the evaluation of its results.

For example, the Portuguese variant "fazer" included in the concept "give birth" (ili-30-00056930-v)[12] is too generic for this sense and gives rise to the proposal "facer", also too generic in Galician and therefore classified as incorrect.

The identification of lexicographic errors in Portuguese through this protocol will also be used for the revision and maintenance of the PULO knowledge base.

5.1.2   Lemmatisation

In some cases, the wrong candidates come from the incorrect lemmatisation performed by FreeLing. These cases can not be considered as errors of the extraction methodology, so they will not be included in the evaluation of its performance.

By way of illustration, some of these cases are the wrong candidates "bendiciu" (past tense inflected form of the verb) instead of "bendicir" (infinitive of the verb by which it must be lemmatised); "costa" instead of "costas" for the concept in ili-30-05588174-n,[13] where the lemma should go in plural because the word can only be used in the plural form in this

---

[12] http://sli.uvigo.gal/galnet/galnet_var.php?version=dev&ili=ili-30-00056930-v
[13] http://sli.uvigo.gal/galnet/galnet_var.php?version=dev&ili=ili-30-05588174-n

sense; or "testemuño" instead of "testemuña" for the concept in ili-30-10786517-n,[14] where the lemma should be feminine because the word can only be used in feminine in this sense.

In all these cases, the extraction algorithm generates a wrong candidate for Galnet because of the bad selection of the lemma during the automatic linguistic analysis carried out by FreeLing.

5.1.3   Galician normative

In a few cases, the Galician candidate generated from the CLUVI multilingual Bible represents a variant rejected by the current official normative of the Galician language [23]. For example, the proposed candidate "lá", an ancient spelling of the word for the concept of "outer coat of especially sheep and yaks" (ili-30-01899593-n),[15] is not well written following the current regulations, which prescribe it without graphic accent (i.e., "la").[16]

These erroneous candidates cannot be considered as the result of any dysfunction of the extraction methodology, so they cannot be taken into account for the evaluation of the accuracy of their results.

5.1.4   False friends

In some cases, an erroneous candidate is produced because of the orthographic identity – but not identity of meaning – between the proposed Galician form and its source. These false friends occur more frequently between Galician and Portuguese, although they can also occur between Galician and Catalan or Spanish. For example, the erroneous proposal "apagar" for the meaning "put an end to; kill" (ili-30-00478217-v)[17] is generated from the formal identity of the Galician word with the Portuguese form "apagar" which, unlike Galician, does have that meaning.

The cause of this erroneous behaviour lies in certain errors of the semantic tagging of the UKB algorithm and it is limited to polysemous words that are false friends in some of their meanings, but are true friends in others. For example, the word "apagar" in Galician is polysemous. In the example above (ili-30-00478217-v), it is a false friend of the Portuguese verb "apagar". However, in another of its senses, "put out, as of fires, flames, or lights" (ili-30-02761897-v), the Portuguese verb "apagar" and the Galician verb "apagar" are true friends, because they share the same meaning. In the parallel corpus, a number of cases of "apagar" are classified with the shared sense of ili-30-00478217-v in equivalent phrases from Galician and Portuguese. But it also happens that the Galician word "apagar" with the ili-30-02761897-v sense is incorrectly classified by FreeLing as ili-30-00478217-v.

Because of this erroneous semantic tagging, attributable to a malfunction of the UKB algorithm, the extraction algorithm mistakenly proposes the candidate "apagar" as a candidate Galician translation for the sense of ili-30-00478217-v. Hence, these wrong candidates can not be considered as extraction errors, as they are due to the UKB algorithms, and as such, they will not be taken into account in the evaluation of the results of the proposed algorithm.

---

[14] `http://sli.uvigo.gal/galnet/galnet_var.php?version=dev&ili=ili-30-10786517-n`

[15] `http://sli.uvigo.gal/galnet/galnet_var.php?version=dev&ili=ili-30-01899593-n`

[16] `https://academia.gal/dicionario/-/termo/busca/la`

[17] `http://sli.uvigo.gal/galnet/galnet_var.php?version=dev&ili=ili-30-00478217-v`

Table 4 Results of the human evaluation.

Candidatures	1665
No error	1443
MCR source	144
Lemmatisation	8
Galician normative	4
False friends	18
Levenshtein distance	48
Precision	96.8%

5.1.5 Levenshtein distance

Most of the inaccuracies of the methodology presented here are found in candidates formally similar to the source variants, but with different meaning. For example, the system generates the Galician proposal "ver" for the concept "come to pass; arrive, as in due course" (ili-30-00341917-v)[18] due to its formal similarity with the Portuguese variant "vir" for this sense, with which it maintains a Levenshtein distance 1, accepted by the system in words between 1 and 5 letters. However, "ver" ("to see", in Galician) does not have that sense at all, so it is an unexpected error generated by the methodology. The correction of these errors in a programmatic way is not easy, since the elimination of Levenshtein distance would suppose an unwanted decrease in the coverage of the results.

5.2 Evaluation results

After running the extraction algorithm we obtained 4,353 new Galician variants, from which 1,665 were orthographically similar to variants from other languages according to the similarity measure referred to in section 4.2. As previously stated, the evaluation of the results has been carried out completely manually by an expert lexicographer by checking the lexicographical adequacy of the candidates to the concept represented in the WordNet knowledge base, and in case of lack of adequacy, indicating with the corresponding error code the reason for its exclusion. In Table 4 we can observe the results of this evaluation.

As said before, precision is calculated without taking into account the existing lexicographic errors found in the MCR, the errors from the lemmatisation process performed by FreeLing, the errors from changes in the official normative of Galician or the errors in semantic tagging produced by the UKB algorithm, which are not considered inherent errors of the extraction methodology.

6 Final remarks

The results of human evaluation in section 5 show that the presented methodology outperforms the results reported in previous works [8, 16] and discussed above in Section 3 (Tables 2 and 3). This would demonstrate the importance of associating lexical semantic information to lexical alignment for the identification of variant candidates in parallel corpora.

---

[18] `http://sli.uvigo.gal/galnet/galnet_var.php?version=dev&ili=ili-30-00341917-v`

Future work includes both introducing the validated set of Galician extracted variants in the Galnet knowledge base, as generating the Galnet 3.0.27 new distribution.[19] We will also revise the erroneous Portuguese variants in PULO identified during the evaluation. Finally, we will apply this methodology for the expansion of data in the PULO wordnet.

────── **References** ──────

1 Eneko Agirre and Aitor Soroa. Personalizing PageRank for Word Sense Disambiguation. In *Proceedings of the 12th Conference of the European Chapter of the ACL*, pages 33–41, 2009.

2 Purya Aliabadi, Mohamed Sina Ahmadi, Shahin Salavati, and Kyumars Sheykh Esmaili. Towards building KurdNet, the Kurdish WordNet. In *Proceedings of the 7th Global Word-NetConference*, Tartu, Estonia, 2014.

3 José João Almeida, Sílvia Araújo, Nuno Carvalho, Idalete Dias, Ana Oliveira, André Santos, and Alberto Simões. The Per-Fide Corpus: A new resource for corpus-based terminology, contrastive linguistics and translation studies. In Tony Berber Sardinha and Telma de Lurdes São Bento Ferreira, editors, *Working with Portuguese Corpora*, pages 177–200, London, 2014. Bloomsbury Publishing.

4 Jordi Atserias, Salvador Climent, Xavier Farreres, German Rigau, and Horacio Rodriguez. Combining multiple methods for the automatic construction of multilingual Word-Nets. In *Recent Advances in Natural Language Processing II. Selected papers from RANLP*, volume 97, pages 327–338, 1997.

5 Laura Benítez, Sergi Cervell, Gerard Escudero, Mònica López, German Rigau, and Mariona Taulé. Methods and tools for building the Catalan WordNet. In *In Proceedings of the ELRA Workshop on Language Resources for European Minority Languages*, 1998.

6 Christos Christodouloupoulos and Mark Steedman. A massively parallel corpus: the Bible in 100 languages. *Language Resources and Evaluation*, 49(2):375–395, 2015.

7 Christiane Fellbaum, editor. *WordNet: An electronic lexical database.* MIT Press, Cambridge, 1998.

8 Xavier Gómez Guinovart and Antoni Oliver. Methodology and evaluation of the Galician WordNet expansion with the WN-Toolkit. *Procesamiento del Lenguaje Natural*, 53:43–50, 2014.

9 Xavier Gómez Guinovart and Miguel Anxo Solla Portela. Building the Galician wordnet: methods and applications. *Language Resources and Evaluation*, 52, 2017. `doi:10.1007/s10579-017-9408-5`.

10 Hugo Gonçalo Oliveira. *Onto.PT: Towards the Automatic Construction of a Lexical Ontology for Portuguese.* Tese de doutoramento, Universidade de Coimbra, 2013. URL: `http://eden.dei.uc.pt/~hroliv/pubs/GoncaloOliveira_PhdThesis2012.pdf`.

11 Hugo Gonçalo Oliveira and Paulo Gomes. ECO and Onto.PT: a flexible approach for creating a Portuguese wordnet automatically. *Language Resources and Evaluation*, 48(2):373–393, 2014. `doi:10.1007/s10579-013-9249-9`.

12 Aitor Gonzalez-Agirre, Egoitz Laparra, and German Rigau. Multilingual Central Repository version 3.0. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, 2012. ELRA.

13 Vladimir Levenshtein. Binary Codes Capable of Correcting Deletions and Insertions and Reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

─────────

[19] `http://sli.uvigo.gal/download/SLI_Galnet/`

14 Márton Miháltz, Csaba Hatvani, Judit Kuti, György Szarvas, János Csirik, Gábor Prószéky, and Tamás Váradi. Methods and results of the Hungarian wordnet project. In *Proceedings of the Fourth Global WordNet Conference. GWC*, pages 387–405, Szeged, Hungary, 2008.

15 George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. WordNet: An on-line lexical database. *International Journal of Lexicography*, 3:235–244, 1990.

16 Antoni Oliver. WN-Toolkit: Automatic generation of wordnets following the expand model. In *Proceedings of the 7th Global WordNet Conference*, Tartu, 2014. GWN.

17 Lluís Padró and Evgeny Stanilovsky. Freeling 3.0: Towards wider multilinguality. In *Proceedings of the Language Resources and Evaluation Conference (LREC 2012)*, Istanbul, Turkey, May 2012. ELRA.

18 Emanuele Pianta, Luisa Bentivogli, and Christian Girardi. MultiWordNet. developing an aligned multilingual database. In *1st International WordNet Conference*, pages 293–302, Mysore, India, 2002.

19 Elisabete Pociello, Eneko Agirre, and Izaskun Aldezaba. Methodology and construction of the Basque WordNet. *Language Resources and Evaluation*, 45(2):121–142, 2011. `doi: 10.1007/s10579-010-9131-y`.

20 Quentin Pradet, Gaël de Chalendar, and Jaume Baguenier Desormeaux. WoNeF, an improved, expanded and evaluated automatic French translation of WordNet. In *Proceedings of the 7th Global WordNetConference*, Tartu, Estonia, 2014.

21 Desmond Darma Putra, Abdul Arfan, and Ruli Manurung. Building an Indonesian WordNet. In *Proceedings of the 2nd International MALINDO Workshop*, 2008.

22 Ida Raffaeli, Bekavac Božo, Željko Agić, and Marko Tadić. Building Croatian WordNet. In *Proceedings of the 4th Global WordNet Conference*, Szeged, Hungary, 2014.

23 Real Academia Galega. *Normas ortográficas e morfolóxicas do idioma galego*. Editorial Galaxia, Vigo, 2004.

24 Philip Resnik, Mari Broman Olsen, and Mona Diab. The Bible as a Parallel Corpus: Annotating the 'Book of 2000 Tongues'. *Computers and the Humanities*, 33(1-2):129–153, 1999.

25 Benoît Sagot and Darja Fišer. Building a free French wordnet from multilingual resources. In *Proceedings of OntoLex*, 2008.

26 Alberto Simões and Xavier Gómez Guinovart. Bootstrapping a Portuguese WordNet from Galician, Spanish and English wordnets. In *Advances in Speech and Language Technologies for Iberian Languages*, volume 8854 of *Lecture Notes in Computer Science*, pages 239–248, Berlin, 2014. Springer.

27 Alberto Simões and José João Almeida. NATools – a statistical word aligner workbench. *Procesamiento del Lenguaje Natural*, 31:217–224, September 2003.

28 Alberto Simões and Xavier Gómez Guinovart. Dictionary Alignment by Rewrite-based Entry Translation. In José Paulo Leal, Ricardo Rocha, and Alberto Simões, editors, *2nd Symposium on Languages, Applications and Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 237–247, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/OASIcs.SLATE.2013.237`.

29 Jörg Tiedemann. Parallel data, tools and interfaces in OPUS. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, pages 2214—-2218, Istanbul, 2012. ELRA.

30 Piek Vossen, editor. *EuroWordNet: A multilingual database with lexical semantic networks*. Kluwer Academic Publishers, Norwell, 1998.

Path Patterns Visualization in Semantic Graphs

José Paulo Leal

CRACS & INESC-Porto LA
Faculty of Sciences, University of Porto, Portugal
zp@dcc.fc.up.pt
 https://orcid.org/0000-0002-8409-0300

Abstract

Graphs with a large number of nodes and edges are difficult to visualize. Semantic graphs add to the challenge since their nodes and edges have types and this information must be mirrored in the visualization. A common approach to cope with this difficulty is to omit certain nodes and edges, displaying sub-graphs of smaller size. However, other transformations can be used to abstract semantic graphs and this research explores a particular one, both to reduce the graph's size and to focus on its path patterns.

Antigraphs are a novel kind of graph designed to highlight path patterns using this kind of abstraction. They are composed of antinodes connected by antiedges, and these reflect respectively edges and nodes of the semantic graph. The prefix "anti" refers to this inversion of the nature of the main graph constituents.

Antigraphs trade the visualization of nodes and edges by the visualization of graph path patterns involving typed edges. Thus, they are targeted to users that require a deep understanding of the semantic graph it represents, in particular of its path patterns, rather than to users wanting to browse the semantic graph's content. Antigraphs help programmers querying the semantic graph or designers of semantic measures interested in using it as a semantic proxy. Hence, antigraphs are not expected to compete with other forms of semantic graph visualization but rather to be used a complementary tool.

This paper provides a precise definition both of antigraphs and of the mapping of semantic graphs into antigraphs. Their visualization is obtained with antigraphs diagrams. A web application to visualize and interact with these diagrams was implemented to validate the proposed approach. Diagrams of well-known semantic graphs are also presented and discussed.

Figure 1 An antigraph diagram of WordNet 2.1.

1 Introduction

Graphs, like most mathematical entities, are inherently visual. In fact, our mathematical intuition relies heavily on our ability to visualize angles, functions, vectors or geometric figures. It fails us, for instance, when we try to visualize a hypercube. However, the projection of multidimensional solids in 3 or 2-dimensional spaces give us an idea of these entities' shape. The visualization of the hypercube is an apt metaphor of the key insight that drives this research: the understanding of a complex entity may be improved by looking at the shadow it casts.

Graphs have a particular role in visualization since they are the data model of most diagrams. Diagrams enrich graphs in two ways: a graphical syntax for nodes and edges; and the layout of nodes and edges on a surface. Different kinds of diagrams have been developed for many purposes. These diagrammatic languages are used for modeling and visualizing relationships among entities, even when they are purely abstract.

In spite of their ability to show relationships and symmetries, large diagrams are difficult to visualize. Too many nodes and too many entangled edges reduce our perception on the underlying graph. This is particularly the case of the semantic web, where graphs are growing increasingly larger and denser. Section 2 presents different attempts to provide visualizations of large semantic graphs, with efficient approaches to process large quantities of data and methods to abstract them, mostly by omitting nodes and edges with certain features. These diagrams represent the graphs themselves, and the layout may highlight general properties, such as symmetry, but usually they do not reveal specific features such as patterns formed by nodes and edges.

The novelty of the antigraph approach is the abstraction of large semantic graphs into a much smaller graph highlighting its *path* patterns. The abstraction process consist on mapping semantic graphs into antigraphs, a particular kind of graph with an associated diagram type. Sets of edges with the same type are mapped into nodes and sets of nodes into edges. This process reverses the nature of the constituents of a graph, thus the metaphor of antimatter where positrons, rather than electrons, revolve around a nucleus made of antiprotons and antineutrons, rather than the protons and neutrons of regular matter. Section 3 defines antigraphs and their relationship with semantic graphs. It also introduces the antigraph diagrams used for their visualization and provides small examples of this kind of diagram. The final subsection compares antigraphs with other forms of representing the properties of semantic graphs, such as ontologies.

An implementation of the mapping and diagram layout is described in Section 4. It is deployed as a web application for browsing antigraphs and exporting them as vector images, such as the diagram representing Wordnet 2.1, shown in Figure 1. It is available online[1] and is useful to validate the proposed approach. Section 5 presents examples of diagrams produced with this tool to illustrate different techniques to create meaningful visualizations of large semantic graphs and discover relevant path patterns. The last section summarizes the research presented in this paper, highlights its main contributions and identifies opportunities for future research.

2 Related Work

Knowledge bases such as WordNet[2] [8], Yago[3] [13] and DBpedia[4] [4], have a massive amount of information. A typical representation of these knowledge bases are node-link multigraphs, where each node has a type and nodes are connected by links representing the relationship between them.

A convenient way to analyze this data is using data visualization. The most common type of visualization is focused on the analyzes of resources, in particular, those with a high outdegree. The main challenge of semantic graph visualization and management is related to the graph size. This type of graphs has several thousands of nodes and edges and are usually very dense.

The literature presents several approaches to handle the visualization and management of node-link graphs. Most of the related work on massive graphs visualization is handled through hierarchical visualization. This type of approach has low memory requirements, however, it depends on the characteristics of the graph. The graph hierarchy can be extracted using different kinds of methods. Tools such as ASK-GraphView [1], Tulip [3] and Gephi [5] explore clustering and partitioning methods, creating an abstraction of the original graph that can be easily visualized. Another technique used to build hierarchies is based on the combination of edge accumulation with density-based node aggregation [17]. Visual complexity can also be reduced by hub-based hierarchies, where the graph is fragmented into smaller components, containing many nodes and edges, making meta nodes, as described in [15]. GrouseFlocks [2] allows users to manually define their own hierarchies.

There are specific tools when the semantic graph is in Resource Description Framework (RDF) format, however, they require loading the full graph. Some desktop-based tools, such as Protégé [5] and RDF Gravity[6], are mainly used with purpose of aiding developers to construct their ontologies, providing also complex graph visualizations. Of all available tools for linked data visualization the most notable ones are the following. Fenfire [11] is a generic RDF browser and editor that provides a conventional graph representation of the RDF model. The visualization is scalable by focusing on one central node and its surroundings. RelFinder[7] [12] is a tool that extracts from a Linked Open Data (LOD) source the graph of the relationships between two subjects. It provides an interactive visualization by supporting systematic analysis of the relationships, such as highlighting, previewing and filtering features.

---

[1] `http://quilter.dcc.fc.up.pt/antigraph`
[2] `https://wordnet.princeton.edu/`
[3] `https://www.mpi-inf.mpg.de/yago-naga/yago`
[4] `http://wiki.dbpedia.org/`
[5] `http://protege.stanford.edu/`
[6] `http://semweb.salzburgresearch.at/apps/rdf-gravity/`
[7] `http://www.visualdataweb.org/relfinder.php`

ZoomRDF [16] is a framework for RDF data visualization and navigation that uses three special features to support large scale graphs. It uses *space-optimized* visualization algorithms that display data as a node-link diagram using all visual space available. *Fish-eye zooming* is another feature that allows the exploration of selected elements details, while providing the global context. The last feature is the *Semantic Degree of Interest* assigned to all resources that consider both the relevance of data and user interactions. LODeX [6] produces a high-level summarization of a LOD source and its inferred schema using SPARQL endpoints. The representative summary is both visual and navigable. The platform graphVizdb[8] [7] is a tool for efficient visualization and graph exploration. It is based on a *spatial-oriented* approach that uses a disk-based implementation to support interactions with the graph.

3   Antigraph definition

The most distinctive feature of antigraphs is that nodes and edges are reversed relatively to the semantic graphs that generated them. Subsection 3.1 explains the motivation behind this decision and characterizes the main components of antigraphs, namely antinodes and antiedges, as well as their features.

The proposed approach to the visualization of semantic graphs can be divided into two parts. Firstly, the semantic graph is abstracted to another graph – the *antigraph* – that promotes types of edges. Secondly, this abstracted graph is visualized using a special kind of diagram —the *antigraph diagram*— that emphasises path patterns. The following two subsections detail each facet of the antigraph approach.

Finally, Subsection 3.3 discusses antigraphs as abstractions of semantics graphs, in relations to ontologies.

3.1   Motivation

Nodes have the main role in a graph. Edges connect nodes and establish relationships among them. The goal of antigraphs is to abstract a given graph, highlighting edges and reducing its size. Hence, in an antigraph nodes and edges are reversed, i.e. an *antinode* abstracts edges and an *antiedge* abstracts nodes. A graph and its *antigraph* have the same duality of matter and antimatter (where electrons are replaced by positrons and, protons by antiprotons and neutrons by antineutrons).

It is important to note that an antinode abstracts an edge *type* rather than a single edge. Hence the order (the number of nodes) of an antigraph is in general much smaller than that of the graph it abstracts. For instance, the graph of WorwdNet 2.1 has about 2 million edges with 27 edge types, hence 27 is the order of the of reductions that abstract it.

An antiedge expresses a relationship between a pair of antinodes, namely that the edge types it represents can be connected to form a length 2 path. Two edges form a length 2 path when the target of the first is the source of the second. Since an antiedge represents a set of nodes, the size (the number of edges) of an antigraph is much smaller than the size of the graph it abstracts. Considering that antiedges can be laces, the number antiedges is less or equal to n^2, where n is the number of antinodes. For instance, the size of the WordNet 2.1 graph is about half a million but the size of its antigraph is only 214, well bellow the maximum of $27^2 = 729$.

---

[8] `http://graphvizdb.imis.athena-innovation.gr/`

The expressiveness of antinodes and antiedges is increased by adding weights to them. The weight of an antinode is the percentage of edges with the type it abstracts. For instance, if a graph has half of its edges of type t then the antinode reflecting t has weight $1/2$. Hence, antinodes with higher weight reflect edge types that are more frequent in the graph. Obviously, the sum of antinode weights must be 1.

By the same token, the weight of an antiedge is the percentage of nodes that participate in length 2 paths involving edge types they have as source and target, respectively. For instance, if an antinode reflects the edge type t_1 and another the edge type t_2, and $1/3$ of the nodes are target of t_1 and source of t_2, then the weight of the antiedge $t_1 \rightarrow t_2$ is $1/3$.

One would expect every node to be reflected by an antiedge, but for that to happen the nodes that are just sources (not the target of any edge) or just targets (not the source of any edge) must also be abstracted by antiedges. To ensure that all nodes are reflected by antiedges it is necessary to introduce two special antinodes: *bottom*, denoted as $\perp$; and *top*, denoted by $\top$. The bottom antinode represents a nonexisting edge type that would come before the start of a path. Conversely, the top antinode represents a nonexisting edge type that would come after the end of a path. Both special antinodes have weight 0, thus maintaining the invariant that the sum of all weights is 1.

The two special antinodes – bottom and top – allow the definition of antiedges that abstract nodes that are only source or target of edges. These antinodes are considered *special* to differentiate them from *regular* antinodes, that have an associated edge type. The antiedge $\perp \rightarrow t$ abstracts the nodes with a null indegree that are sources of edges with type t, and the antiedge $t \rightarrow \top$ abstracts the nodes with a null outdegree that are targeted by edges of type t.

In fact, the in(out)degrees of nodes must be taken into consideration in the weight of all antiedges. Consider a node n with indegree 2 and outdegree 3. For instance, if the two incoming edges and the three outgoing are of different types then the contribution of that node to each antiedge is $1/6$. Thus, the weight of an antiedge is the percentage of connecting nodes in paths formed by the edge types, pondered by their in(out)degrees. With this definition, the sum of antiedges weights is also 1.

As explained above, the introduction of the special antinodes bottom and top is essential to abstract all the nodes of the original graph in antiedges connecting them. One may wonder what other antinodes types should be considered. It should be noted that antinodes may have antiedge laces if the graph contains homogeneous paths, i.e. paths formed by a single type of edge. Since the goal of antigraphs is to highlight path patterns, it is important to distinguish different cases that would be amalgamated by generic antinodes with laces.

Certainly, not all antinodes have laces. These are considered *shallow* antinodes since they have at most paths of length 1. In contrast a *deep* antinode has homogeneous paths of higher length through its lace. Special cases of deep antinodes can be also considered: *cyclical*, where the laces contain homogeneous cycles, i.e. cycles using only the type of edge represented by the antinode; and *hierarchical*, where the laces represent confluent paths, i.e. where the nodes in homogeneous paths have branching factor above 2. These types provide information on the kind of paths formed "within" an antinode, similar to the information that can be extracted from other antiedges relating different antinodes.

In summary, an antigraph is an abstraction of a semantic graph. This does *not* mean that an antigraph is a sort of schema. A semantic graph does not comply with its antigraph, its the other way round: antigraphs have a functional dependency to semantic graphs. Thus, the information provided by an antigraph is of a different nature of that of an RDF or OWL ontology. This point is analyzed in greater detail in Subsection 3.3, after formalizing the definitions of antigraph and antigraph diagram.
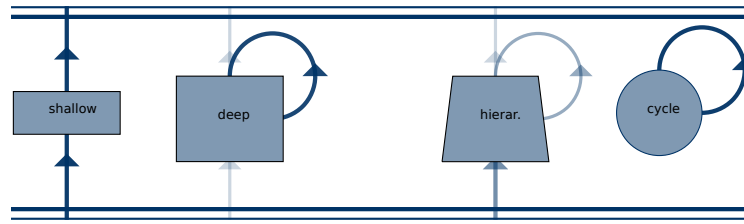
█ **Figure 2** Catalog of antinode types.

3.2   Diagram language

As explained in the previous subsection, an antigraph is an abstraction of a semantic graph. The antigraph diagram language is a visual representation of an antigraph intended to highlight the path patterns of the abstracted semantic graph. An antigraph has antinodes of different types connected by antiedges.

The type of an antinode is conveyed by its shape. A shallow antinode is represented by a horizontal rectangle, while a deep antinode is represented by a vertical rectangle. A cyclical antinode is represented by a circle or an ellipse, and a hierarchical antinode is represented by an isosceles trapezoid. The position of these shapes is their geometric center. The antigraph depicted in Figure 2 is a sort of catalog of antinode types, where the label of each regular antinode is the type's name.

The bottom and top antinodes are represented by a pair of parallel lines rather than shapes. As can be seen also in Figure 2, the parallel lines that represent each of these antinodes have different widths. The bottom antinode has a larger upper line and the top antinode is the reverse. The bottom and top antinodes are located respectively at the bottom and top of the diagram, as their names suggest. This way the paths created by antiedges tend to be directed upwards.

Unlike antinodes, antiedges have a single type. Hence, they are represented all by solid lines with an arrowhead positioned in their middle pointing to the target. Lines connecting from the bottom antinode, or to top antinode, are straight. All the others are curved so that antiedges with opposite directions do not overlap.

Antinodes and antiedges have weights in the $[0, 1]$ interval. Actually, both regular antinodes and antiedges have always nonnull weights; special antinodes (top and bottom) have null weights by definition. The nonnull weights of regular antinodes and antiedges are conveyed graphically too. The weight of an antinode is shown as a transparency, making dimmer the antiedges representing a smaller number of edges in the abstracted graph. The same principle applies to weights of antiedges. In this case, the weight is also shown as line width, making thicker the antiedges that represent a larger number of nodes. The semantic graph that originated the antigraph in Figure 2 has all edge types with the same number of edges, hence all antinodes have the same weight, thus they all have the same shade. A different thing happens with antiedges; each has a different shade, reflecting their different weights.

The regular antinodes in the catalog diagram are not connected to each other, just to bottom and top (with the exception of the cycle). This means that they do not form "joins". Using a syntax borrowed from SPARQL, it can be said that the semantic graph that generated it lacks triple patterns of the form

```
?a ?p ?b .
?b ?q ?c .
```
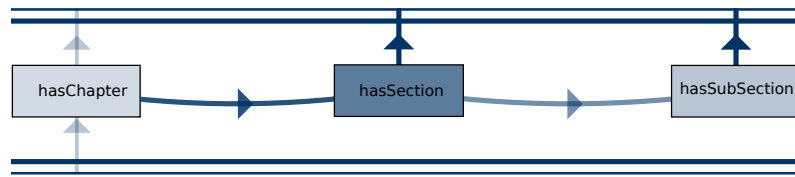
Figure 3 Book structure.
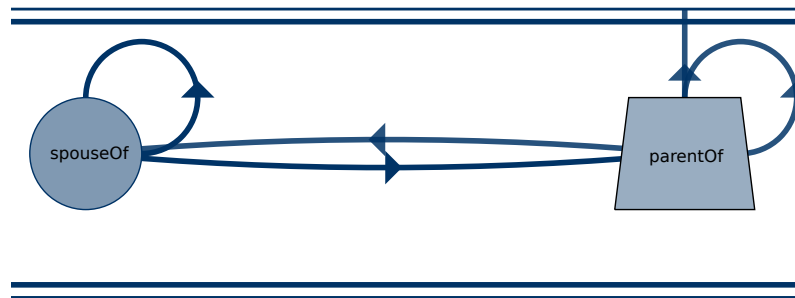
Figure 4 Family relationships.

The example in Figure 3 represents the structure of books, where a book has chapters and these have sections. The antigraph of such semantic graph has the properties *hasChapter*, *hasSection* and *hasSubSection*.

In this case "joins" are created using multiple edge types hence the antinodes have antiedges connecting them. In particular *hasChapter* is connected to *hasSection* and this to *hasSubSection*. The reader should note that the three regular antinodes are connected to the top, meaning that there are chapters, sections and subsections that are not subdivided, and that only *hasChapter* is connected from bottom, meaning that only these are connected from root elements of the hierarchy.

The previous example reflects a hierarchical structure, although with a different type of edge for each layer. The example in Figure 4 reflects a semantic graph with a couple of family relationships, namely *spouseOf* and *parentOf*. Their associated antinodes both have laces, which means that paths with a single type of edges can be created. The *parentOf* antinode has hierarchic as type, meaning that paths of length greater than 3 can be created and has an average branching factor above 2.

The simple patterns identified in the small examples above occur also in larger semantic graphs. Section 4 presents an antigraph browser that allows us to discover combinations of these patterns in in larger examples, as those analyzed in Section 5.

3.3 Antigraphs and ontologies

Semantic graphs are frequently encoded as sets of triples in the Resource Description Framework (RDF). This framework supports multiple vocabularies, including a vocabulary to describe other vocabularies – RDF Schema (RDFS) – which in turn lays the foundations for a richer ontological language – OWL. RDFS and OWL describe vocabularies in terms of classes and properties, where classes provide types for nodes and properties types for edges of semantic graphs, and define hierarchical relationships among those types.

The definition of semantic graph on which the definition of antigraphs relies is also based on types. However, these types are of a different nature. These node and edge types are not RDFS or OWL classes and properties, and they are not hierarchically related among

themselves. The node and edge types in the definition of antigraphs are the actual URIs used to label them.

Nevertheless, ontologies and antigraphs are somehow related in the sense that they both abstract semantic graphs. Thus, it is relevant to question if these two concepts – ontologies and antigraphs – overlap or compete in any way.

The concept of ontology varies for different communities [9]. In the semantic web, an ontology is usually understood as a formal definition of a domain of discourse. It declares a taxonomy of concepts and relationships among them. For instance, an ontology may declare *cat* and *dog* as classes, both as subclasses of *pet*, and the property *hasName* associating pets to their names (strings). RDFS and OWL ontologies are themselves RDF graphs, although not all RDF graphs are ontologies. In fact, most RDF graphs assert facts on resources using types and properties, such as "Rex is a dog"[9], but they do not define hierarchies of classes (concepts) and properties (relationships).

By using inference with an ontology it is possible to entail new facts from existing ones, such as "Rex is a pet". The reverse, to induce an ontology from a collection of facts, is much more complex. It is possible to process statements such as "Rex is a dog" and "Fifi is a cat", "Rex is a pet" and "Fifi is a pet" and induce an ontology similar to the example in the previous paragraph. However, ontologies are not usually created this way.

Ontologies prescribe how certain semantic graphs must be. They are not a sort of a "summarization" of existing semantic graphs. If an ontology is applicable to a particular semantic graph then the facts of the later should be consistent with the former; and as more facts are added, that consistency should be preserved without changing the ontology.

An antigraph is, in fact, a summarization of a semantic graph. It maps edges into antinodes and nodes into antiedges in a way that the antigraph paths condense several paths of the semantic graph it abstracts. However, only paths that actually exist in the semantic graph are abstracted into antigraph paths, not all the paths that would be consistent with the ontology. Moreover, since antinodes and antiedges have weights, the path frequency is also presented by the antigraph, which has no parallel in ontologies. As a semantic graph evolves and new nodes and edges are added (or removed), its antigraph may change to reflect it. In some cases, only the weights will be affected, if no kinds of path are created. In other cases, new antinodes result from edge types that did not exist before.

In summary, antigraphs and ontologies are different kinds of abstractions. Antigraphs abstract paths, highlighting the most frequent ones. Ontologies abstract relationships among concepts. The two abstractions are non-overlapping and are in fact complementary.

4   Antigraph browser

This section describes the design and implementation of a web application developed to validate the concept of antigraph. This web application – the antigraph browser – produces interactive antigraph diagrams from several data sources and is freely available online[10].

The antigraph browser is a Java web application developed with the Google Web Toolkit (GWT). It is composed of a client front-end running on a web browser and a server back end. The server is responsible for transforming a semantic graph in RDF format into an antigraph that is sent to the client. The front end is responsible for laying out diagrams and managing user interaction, as explained in the following subsections.

---

[9] "Rex is a dog" are two RDF facts. Assuming `ex` as an alias for a namespace, that sentence would be represented by the RDF facts «`ex:rex ex:hasName "Rex"`» and «`ex:rex rdf:type ex:dog`».

[10] `http://quilter.dcc.fc.up.pt/antigraph`

4.1 Back end processing

The mapping of semantic graphs into antigraphs is performed in two stages by the back end. Firstly, a set of graph reductions is produced from the semantic graph triples. Secondly, the antigraph data is computed by processing these graph reductions.

A graph reduction instance aggregates edges of a single type, that is, the semantic graph obtained by considering only the edges of that type. It records the nodes that are sources and those that are targets, and computes their in and outdegrees. The links between theses nodes are also recorded to compute aggregate measures on the reduction such as the number of cycles, depth and branching factor.

Graph reductions are computed by processing a stream of RDF triples. For each subject-predicate-object triple the reduction corresponding to its predicate the subject is selected and recorded as source and the object as target.

Each reduction corresponds to an antinode. Thus the second stage creates an antinode for each reduction found in the first stage, assigning it a weight computed as the percentage of edges in the graph. The top and bottom antinodes, with null weight, are also created. Then it iterates over the pairs of reductions to create antiedges.

The computation of antiedges' weights is more complex than that of antinodes, as it involves determining the intersection of the targets and source sets of nodes respectively of the source and target antinodes of each antiedge. Also, the contribution of each of these nodes depends both on their in(out)degrees on the reduction. The pairs of antinodes with nonnull weights create antiedges.

Antiedges connecting antinodes to top and bottom need also to be considered. These are created with the nodes that are not fully consumed to create antiedges among regular antinodes, following the same approach to compute weights. It should be noted that links between top and bottom are impossible.

4.2 Diagram layout

Antigraphs serialized in JSON are sent to the front end where they are visualized as diagrams. The layout of these diagrams is computed using a force-directed algorithm [14]. Antinodes repel each other according to Coulomb's law as if they were electrically charged particles with the same signal. Antiedges bind them together as springs following Hooke's law.

The top and bottom antinodes, as well as the antiedges connecting them, are ignored in this process. The layout is performed in a rectangular area that acts as a boundary that confines regular antinodes. Top and bottom antinodes are positioned respectively at the top and bottom of this rectangle, and antiedges connecting then are plotted perpendicularly to them.

One of the advantages of a force directed algorithm is that it adjusts to changes, either of window dimensions or in the number of nodes. This enables the selection of antinodes, choosing which to display and which to hide, with the quick readjustment of the layout. When an antinode is hidden so are the antiedges that link to it.

Antigraphs with a large number of antinodes tend to have an even larger number of antiedges, which may difficult their visualization. In this case, the natural candidates to hide are those with smaller weight since they represent a smaller number of edges in the semantic graph. To simplify this kind of selection the antigraph browser provides a node weight threshold. If this threshold is provided then antinodes are sorted by weight and their accumulated weight is computed in this order. When this value exceeds the threshold the remaining antinodes are hidden, as well as the antiedges linking to them.
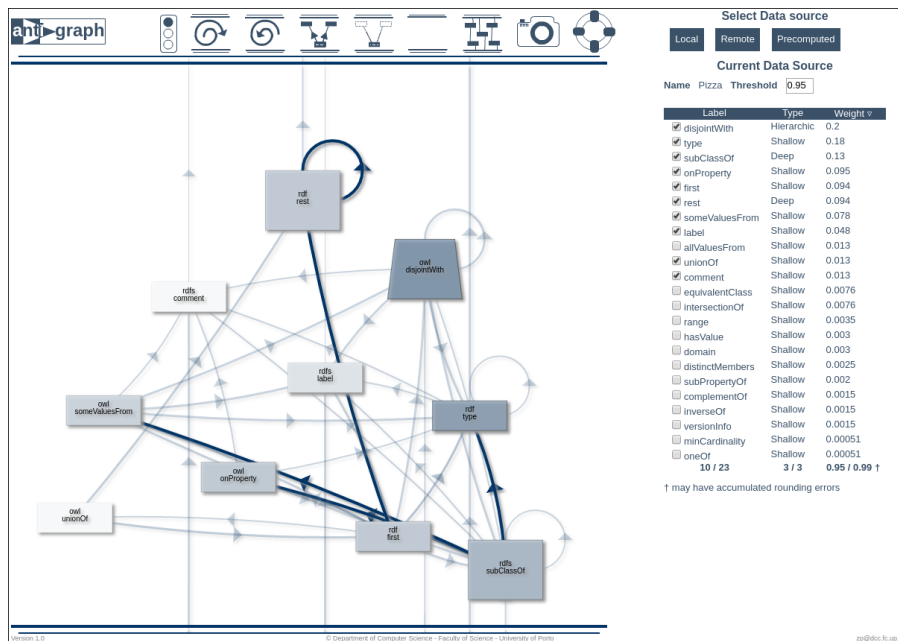
Figure 5 Antigraph browser.

4.3 User interface

Figure 5 depicts the user interface of the antigraph browser available online. The main part is the left central region where the diagram is displayed, following the approach described in the previous subsection. Above this area, there is a toolbar with tools for controlling the diagram layout. The smaller panel on the right contains a data source selector and displays the current data source features. The remainder of this subsection describes these panels in detail.

The antigraph browser has a number of features to control the diagram layout. These features are accessible trough the icons on the header toolbar. To start with, the incremental layout can be toggled on and off using the traffic light icon, on the left of the toolbar. The icons to its left provide ways to show and hide antinodes, as well as the antiedges connecting them. The most relevant (with higher weight) hidden antinode is shown by pressing the outward spiral icon. Using this tool it is possible to gradually enlarge the diagram. The reverse tool, bound to the inward spiral icon, hides the least relevant shown antinode.

The following two icons operate on the currently selected antinode: to show all currently hidden antinodes connected to it, or to hide all antinodes connected to it. Antinodes are selected just by clicking on them. Clicking an antinode with the mouse's middle button also toggles a tool tip hovering the node. This tool tip displays the characteristics of the antinode, such as label, type and weight.

The hide all and show all tools allow the user to set the layout at the two extremes. These tools are respectively bound to the icons with an antigraph with no antinodes and the antigraph with several antinodes and antiedges. The header toolbar includes two other icons on its right side: the camera icon and the life saving icon. The later open a help window expanding the information in this paragraph.

The camera icon produces a *vector* image of the diagram presented in the browser. Using the normal browser features, it is possible to obtain a raster image of the diagram. However,

this kind of image is inadequate for publication since it has a fixed and typically low resolution. The camera icon activates a feature that produces an SVG file with the diagram, using the same layout algorithm described above. This conversion uses the SVGKit[11] package, that works well for graphic primitives (e.g. lines, rectangles, ellipses) but has some limitations regarding fonts and shadows. The vector images look slightly different from their raster counterparts, but have better quality when printed. The diagrams of the next section, as well as those of Subsection 3.2, where produced using this tool.

The antigraph browser presents a second panel next to the diagram. Depending on the width of the web browser's window, this panel may be placed either on the right side (as in Figure 5) or bellow the diagram. The panel contains a data source selector and displays the main features of the current data source.

The upper part of the side panel is used for selecting a semantic graph as data source for generating an antigraph. It provides three kinds of semantic graph sources: local, remote and precomputed.

Local sources include small examples for testing the basic features of antigraphs, and were presented in Subsection 3.2. The dialog box for the selection of local graphs presents the RDF triples that will be processed to produce the antigraph. These triples are in N-Triples format in an editable window. The user may modify, add or delete these triples, to better understand how these changes are reflected on the antigraph diagram.

The remote sources are RDF graphs available on the web in XML/RDF format. This dialog box presents each graph's URLs and a threshold – the weight above which antinodes are included in the diagram. The last entry of this dialog box allows the user to enter an URL to any RDF/XML file available on the web, and assign it an initial threshold. This threshold may be changed later on the current data source panel.

Both the local and remote data sources are processed on the fly by the server. The precomputed data sources provide access to larger semantic graphs that require long processing times and are already available on the client side. Most of these examples are analyzed in detail in the next section.

The current data source panel displays its name, threshold and a grid listing its antinodes. This grid lists all the antinodes in the antigraph, showing which are currently visible, their type and weight. By default, this information is ordered by descending antinode weight, but the user can change it. The user can also (de)select the visible antinodes, which immediately changes the diagram layout. Also, changes in the diagram resulting from the tools described above are also immediately reflected in this grid.

5 Validation

This section shows with concrete examples how antigraph diagrams emphasize the most relevant path patterns of a semantic graph. It also explains how the tools in the antigraph browser help the discovery of path patterns in large semantic graphs, by temporarily hiding some of their antinodes and the antiedges connecting them, thus producing meaningful diagrams with a reasonable small size.

---

[11] http://svgkit.sourceforge.net/
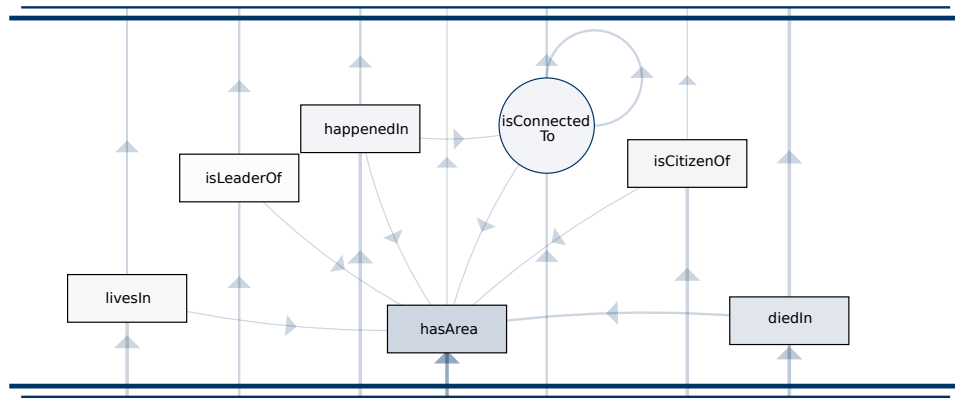
■ **Figure 6** Yago core – antinodes connecting to *hasArea*.

5.1   WordNet

Wordnet[8] 2.1, whose antigraph diagram is depicted in Figure 1, is a much larger graph than those presented in Subsection 3.2. However, this figure refers only to 95% of Wordnet 2.1 since the 5% least representative edges are omitted. By default, when this example is selected the threshold is set to 95%, but this value may be edited or removed in the corresponding field.

The WordNet 2.1 graph has 27 types of nodes and their corresponding antinodes would clutter this figure. This approach quickly produces a simple visualization by temporarily hiding the 2/3 least representative antinodes, i.e. edge types. It is important to point out that this is not specific of WordNet. All the semantic graphs tested with the antigraph browser have most of their paths concentrated in a fairly small number of edge types, hence this approach can be systematically used to improve the antigraph visualization.

This diagram immediately shows that the edges types that participate in most triples are from imported namespaces – *rdf:type* and rdfs:label – since the corresponding antinodes are darker. Two antinodes of the *wn20schema* namespace stand out from the pack for having links to several others, namely *hyponymOf* and *containsWordSense*, but the former participates in more "joins", as evidenced by the darker antiedges.

WordNet is frequently used as a semantic proxy by path based semantic measures [10]. These measures rely on taxonomic relationships to identify a least common ancestor between two concept nodes and compute the shortest path between them. Taxonomic relationships are created using *partOf* (hierarchical) and *isA* relationships. For instance, the RDF and RDFS vocabularies provide the `rdf:type` and `rdfs:subclassOf` properties that can be used to create a taxonomic relationship between typed resources. However, in this version of WordNet `rdf:type` is available, but `rdfs:subclassOf` is missing.

The antigraph diagram in Figure 1 shows an alternative hierarchic relationship – `hyponymOf` – that complemented with other relationship can be used to create a taxonomic relationship. The `rdfs:label` relationship is connected by an antiedge with `hyponymOf`, hence they can be combined to create a taxonomic relationship on words.

Of course, this is not new knowledge. It is well known that WordNet can be used as a semantic proxy using `hyponymOf` and another property to create a taxonomic relationship. The point is that the antigraph diagram highlights the most promising candidates to create a taxonomic relationship. This should be useful to discover candidates for taxonomic relationships in even larger semantic graphs, such as DBpedia [4].

Listing 1 SPARQL query to count leaders of geographic areas.

```
SELECT COUNT(*)
WHERE {
  ?p   yago:isLeaderOf ?g.
  ?g   yago:hasArea ?a.
}
```

Listing 2 Counting places connected to where something happened.

```
SELECT COUNT(*)
WHERE {
  ?s yago:happendIn ?g .
  ?g yago:isConnectedTo ?p .
}
```

5.2   Yago

Yago[12] [13] is a well known semantic knowledge base derived from several sources, such as
DBPedia, WordNet, and GeoNames. It has over 10 million entities but for this study, only
the core was used and labels were omitted. Still, this corresponds to over 20 million triples
with 60 property types. Hence it produces an antigraph with that order and size 487. Even
with a threshold of 80%, as it is by default on the antigraph browser, it is difficult to grasp.

The diagram in Figure 6 was obtained by selecting a single antinode, *hasArea*, the second
most frequent edge type in this graph. Afterward, it was used the tool that unhides antinodes
connected to the one currently selected. The point is to find property types related to concepts
that have an area. Examples of such concepts would be cities, regions or countries. In a sense,
hasArea can be seen as a defining property for a class of geographic concepts, although that
is not explicit. The diagram shows that these geographic concepts are connected to other
properties, such as *livesIn*, or *isLeaderOf*. That is, it is possible to retrieve information about
who lives in or who is the leader of an concept that has an area. The SPARQL query in
Listing 1 should produce a nonempty result set. In fact, it was checked on a Yago SPARQL
endpoint[13] and the result is 5666.

Also, one can determine the area of entities where something happened, *happenedIn*, or
that are connected to each other. The type of this last antinode is cyclic, meaning that its
corresponds to a reflexive edge type. These two antinodes are the only that are directly
connected without using em hasArea. Hence, it must be possible to obtain a non empty
answer to the query "what places are connected to the place where something happened?",
using the query in Listing 2, and it actually returns 888 solutions.

Surprisingly, the graph also indicates that one should not expect results for the query
"what places are connected to place that is lead by x" since these two antinodes are not
connected. Running the SPARQL query in Listing 3 verifies that conclusion as the result
is zero.

---

[12] https://www.mpi-inf.mpg.de/yago-naga/yago
[13] https://linkeddata1.calcul.u-psud.fr/sparql

Listing 3 Are citizens connected to other places?

```
SELECT COUNT(*)
WHERE {
  ?s yago:isCitezenOf ?g .
  ?g yago:isConnectedTo ?p .
  }
```

6   Conclusions and future work

Semantic graphs are hard to visualize due to a large number of typed nodes and edges. The antigraph approach to abstract semantic graphs maps edge information into antinodes and node information into antiedges. This reversal in the nature of the main constituents of a graph is the reason for the prefix "anti". The abstraction mapping produces a smaller graph that is easier to visualize and highlights the patterns of paths in the original semantic graph.

Antinodes and antiedges are assigned with weights that reflect the relevance of the edges and nodes they represent, and that can be used for further abstractions. For instance, antinodes with small weights, corresponding to types of edges that seldom occur in the semantic graph, can be omitted to unclutter large antigraph diagrams.

The antigraph diagram is the proposed graphical syntax to represent antigraphs, and thus visualize the semantic graphs. This kind of diagrams uses different shapes to represent antinodes according to their types, and transparency to denote weights. The special antinodes top and bottom are represented as parallel lines respectively on the top and bottom of the diagram. In an antigraph, diagram paths are in general upwards, which facilitates their detection.

The web application for visualizing and interacting with antigraphs is also an important contribution of this research. It uses a force direct algorithm, which allows the incremental layout of the diagram after reposition or removal of antinodes. This application can use data from different sources: local data entered on the interface, remote data available on the web and precomputed data for a few preprocessed semantic graphs.

The proposed approach still faces the challenge of dealing with massive semantic graphs with millions of triples, such as those of Yago and DBpedia. The major problem is due to the computational complexity involving antiedge weights. However, there are approaches to curb this complexity that are currently being researched. After tackling this issue, the antigraph browser will be easier to evaluate with real users interested in discovering path patterns in large semantic graphs.

──── **References** ────

1   James Abello, Frank Van Ham, and Neeraj Krishnan. ASK-GraphView: A large scale graph visualization system. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):669–676, 2006.

2   Daniel Archambault, Tamara Munzner, and David Auber. GrouseFlocks: Steerable exploration of graph hierarchy space. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):900–913, 2008. `doi:10.1109/TVCG.2008.34`.

3   David Auber. *Tulip — A Huge Graph Visualization Framework*, pages 105–126. Springer, 2004. `doi:10.1007/978-3-642-18638-7_5`.

4 Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *The Semantic Web*, pages 722–735, 2007.

5 Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *8th International AAAI Conference on Weblogs and Social Media*, pages 361–362, 2009.

6 Fabio Benedetti, Sonia Bergamaschi, and Laura Po. A visual summary for linked open data sources. In *International Semantic Web Conference*, 2014.

7 Nikos Bikakis, John Liagouris, Maria Kromida, George Papastefanatos, and Timos Sellis. Towards scalable visual exploration of very large RDF graphs. In *The Semantic Web: ESWC 2015 Satellite Events*, pages 9–13. Springer International Publishing, 2015. `doi: 10.1007/978-3-319-25639-9_2`.

8 Christiane Fellbaum. *Wordnet: An electronic lexical database.* MIT Press, 1999.

9 Nicola Guarino, Daniel Oberle, and Steffan Staab. What is an ontology? In *Handbook on ontologies*, pages 1–17. Springer, 2009.

10 Sébastien Harispe, Sylvie Ranwez, Stefan Janaqi, and Jacky Montmain. Semantic similarity from natural language and ontology analysis. *Synthesis Lectures on Human Language Technologies*, 8(1):1–254, 2015.

11 Tuukka Hastrup, Richard Cyganiak, and Uldis Bojārs. Browsing linked data with Fenfire. In *Linked Data on the Web*, 2008.

12 Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. RelFinder: Revealing relationships in RDF knowledge bases. In *Semantic Multimedia*, pages 182–187, 2009. `doi:10.1007/978-3-642-10543-2_21`.

13 Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pages 3161–3165, 2013.

14 Stephen G. Kobourov. Spring embedders and force directed graph drawing algorithms. *CoRR*, abs/1201.3011, 2012. URL: `http://arxiv.org/abs/1201.3011`.

15 Zhiyuan Lin, Nan Cao, Hanghang Tong, Fei Wang, U. Kang, and Duen Horng Polo Chau. Demonstrating interactive multi-resolution large graph exploration. In *Proceedings of the 2013 IEEE 13th International Conference on Data Mining Workshops*, pages 1097–1100, 2013. `doi:10.1109/ICDMW.2013.124`.

16 Kang Zhang, Haofen Wang, Thanh Tran, and Yong Yu. ZoomRDF: semantic fisheye zooming on RDF data. In *19th international conference on World Wide Web*, pages 1329–1332, 2010.

17 Michael Zinsmaier, Ulrik Brandes, Oliver Deussen, and Hendrik Strobelt. Interactive level-of-detail rendering of large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2486–2495, 2012. `doi:10.1109/TVCG.2012.238`.

Comparison of Segmentable Units as Indicators of Two Texts Being Parallel

Afonso Xavier Canosa

University of Santiago de Compostela, Galiza, Spain
canosarodrigues@gmail.com
https://orcid.org/0000-0002-8767-3640

Abstract

A bitext produced from a Portuguese historical text and its English translation, Fernão Mendes Pinto's *Pilgrimage*, serves as a case study to describe the creation of a parallel corpus and investigate which linguistic and textual units are the best indicators of alignability. The process of building the corpus goes through preparation of transcriptions, annotation, segmentation and sentence alignment. Once the bitext is ready, the corpus is used to inquire which units appear as more relevant to predict that both texts are parallel. From the largest content units, those of chapters, to sentences, word types, tokens and characters, the latest, despite being the unit with less textual and linguistic significance, were found to be the best indicator of both texts being alignable.

1 Introduction

Parallel corpora are increasingly important for the development and evaluation of machine translation and Natural Language Processing applications. Yet, parallel texts can serve more specific research purposes, such as careful examination and comparison of versions and translations in classical humanities research. This is the case of Tartaria, a parallel corpus created from the the section that describes Fernão Mendes Pinto's stay with the Tartars, comprising chapters 117-131 from the Portuguese first edition *Peregrinacam* (PT 1614)[1] and chapters 38-41 from its English version (EN 1653)[2]. As translation was one of the reasons for misreadings of Pinto's report (e.g. Figure 1 the exotic term *bada* is translated as rhinoceros without any apparent motivation in the source), a parallel corpus allows researchers to detect those segments that they may be more interested in and focus on relevant sentences only, allowing for optimization of expensive and time-consuming translation tools. The expected result should output a table with two texts, a source and its translation, in such a disposition that enables an easy comparison of both (Figure 1). The process of creating this parallel corpus required a balance between machine and human-performance. One of the first tasks to solve was to find out if the English version is a direct translation of the Portuguese text, or, on the other hand, if the target only follows a narrative and offers an independent free version of the source. To answer this question without direct inspection of both texts, textual units

---

[1] http://purl.pt/82

[2] http://purl.pt/16425

| 200 | E auido entre ambos cõselho sobre esta noua, assentaraõ de mãdarem as embarcaçoẽs todas quatro a Huzamguee, & elles ambos com poucos dos seus irẽse por terra a Fanaugrem onde tinhão por nouas que el Rey estaua, o que logo se pós em effeito co parecer tambem desta princesa, a qual lhes mãdou dar todas as caualgaduras que ouuerão mister para sy & para os seus, & oito **bada**s para leuarem o seu fato. | When as they had consulted a while upon these news, they resolved to send their four vessels away to Usamguee, and themselves to travel by land to Fanaugrem, where they understood the King was. This deliberation taken they put incontinently into execution, & that by the advice of this Princess, who for that purpose caused them to be furnished with horses for themselves, & their people, as also with eight Rhinocerots for the transportation of their baggage. |

■ **Figure 1** Example of alignment used to research the translation of the term *bada*.

can be evaluated in order to search for the one that shows a closer correspondence between both texts, hence serving as evidence for alignability of source and target in a relation of direct translation.

2 Parallel corpora and units considered in alignments

Bilingual and multilingual corpora are core resources for the training and evaluation of automate machine translation and natural language processing tools [4, 1]. A distinction can be made between corpora that represent direct translations and those that are only comparable, showing a similarity in content, yet not being a literal translation of each other [14, 12]. In a parallel corpus, texts are aligned so that a direct correspondence is made between text sequences from one language to the other. The final alignments show not only sentences with the same content, but also omissions (1:0), additions (0:1), and more complex correspondences when added or omitted text comes together with perfect matches [8]. The corpus can be further enriched through annotation and is usually encoded with standard schemas [9], though output formats vary depending on the final use [8].

It is in the field of automate alignment that the issue of which units represent an indicator of two texts being parallel appears as a relevant question. Two units are considered in common sentence alignment algorithms [8, 10]: sentence length [2] and word matches [5]. The use of both resulted in hybrid solutions [6, 11]. Even more elaborated models claiming to outperform previous hybrid methods [7], use the semantic similarity of sentences as a result of computing TF-IDF values (hence word-based) across each language to later align target and source based on sentence metrics (sentence again).

3 Tartaria parallel corpus

The process of building a parallel corpus of the Portuguese and English chapters related to the Tartars in Fernão Mendes Pinto's travels was aimed not only at producing an output for comparative studies, but also for research on NLP tasks such as NERC. An experimental approach was considered to find a balance between automation and the need for a high quality product, only achieved with human validation.

3.1 Transcriptions

The first step was text transcription. Although the Portuguese version of Pinto's travels has been partially transcribed and published online [3, 13], the chapters relevant to Tartaria are not included in public corpora. It was possible to find, though, a transcription of the whole 1614's text from a publishing house. Even if the provided transcription had some important unreported errors (gaps of whole pages and typos), those chapters relevant for the Tartar corpus could be used without any modification other than small typos, validated against the facsimile of the DLNLP.

```
-<text>
 -<div type="chapter" n="38">

   <head>CHAP. XXXVIII</head>
  -<head>
   -<div2 type="page" n="149">
      A Tartar Commander enters with his Army into the Town of Quincay,
      the taking of it by the means of some of us Portugals.
     </div2>
    </head>
  -<div1>
   -<div2 type="page" n="149">
      WE had been now eight months and an half in this captivity, where
```

◼ **Figure 2** Excerpt showing all elements and attributes required to annotate Tartaria corpus.

The English text was in-house transcribed following the edition available at the DLNLP. Despite the use of OCR to start, the whole process of transcription required careful manual revision. Hyphens and reformed words at the end of the line were discarded and a regular font format was used for the whole text, even if the original displays place names, demonyms and anthroponyms in italics. Missing characters were marked with square brackets. During the process of text alignment, once the corpus was already built, some words still needed correction, usually due to confusion of similar characters such as *s* and *f*.

At the end of this stage there were two text files containing raw transcriptions of the first editions.

3.2 Annotation

The corpus was annotated for main structural elements (Figure 2), showing chapters, pages (folios in the Portuguese edition) and divisions to distinguish chapter headings from main content.

Geographical named entities were annotated in each corpus using NERC tools and human-validated to produce a gold standard of annotated place-names.

At the end of this stage the corpus comprised two annotated documents showing chapter, title, main text and either folio or page components.

3.3 Text segmentation

Each text was segmented in chapters and sentences. A direct observation in chapter segmentation is that there is no direct correspondence between the number of chapters in both languages. As a first intuition, this could be explained by the English version bringing more content together, but by important omissions in content as well. The purpose of creating an aligned corpus was also to answer this question and find content gaps if any.

As all chapters have a heading with a pseudo-paragraph, and there is a different number of chapters, it was obvious from the very beginning that some sentences would result in an omission in the translation. A script parsing the annotated corpus stripped tags and split sentences using dots, semicolons, exclamation and interrogation marks as delimiters. Regular expressions handled exceptions for chapter headings in 1653's text, where semicolons have a function similar to that of commas.

■ **Table 1** Processed segments during corpus preparation. *Tag*: annotated with specific tag. *Script*: retrievable using a script to parse annotated text, even if the category is not annotated. *Db*: relational table in a database. *Txt*: file with raw text in txt format. *HTML*: displayed as web page.

Segment type	PT 1614	EN 1653	Total	Retrieval
Chapters	15	5	20	tag, script, db
Pages	36	20	56	tag, script, db
Sentences	222	353	575	script, db, txt
Word types	3401	2806	6027	script
Tokens	18040	19159	37199	script
Aligned sentences	240	230	470	script, db, HTML

3.4   Bitext

Hunalign[3] [11], an open source for automate alignment, was first applied to create the parallel structure of the corpus. Test rounds considered the use of an in-house built dictionary with the 100 words of highest frequency and a gazetteer built from geographical named entities from the NERC annotation of the Portuguese corpus. The best result was converted to a table with two columns, one for the source language, another for the translation. Even if the automate alignment brought related sentences close enough to prefer this procedure over a bitext produced manually from scratch, in order to obtain a golden corpus, results had to be manually corrected and validated. Misaligned rows were arranged and grouped to fit the original style of the source where sentences, defined as a stretch of text ending in a given punctuation mark, are more similar to pseudo-paragraph than to sentences as perceived by modern standards.

4   Results

Through the process of annotation and sentence alignment, a web environment enabled the visualization of the results and served as a repository for the experimental data and related products. Table 1 shows the textual segments obtained in the final parallel corpus.

A web-based interface allows the retrieval of text in different dispositions for either the bitext or any of the Portuguese and English versions only. As an example, figure 3 shows how the tagged text from figure 2 is displayed in a more readable format.

The following units were considered for visualization outputs.

Chapters. A series of scripts evaluates the number of chapters and displays them in HTML format.

Pages. The text grouped by pages following the disposition as in the original first editions.

Sentences. List of sentences using punctuation marks as delimiters. This was also the starting point to generate the bitext. The final segment is, however, not the result of delimiters only, but of the match of both versions for alignment. It is not always the case that an aligned row contains only one grammatical sentence in either the source or the translation. A larger unit, pseudo-paragraphs, may apply. Nevertheless, neither term defines the category well. Apart from those alignments where sentence is an accurate description of the matched segments, there are also complex and compound-complex

---

[3] http://mokk.bme.hu/resources/hunalign/

Showing 5 chapters from *Tartaria Corpus EN (1653).*

(1) Chap. 38

Page 149

CHAP. XXXVIII
A Tartar Commander enters with his Army into the Town of Quincay, and that which followed thereupon; with the Nauticors besieging the
Castle of Nixiamcoo, and the taking of it by the means of some of us Portugals.

WE had been now eight months and an half in this captivity, wherein we endured much misery, and many incommodities, for that we had
nothing to live upon but what we got by begging up and down the Town, when as one Wednesday, the third of July, in the year 1544, a little
after midnight there was such a hurly burly amongst the people, that to hear the noise and cries which was made in every part, one would

Figure 3 Excerpt retrieved from the web environment showing chapters in English.

sentences split in two different alignments. These units fall two levels below pseudo-paragraphs in a hierarchy of text segments. As sentence was the initial term to describe this segment type, aligned sentences was kept as the most representative categorical label. The actual aligned unit represents a balance between single sentences and pseudo-paragraphs made of coordinated sentences (part of a bigger segment which is still a sentence).

Words. Ranking of word types arranged by its Zipfian distribution and aligned in two rows, PT (1614) and EN (1653). Lexical items with highest frequencies and named entities were selected to create dictionaries for the automatic text alignment tests.

Aligned sentences. An alignment considers an empty sentence when there is no counterpart in either source or translation resulting in an omission (1:0). Relevant complex alignments (m:n) appear in pseudo-paragraphs with embedded transcriptions of the language of the Tartars along with the Portuguese translation (hence allowing for a triple alignment). The whole corpus shows all the alignment types as a list in a table following the narrative sequence.

5 Discussion: which units show evidence of both texts being alignable?

Initial analysis was directed towards answering if both texts were able to generate a parallel corpus. This would mean that for most segments in the source PT 1614 (L1) from chapter to sentence, there is an equivalent segment in the target translation EN 1653 (L2). It could be the case that both texts were not direct translations of each other, but just an account of similar events following a common narrative, though still comparable corpora. This may still allow an alignment at the top of the hierarchy, chapters in our segmentation. On the other hand, if the target is a literal translation, an alignment at the level of sentences is expected. Another possibility is neither chapters nor sentences having the same number of units in L1 and L2, as in a less literal translation that modifies text disposition. In this last case, some units may show no equivalents, though some others would be expected to emerge as indicators of both texts having a translation relation. The procedure was finding out which category allows evaluation in terms of L1 → L2 having ratio 1:1, that is, a proportion of 50% for both L1 and L2 taking the corpus as a whole. Figure 4 shows a comparative graph of the size of the corpus for each language in absolute and relative terms.

Very early in the process, it was noticeable that the highest segmented category, chapters, shows dissimilarity (L1 75%, L2 25%). Each chapter has a heading that adds extra blank lines to a page, so more chapters would slightly affect the number of pages too. This extra
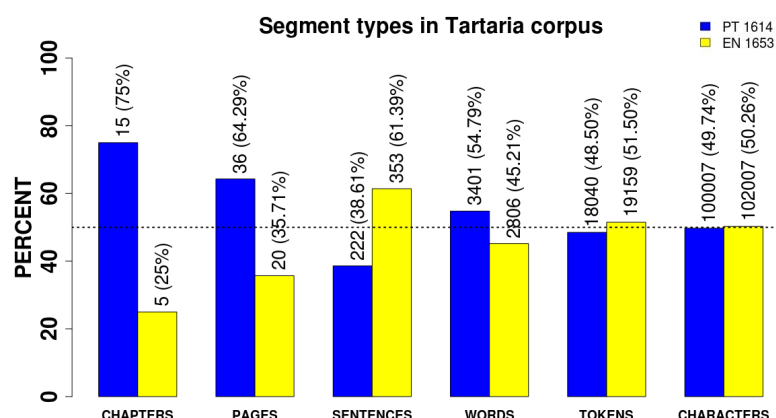
Figure 4 Comparative graph of segment types in Tartaria corpus.

content is not, however, conclusive enough for the different number of pages. Issues such as typography, page size and layout, not considered in the annotation, may explain a different number of pages in both editions.

The number of sentences split by punctuation marks has unbalanced ratios too (L1 38%, L2 61.39 %). It is worth noting that L1 < L2, hence there is a contradiction with the higher categories (chapters and pages) where L1 > L2. Again, non-considered variables such as editorial preferences, different punctuation standards and more grammar-dependent syntactic disposition of clauses may also explain the different number of sentences in each language regardless of both texts being alignable.

The category of word types still shows a difference between both texts (L1 54.79%, L2 45.21 %). A direct observation of the data shows that morphological features are relevant factors to explain word forms variability. The word form at the top of the Zipfian distribution is the determinate article with ratio 4:1 and values L1 (a, o, os, as) : L2 (the). However, following the same example, the fact that the same word form has more than one different expression in another language, does not necessarily affect the number of tokens. Thus, tokens, the variable appearing as a direct measure of text length, show a more balanced ratio 1 : 1.07 (L1 48.5%, L2 51.5%), an indicator of one text being a translation of the other.

Finally, characters show the closest balance. In fact, round percentages stand for the desired 1:1 (L1 49.74%, L2 50.26%). An explanation for this highest accuracy is that characters do not only represent a similar number of tokens in both texts, but also capture some phonetic and morphological properties of words. In fact, if a word in the source language is polysyllabic or has a derivational or compound structure, its equivalent is most often expected to show a more complex structure in the target language too.

6 Conclusion

Different units were compared to research which one would be the best predictor of two texts being alignable in terms of source and translation. Characters show a parallel ratio, around 1:1, becoming the most accurate feature for predicting the alignability of both texts. From a linguistic point of view, it is intriguing that a unit without semantic value stands as more relevant than morphologically rich and syntactic relevant tokens and word types. The inferred hypothesis to consider for future work is that, when used as a measure of length for

the largest units, characters capture some aspects of the morphologic-syntactic structure. Although they have been extensively used as indicators in text-alignment tasks, to the best of my knowledge, the linguistic implications of such a basic and easily observable phenomenon have not been explained and are still open for further research.

References

1   Christian Buck and Philipp Koehn. Findings of the WMT 2016 bilingual document alignment shared task. In *First Conference on Machine Translation – Shared Task Papers*, volume 2, pages 554–563, 2016.

2   William A. Gale and Kenneth W. Church. A program for aligning sentences in bilingual corpora. *Computational linguistics*, 19(1):75–102, 1993.

3   Charlotte Galves, Aroldo Leal de Andrade, and Pablo Faria. Tycho brahe parsed corpus of historical Portuguese. `http://www.tycho.iel.unicamp.br/~tycho/corpus/texts/psd.zip`, 2017.

4   Philipp Koehn. EuroParl: A parallel corpus for statistical machine translation. In *Machine Translation Summit*, pages 79–86, 2005.

5   I. Dan Melamed. A geometric approach to mapping bitext correspondence. *CoRR*, 1996. URL: `http://arxiv.org/abs/cmp-lg/9609009`.

6   Robert C. Moore. Fast and accurate sentence alignment of bilingual corpora. In *Conference of the Association for Machine Translation in the Americas*, pages 135–144, 2002.

7   Xiaojun Quan, Chunyu Kit, and Yan Song. Non-monotonic sentence alignment via semisupervised learning. In *51st Annual Meeting of the Association for Computational Linguistics*, volume 1, pages 622–630, 2013.

8   André Santos. A survey on parallel corpora alignment. In *Master of Insformatics Internal Conferece, Universidade do Minho*, pages 117–128, 2011.

9   Alberto Simões and Sara Fernandes. XML schemas for parallel corpora. In *XATA 2010: 9ª Conferência Nacional em XML, Aplicações e Tecnologias*, pages 59–69, 2011.

10 Hai-Long Trieu, Phuong-Thai Nguyen, and Kim-Anh Nguyen. Improving moore's sentence alignment method using bilingual word clustering. In *Knowledge and Systems Engineering*, pages 149–160, 2014. `doi:10.1007/978-3-319-02741-8_14`.

11 Dániel Varga, Péter Halácsy, András Kornai, Viktor Nagy, László Németh, and Viktor Trón. Parallel corpora for medium density languages. In *Recent advances in natural language processing IV : selected papers from RANLP 2005*. John Benjamins, 2007.

12 Krzysztof Wołk and Krzysztof Marasek. Unsupervised comparable corpora preparation and exploration for bi-lingual translation equivalents. *CoRR*, 2015. URL: `http://arxiv.org/abs/1512.01641`.

13 Marcos Zampieri and Martin Becker. Colonia: Corpus of historical Portuguese. In *Nonstandard Data Sources in Corpus-based Research*. Shaker Verlag, 2013.

14 Federico Zanettin. *Translation-driven corpora: Corpus resources for descriptive and applied translation studies*. Routledge, 2014.

Less is more in incident categorization

Sara Silva
Instituto Universitário de Lisboa (ISCTE-IUL) Lisbon, Portugal
satsa@iscte-iul.pt

Ricardo Ribeiro
INESC-ID Lisboa
Instituto Universitário de Lisboa (ISCTE-IUL), Lisbon, Portugal
ricardo.ribeiro@iscte-iul.pt
🆔 https://orcid.org/0000-0002-2058-693X

Rubén Pereira
Instituto Universitário de Lisboa (ISCTE-IUL) Lisbon, Portugal
Ruben.Filipe.Pereira@iscte-iul.pt
🆔 https://orcid.org/0000-0002-3001-5911

── Abstract ──

The IT incident management process requires a correct categorization to attribute incident tickets to the right resolution group and obtain as quickly as possible an operational system, impacting the minimum as possible the business and costumers. In this work, we introduce automatic text classification, demonstrating the application of several natural language processing techniques and analyzing the impact of each one on a real incident tickets dataset. The techniques that we explore in the pre-processing of the text that describes an incident are the following: tokenization, stemming, eliminating stop-words, named-entity recognition, and TF×IDF-based document representation. Finally, to build the model and observe the results after applying the previous techniques, we use two machine learning algorithms: Support Vector Machine (SVM) and K-Nearest Neighbor (KNN). Two important findings result from this study: a shorter description of an incident is better than a full description of an incident; and, pre-processing has little impact on incident categorization, mainly due the specific vocabulary used in this type of text.

1 Introduction

An incident is defined by ITIL as "An unplanned interruption to an IT service or reduction in the quality of an IT service". These incidents can be related with failures, questions, or queries and should be detected as early as possible [7]. It is crucial to have an appropriate incident classification, the process that assigns a suitable category to an incident, so they are routed more accurately [7]. Automating incident classification process means to avoid human error, reducing the waste of resources and avoiding incorrect routing due to wrong classification [3]. To automate the classification is relevant to take into account the structure of the incidents. The incident tickets are composed by several attributes, which are mandatory fields when the user is recording the incident in the Incident Ticket System (ITS). Two attributes are crucial for the categorization process: the short and full descriptions are the key attributes to

obtain a good classification performance. The dataset used to test and assess the algorithms contains information about real-word incident tickets provided by a specific company. Due to privacy questions, the company cannot be mentioned and the dataset is not available.

The remainder of this document consists of five sections that are structured as follows. Some related work is presented in Section 2. The proposed method is presented in Section 3. The implementation is described in Section 4. Section 5 presents the obtained results. Finally, the document closes with the conclusions and some possible future work.

2   Related Work

Over the years, approaches that automate Incident Management (IM) and particularly incident classification have being studied and developed. In this section, we describe relevant work developed in this area, used algorithms, and the results obtained with the respective implementations. To automate IM, the authors of [4] use Machine Leaning (ML) and information integration techniques to develop an algorithm for correlating incoming incidents. The authors used the incident description to extract keywords and their annotations as features. SVM is the algorithm used to attribute a category to an incoming incident. This approach generates the list of keywords which better identifies each category. In [1], the authors used for the same problem, incident tickets categorization, SVM, KNN, decision trees, and Naïve Bayes. They used four datasets with different categories. In this classification process, they present three approaches for each algorithm: accuracy results using TF-IDF, using only TF, and using boolean weighting. On average, SVM had accuracy results of 90% approximately, in the three approaches. KNN achieves 75% of accuracy, also with the three approaches. Decision trees have similar results in the three approaches, of around of 90%. Finally, Naïve Bayes was the only approach which presented different results for the three approaches: boolean weighting with 85% and TF-IDF and TF, both with 55% of accuracy.

3   Proposed Method

To propose the best approach to automatically categorize an incoming incident there are critical steps that have to be taken into account in order to ensure a good performance of the classifier. The automatic incident categorization is possible due to the data related to each incident. The main attribute for the categorization of an incoming incident is the description of the incident. There are two attributes related with the description assigned to each incident. These attributes are the short and the full description. Both consist in unstructured natural language text. The difference between both attributes is implicit in the name. Both descriptions are provided by the user who have created the incident in the ITS. The full description is a long and detailed description that contains all the relevant aspects to categorize the whole incident. The short description is much shorter than the detailed one, consisting in a summary, ordinarily one phrase that categorizes briefly the incident.

The analyzed dataset has incidents written in several languages like English, German, Spanish, French, and Portuguese. However, since the most common language is English, we have decided to choose English as the one to be studied and excluded all the incidents in other languages from our analysis. As referred in the previous section, there are other attributes that also contribute to the categorization, however, the short and full descriptions are the critical ones to obtain a good performance of the classifier, requiring a special process.

The dataset under study contains incidents that have several levels of categorization. This study only takes into consideration the first and second level. Moreover, we compare the performance achieved using the short and the full description, at both levels of categorization.

■ **Table 1** Incident ticket example.

Short description	Adobe Reader XI
Full description	User cannot open Internet Explorer.Error message displayed.
Category	Software
Subcategory	Managed Software Workplace
Caller id	User X
Affected Location	Location Y
Severity	4 - Low

Concerning the second level of categories, we explored two different approaches. The first one is performing the categorization assuming that the first-level category is correctly assigned to the incident. Basically, we use the first-level category as an attribute to build the classifier that assigns the second-level category to a given incident. The second one does not take into consideration the first-level categorization. Therefore, the incident is categorized with the same data that we use in the first-level categorization. When building a classifier to automatically assign a second-level category, we are also automatically assigning the respective first-level category. The set of first-level categories is composed by the ten following categories: application, collaboration, enterprise resource planning (ERP), hosting services, network, security and access, output management, software, workplace, and support. After assigning a first-level category to an incident, a second-level category is assigned. The set of second-level categories is composed by 94 categories: 47 belong to application, 9 to collaboration, 3 to ERP, 7 to hosting services, 6 to network, 3 to output management, 4 to security and access, 2 to software, 3 to support, and, finally, 10 to workplace.

In this work, we analyze the impact of different natural language pre-processing techniques, as tokenization, TF-IDF, stemming, stop word removal, and entities recognition on incident classification based on their textual description. We explore two classification approaches commonly used for this task: support vector machine (SVM) and K-Nearest Neighbors (KNN). SVM is an appropriate technique for text categorization, proving to be more robust than other conventional techniques of text classification [6]. KNN is considered to be simple, easy to implement [10] and a popular one in text categorization [11]. The goal is compare the different approaches and conclude which performs better.

4 Implementation

To train the classifiers, we use a dataset composed by incident tickets correctly classified with an appropriate first-level category and a second-level category. In this dataset, each incident ticket has a short description, a full description, a caller id, which is the person who opens the ticket, a severity, and, finally, the respective first-level (category) and a second-level (subcategory) categories, as it is possible observe in Table 1. For the first-level categories, we used 2000 incidents per category. For the second-level, we use a dataset with 2000 incidents per subcategory whenever was possible. The most times it was, however, there are second-level categories with less incidents in dataset.

We start by analyzing the impact of using the short and full descriptions on the first-level categories. Then, we consider that the first-level categorization is correct and analyze the difference of using short and full descriptions on the second-level categorization. Finally, we study the how second-level categorization performs when not using first-level information and, again, the impact of using the short and full descriptions at this level.

The obtained results of the different approaches are presented in the next section.
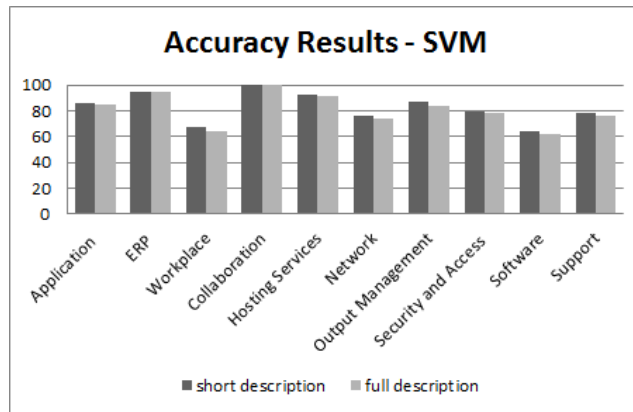
Figure 1 Short *vs* full description at the first-level categorization.
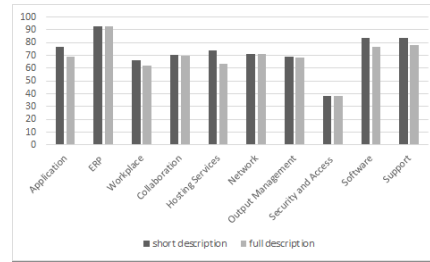
5 Results

To train and test the different techniques previously presented, we use cross-validation, which consists on dividing the dataset into subsets with the equal number of tickets. One subset is used to test the classifier, while the others are used to train the classifier. This process has the advantage of preventing overfitting, because the training data is fully independent of the test data [2, 5].

Figure 1 presents the accuracy results after applying the SVM algorithm, showing the difference between using the short and the full description. Previous work [9] showed that at the first-level categorization, the best results were achieved using an SVM (specifically, when compared to KNN). When using the short description, we achieve an accuracy of 83%. Surprisingly, when using the full description the accuracy decreases to 81%. Figure 2 presents the results related to the second-level categorization (considering the correct first-level categorization), also comparing the impact of using the short *versus* the full description. The same behavior is observed at this level, with both approaches, SVM and KNN, achieving better results using the short description. As we can see in Figure 2a, when using the SVM the overall accuracy for the short description is 86% and for the full description is 85%. Figure 2b shows the achieved results using the KNN algorithm: with the short description, we obtained 73% of accuracy and with the full description we obtained 69%. Overall, using the short description leads to better results, with the SVM consistently achieving the best results. The results of the second-level categorization when considering all the categories at this level without taking into account the first-level, as expected, decrease to accuracies of 75% using SVM and 65% using KNN.

Concerning the impact of pre-processing on the classification process, we start by analyzing two different tokenization strategies: alphabetic and word tokenization. The alphabetic tokenizer is a simple tokenizer that considers only tokens composed by alphabetic sequences. The word tokenizer is a standard word tokenizer that splits words according to predefined tokens, such as space, punctuation, etc. For this part of the work, we focus on the first-level categorization using the short description, since the best results were achieved using this attribute. Figure 3 presents the results of the application of the two tokenizers: with the alphabetic tokenizer achieving an accuracy of 83% and the word tokenizer achieving an accuracy of 82%.
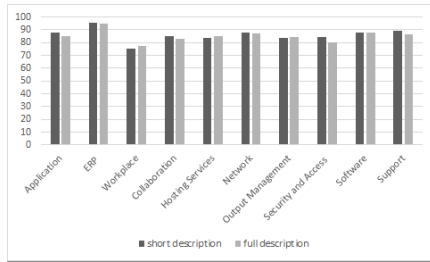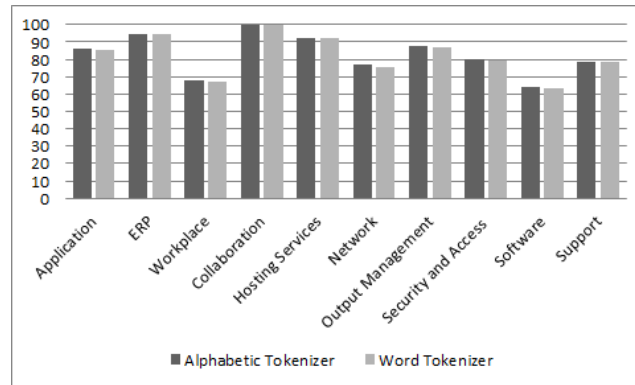
(a) Results using a SVM.

(b) Results using KNN.

Figure 2 Accuracy at the second-level categorization (considering the correct first-level categorization).

**Figure 3** Accuracy of the alphabetic tokenizer *vs* the word tokenizer.

Another aspect that we have explored was the descriptions representation. In that sense, we represent descriptions as feature vectors of term frequencies (TF), $log(1 + f_{ij})$, f_{ij} is the frequency of word i in document j (other dampening strategies could be used); inverse document frequencies (IDF), log(num of Docs/num of Docs with word i); and, TF × IDF. TF × IDF increases with the number of times a term occurs in a document, but is offset by the document frequency of the term in the corpus. The results are shown in Figure 4. As it can be seen, the results are very similar (accuracies of approximately 82%). Our intuition is that as we perform stop word removal and use short descriptions (small documents), the impact of varying the weighting scheme in the feature representation is not significant.

We also explored stemming. Stemming consists on reducing the words to their base form, lowering the number of entries of the dictionary. We compare the use of the Porter Stemmer [8] to not using stemming at all. The use of stemming did not have any impact on the results. Our intuition is that description representation did not benefit of the use of stemming due to the specific vocabulary used in incident description.

Finally, we explored named-entity recognition focusing on the identification of organizations and use them as features to improve the categorization. This process did not impact the results.

6 Conclusions and Future Work

Text processing plays an important role in incident categorization. In this work, we analyze different natural language processing techniques and evaluate their impact on a real incident tickets dataset. An interesting outcome of this study was that the use of the short description
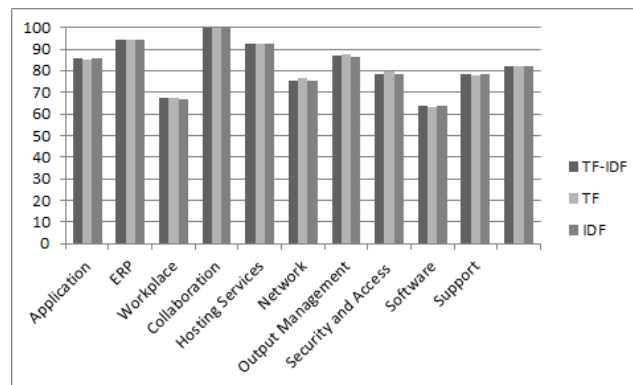
■ **Figure 4** Accuracy when using TF×IDF, TF, and IDF.

of an incident leads to a greater accuracy than using the full description, on the different levels of categorization (first level, 10 categories; second level, 49 categories) under analysis. A possible reason that we found to justify such finding might be the fact that when the user describes an incident with limited text that results in a greater focus on explaining the incident. On the other hand, in the full description the user has tendency to disperse.

This analysis is critical to produce a positive impact in the categorization process and is determinant to obtain a correct assignment and consequently improve the whole incident route. This paper is part of a work related with the integration of a module in an Incident Ticket System in a specific company. So as future work we plan to carry out the integration and assess its impact by performing interviews to the IT teams responsible for the Incident Management (IM) process. It is also intended to extend the categorization to the whole activity of classification (which includes a third level) and initial support in the IM process, which includes automating the assignment of a priority and urgency to incidents. Moreover, we pretend to automate the resolution and recovery activities, finding and suggesting automatically a possible resolution to an incoming incident.

References

1   Muchahit Altintas and A. Cuneyd Tantug. Machine learning based volume diagnosis. In *International Conference on Artificial Intelligence and Computer Science (AICS)*, pages 195–207, 2014.

2   Sylvain Arlot and Alain Celisse. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4:40–79, 2010. `doi:10.1214/09-SS054`.

3   Rajeev Gupta, K. Hima Prasad, Laura Luan, Daniela Rosu, and Chris Ward. Multi-dimensional knowledge integration for efficient incident management in a services cloud. In *IEEE International Conference on Services Computing*, pages 57–64, 2009. `doi:10.1109/SCC.2009.48`.

4   Rajeev Gupta, K. Hima Prasad, and Mukesh Mohania. Information integration techniques to automate incident management. In *IEEE Network Operations and Management Symposium (NOMS)*, pages 979–982, 2008. `doi:10.1109/NOMS.2008.4575262`.

5   Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. *BJU international*, 101(1):1396–1400, 2008. `doi:10.1177/02632760022050997`.

6 Thorsten Joachims. Text categorization with Support Vector Machines: Learning with many relevant features. In *Machine Learning: ECML-98*, volume 1398 of *Lecture Notes in Computer Science*, pages 137–142. Springer, Berlin, Heidelberg, 1998.

7 John O. Long. Service operation. In *Itil Version 3 at a Glance: Information Quick Reference*, pages 55–74. Springer, 2008. `doi:10.1007/978-0-387-77393-3_5`.

8 Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

9 Sara Silva, Rúben Pereira, and Ricardo Ribeiro. Machine learning in incident categorization automation. In *Proceedings of CISTI'2018: 13th Iberian Conference on Information Systems and Technologies*, 2018.

10 Yang Song, Jian Huang, Ding Zhou, Hongyuan Zha, and C Lee Giles. IKNN: Informative K-Nearest Neighbor Pattern Classification. *Proceedings of the European conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*, pages 248–264, 2007. `doi:10.1007/978-3-540-74976-9{\_}25`.

11 Bruno Trstenjak, Sasa Mikac, and Dzenana Donko. KNN with TF-IDF based framework for text categorization. *Procedia Engineering*, 69:1356–1364, 2014. `doi:10.1016/j.proeng.2014.03.129`.

NLPPort: A Pipeline for Portuguese NLP

Ricardo Rodrigues
CISUC / ESEC, Polytechnic Institute of Coimbra, Portugal
rmanuel@dei.uc.pt
 https://orcid.org/0000-0002-6262-7920

Hugo Gonçalo Oliveira
CISUC / Department of Informatics Engineering, University of Coimbra, Portugal
hroliv@dei.uc.pt
 https://orcid.org/0000-0002-5779-8645

Paulo Gomes
CISUC, University of Coimbra, Portugal
pgomes@dei.uc.pt
 https://orcid.org/0000-0002-4122-9018

Abstract

Although there are tools for some the most common natural language processing tasks in Portuguese, there is a lack of available cross-platform tools specifically targeted for Portuguese, from end to end, namely for integration in projects developed in Java. To address this issue, we have developed and tweaked, over the last half-dozen years, NLPPort, a set of tools that can be used in a pipelined fashion, which we have made publicly available. In this paper, we present the major features of such set of tools.

1 Introduction

Many high-level natural language processing (NLP) tasks rely heavily on some kind of language-specific pre-processing. Texts must be split into sentences and sentences into tokens (words and punctuation), and words often have to be further analysed. For instance, when searching for specific words, as happens in information retrieval (IR), their inflections must be considered to broaden the results. Likewise, for information extraction (IE), tools are also required for processing text and classifying its contents. Each of these tasks must be addressed, mostly in a sequential way, with the output of one tool serving as the input of the next, keeping them modular and easy to adapt and maintain.

Around 2010, in the early development stages of a question-answering system [10], in Java, we felt the need for such kind of tools for Portuguese. Although at the time there were not so many tools as those currently available, some did exist, but with several limitations, specifically when it came to language-specific knowledge. We originally relied on OpenNLP, but soon noticed that some of the tools underperformed when specific constructs of the Portuguese language were not addressed, as is the case of contractions and clitics, that conceal part of their constituents. Also missing were some essentials, like the lemmatizer. Since then, we have been developing NLPPort, with OpenNLP being used as a starting

point for some of the tools – however with new models trained and other tweaks added, some of which based on the manual compilation of language-specific knowledge –, alongside MaltParser, and including other tools developed from scratch.

After a brief overview on related work, the next section present the tools of the NLPPort suite, where key features and settings are described, together with some examples.

2   Related Work

Though some include other tasks, NLP suites of tools typically address: tokenization, part-of-speech (POS) tagging, chunking, and named entity recognition (NER) or classification. Well-established open suites, include Apache OpenNLP,[1] Stanford CoreNLP [7],[2] NLTK [6],[3] FreeLing [9],[4] spaCy,[5] and LinguaKit [3].[6] Most of them have a pipeline including the most common NLP tasks. Still, although most of these suites support or can be trained for many languages, available models do not fully support Portuguese, missing some essential tools (for instance, OpenNLP and NLTK don't include a lemmatizer for Portuguese), or are limited when it comes to using language-specific knowledge, that is not always available or is harder to compile without human intervention.

As for the application programming interfaces (API), LinguaKit uses Perl, NLTK and spaCy use Python, FreeLing uses `C++`, and OpenNLP uses Java. Moreover, LinguaKit and spaCy are recent – at least in the sense that when our work started, they were not available. All things considered, even though it could be possible to develop an interface between APIs in different programming languages, there was still a need for a suite of NLP tools in Java for a more streamlined integration process.

Besides the tools commonly expected in a NLP pipeline, a tool that is increasingly important is a fact extractor. Although there are some reference tools, mainly for English (e.g., ReVerb, OLLIE, ExtrHech, ClausIE) [2], for Portuguese, the scenario is still mostly barren, with the exception of LinguaKit.

That led to a decision to develop a set of tools in Java, although borrowing from already proven tools whenever possible, as is the case of the OpenNLP suite, and the MaltParser[7] dependency parser.

3   Tools

NLPPort includes a set of tools for Portuguese NLP that are either tweaked versions of OpenNLP's tools and MaltParser (including the creation of new models and compilation of rules) or tools entirely developed by us. These go from a sentence splitter to a fact extractor and can be used in a pipeline. In Figure 1, we can observe an overview of the pipeline of tools and how they interact.

---

[1]   *Apache OpenNLP*: `http://opennlp.apache.org/` [Accessed: April 2018].
[2]   *Stanford CoreNLP*: `http://stanfordnlp.github.io/` [Accessed: April 2018].
[3]   *NLTK*: `http://www.nltk.org/` [Accessed: April 2018].
[4]   *FreeLing*: `http://nlp.lsi.upc.edu/` [Accessed: April 2018].
[5]   *spaCy*: `http://www.spacy.io/` [Accessed: April 2018].
[6]   *LinguaKit*: `http://linguakit.com/` [Accessed: April 2018].
[7]   *MaltParser*: `http://www.maltparser.org/` [Accessed: April 2018].
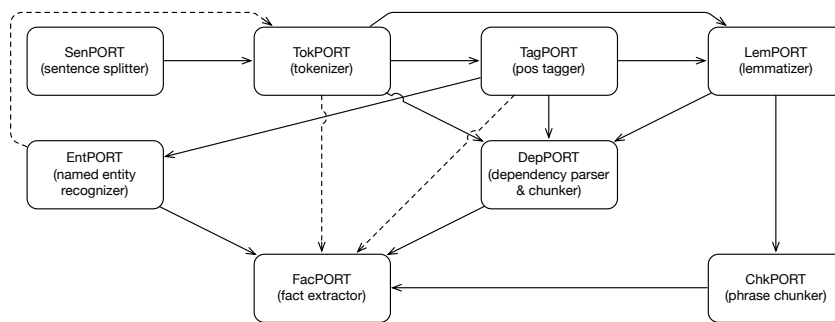
Figure 1 NLPPORT's Pipeline Overview.

```
<replacement target="q.b."></replacement>
<replacement target="q.e.d."></replacement>
<replacement target="q.g."></replacement>
```

Figure 2 Examples of abbreviations used in the sentence splitter.

3.1 SenPORT

SENPORT splits text into sentences, enabling their individual processing by other tools. It is based on OpenNLP's sentence detector, the `SentenceDetectorME` class,[8] with two major tweaks on its input and output:

- A list of abbreviations is used for avoiding splitting sentences on periods that commonly occur in abbreviations. It is applied after sentence splitting: whenever a sentence ends with an abbreviation, it is reattached to the following sentence, if any. Figure 2[9] shows a few examples from the compiled abbreviation list.
- An option for line breaks always resulting in a new sentence – arguably true for many corpora. It is performed before applying OpenNLP's sentence detector and is addressed with a regular expression for the purpose: `(\n\r?)|(\r\n?)`.

For sentence splitting, OpenNLP's sentence detection model available for Portuguese (`pt-sent.bin`) is used[10]. This model was trained on *CoNLL-X Bosque 8.0* [1] data.

3.2 TokPORT

TOKPORT is our tokenizer, which relies on OpenNLP's `TokenizerME` class, the pre-trained model for Portuguese (`pt-token.bin`), but includes some tweaks for considering contractions and clitics when pre-processing the sentences, and for considering abbreviations in post-processing. More precisely, after sentences are tokenized, tokens are optionally checked for the presence of contractions and clitics, in order to better address POS tagging later, as expanding clitics in tokens, clearly separating the verb from the personal pronouns, makes it easier to identify the latter. For example, using one of the rules in Figure 3, we get the form

---

[8] The `ME` suffix in some of OpenNLP's classes denotes the use of a maximum entropy model.

[9] In this figure and in others that follow, excerpts of an XML based format are shown, where all "rules" are defined by means of a *target*, used to find matches in text, and of an eventual *replacement*, that, depending on the task at hand, may be optional.

[10] This and other OpenNLP's pre-trained models used in this suite of tools can be downloaded from `http://opennlp.sourceforge.net/models-1.5/` [Accessed: April 2018].

```
<replacement target="-me-emos">emos a mim</replacement>
<replacement target="-me-ia">ia a mim</replacement>
<replacement target="-me"> a mim</replacement>
```

■ **Figure 3** Examples of clitics processed by the tokenizer.

```
<replacement target="à">a a</replacement>
<replacement target="ao">a o</replacement>
<replacement target="aos">a os</replacement>
```

■ **Figure 4** Examples of contractions processed by the tokenizer.

```
<replacement target="em abono de"></replacement>
<replacement target="em bloco"></replacement>
<replacement target="em breve"></replacement>
```

■ **Figure 5** Examples of token groups used in the tokenizer.

dar-me-ia – in English, *"[it] would give me"* –, that would be POS tagged just as a verb, to *daria a mim* (even if not strictly in line with the structure of Portuguese), yielding as tags a verb, a preposition and a pronoun, respectively for the three resulting tokens. The use of one option over the other may also have implications in the classifications of the other tokens by the POS tagger, by changing the entropy.

The reason for processing contractions is similar to that of clitics. In this case, prepositions and pronouns are broken apart, as shown in Figure 4. For example, *aos* (preposition) is changed into *a os* (preposition and pronoun).

Again, the abbreviation list is used for coupling the period with the respective abbreviation (and classified together) instead of being addressed as punctuation and leading to incorrect POS tags. For abbreviation examples, please refer back to Figure 2. Abbreviations with multiple periods that may have been split by the tokenizer are also put back together (e.g., *q. b.* back to *q.b.*).

We have also opted for grouping tokens during the tokenization process: proper nouns are combined in an "unbreakable" token, to be processed together (feeding back the resulting entities from NER to the tokenizer); and adverbial expressions have their elements grouped together, with some examples presented in Figure 5.

3.3 EntPORT

The named entity (NE) recognizer, ENTPORT, is based on OpenNLP's `NameFinderME` class, and is used straight out-of-the box. Yet, as there was no pre-trained model for Portuguese available among the OpenNLP models, one had to be trained, for which we used the *Floresta Virgem* [1] treebank in the format *árvores deitadas*. Entities are thus classified as one of the following: *abstract*, *artprod* (article or product), *event*, *numeric*, *organization*, *person*, *place*, *thing*, or *time*. The trained model achieves a precision of 81.9%, a recall of 76.%8 and an *F*-measure of 79.3% over the same treebank.

As stated before, the entities recognized by this tool are fed back to TOKPORT in order to bundle together the tokens that compose the entities, so that they can be identified and processed as such. This way, when the tokens of a multiword NE get to the POS tagger, they

get tagged together – for instance, as a proper noun in the case of the name of a person (e.g., {*José da Silva*} for {*José*} {*da*} {*Silva*}) – instead of being individually tagged, with benefits in the POS tagging process itself and later in other tasks that depend on it.

3.4 TagPORT

Given that the previous processing in TokPORT already addressed most of the issues that could affect the outcome of the tagging process, OpenNLP's POS tagger was also used straight out-of-the-box. Specifically, we use the `POSTaggerME` class with the available model for Portuguese (`pt-pos-maxent.bin`). Only a wrapper was created to ease the integration with the other NLPort tools, yielding TagPORT. For reference, the available POS tags are "*adjectivo*," "*advérbio*," "*artigo*," "*nome*," "*numeral*," "*nome próprio*," "*preposição*" and "*verbo*" – and, if considered as such, "*pontuação*"[11].

3.5 LemPORT

For lemmatization, we have developed LemPORT, extensively described elsewhere [11]. Briefly, it allows for *manner*, *number*, *superlative*, *augmentative*, *diminutive*, *gender* and *verb* normalization of words, using two complementary approaches: a lexicon and rules. Among other data, the lexicon contains word inflections, their dictionary form and respective morphological classification, and is used first for attempting normalization of a given word. When that is not possible, the rules, including transformations and classes to which the transformations should be applied to, are introduced iteratively until a match in the lexicon is found or the rules are exhausted. LemPort currently achieves an accuracy over 98% against *Bosque 8.0*.

3.6 ChkPORT

ChkPORT uses OpenNLP's phrase chunker, the class `ChunkerME`, out-of-the-box. Yet, as it happens for NER, no prebuilt model for Portuguese was available. Once more, we have used *Bosque 8.0*, both for training and testing the model, yielding an accuracy of 95%, recall of 96%, and *F*-measure of 95%. The inputs of ChkPORT are the tokens, their POS tags, and the lemmas. Chunks can be classified as nominal (`NP`), verbal (`VP`), prepositional (`PP`), adjectival (`ADJP`) or adverbial (`ADVP`) phrases. Again, except for minor aspects related to the presentation of results (e.g., including the use of the lemmas in the description of the chunks), addressed by a wrapper, the results are used directly in the pipeline.

3.7 DepPORT

For dependency parsing, we have resorted to MaltParser as the basis of both our dependency parser and, inherently, *dependency chunker*, bundled together as DepPORT. Instead of using just the dependencies *per se*, they are used for aggregating tokens from a sentence in chunks. For example, we want to group all the tokens related to the noun identified as the subject of the sentence, rather than just get the noun itself.

The model for MaltParser was also trained with *Bosque 8.0*, after conversion to the CoNLL-X format. Resulting from the application of that model, a token can be assigned

---

[11] In English: *adjective*, *adverb*, *article*, *noun*, *numeral*, *proper noun*, *preposition*, *verb*, and *punctuation*, respectively.

```
[tokens]
id          form        lemma       pos         head        dependency
1           Mel_Blanc   mel_blanc   prop        21          SUBJ
2           era         ser         v-fin       0           ROOT
3           alérgico    adj         adj         21          SC
4           a           a           prp         22          A<
5           cenouras    cenoura     n           23          P<
6           .           .           punc        21          PUNC

↦


[chunks]
id          head        function    tokens
 1          2           SUBJ        [ Mel_Blanc ]
 2          0           ROOT        [ era ]
 3          2           SC          [ alérgico a cenouras ]
```

Figure 6 Dependency parsing and chunking example.

one of many grammatical functions. Of those, only the following can be selected as direct dependents of the *root* token in the next processing step (except for the *root* token itself and *punctuation* tokens):

- (Predicate) Auxiliary Verb (`PAUX`);
- (Predicate) Main Verb (`PMV`);
- Adjunct Adverbial (`ADVL`);
- Adjunct Predicative (`PRED`);
- Auxiliary Verb (`AUX`);
- Complementizer Dependent (`>S`);
- Dative Object (`DAT`);
- Direct Object (`ACC`);
- Focus Marker (`FOC`);
- Main Verb (`MV`);
- Object Complement (`OC`);
- Object Related Arg. Adverbial (`ADVO`);
- Passive Adjunct (`PASS`);
- Predicator (`P`);
- Prepositional Object (`PIV`);
- Punctuation (`PUNC`);
- Root (`ROOT`);
- Statement Predicative (`S<`);
- Subject (`SUBJ`);
- Subject Complement (`SC`);
- Subject Related Arg. Adverbial (`ADVS`);
- Top Node Noun Phrase (`NPHR`);
- Topic Constituent (`TOP`);
- Vocative Adjunct (`VOC`).

Once a sentence is processed by the dependency parser, tokens are grouped in what we call *dependency chunks*. These chunks are formed by selecting the tokens whose head is the *root* of the sentence, and then by aggregating each of those tokens together with all of their dependents. Please refer to Figure 6 for an example of *dependency chunking*. In the same figure, we can observe the results of the tokenizer, the lemmatizer and the POS tagger, alongside dependency parsing and chunking.

3.8 FacPORT

For fact extraction, FACPORT uses NEs and phrase or dependency chunks, combined with a set of rules. Facts are, for all purposes, triples with extra information or metadata, which are composed of:

- an **identifier** – a unique identifier of a fact in relation to each sentence, auto-incremented;
- a **subject** – the subject of the fact, usually a NE or some thing related to the object;
- a **predicate** – the predicate of the fact, typically a verb;

```
<replacement target="[NP][NP]">is a</replacement>
<replacement target="[NP][em:PP]">is in</replacement>
<replacement target="[NP][de:PP]">is part of</replacement>
```

Figure 7 Examples of rules for extracting facts of adjacent phrase chunks.

- an **object** – the object of the fact, usually something related to the subject or a NE, reverse mirroring the contents of the subject;
- a **sentence identifier** – a unique identifier of a sentence in relation to each document, auto-incremented;
- a **document identifier** – a unique identifier for each document (e.g., its filename).

Rules for fact extraction from sentences are only used in the case of adjacent *phrase chunks*. For *dependency chunks*, we use directly the dependency classification of the chunks, as in SUBJ + ROOT + OBJ → subject + predicate + object[12]. Figure 7 shows some rules for extracting facts from adjacent phrase chunks.

When using phrase chunks, the fact extractor checks them for the presence of NEs. When a match occurs, adjacency relations between chunks are used to extract facts. For instance, if a NP chunk contains a person NE and is immediately followed by another NP chunk, it is highly probable that the second chunk is a definition or specification of the first, thus yielding the fact NP_n *is a* NP_{n+1}. The rules specify the classification of the adjacent chunks and also some elements that the chunks may or must contain, appended using a prefix or a suffix with a colon (:).

For dependency chunks, the *subject* and *object* chunks are checked for the presence of entities, and a triple is built using the corresponding predicates and objects or subjects, accordingly. Subject, predicate, and object chunks are then transposed into the corresponding fields of a newly created fact.

Fact extraction is currently limited to the presence of NEs (or proper nouns, when no NEs are available or recognized as such) in both the phrase and dependency chunks. It does not have to be like that, but it is a form of reducing the extraction of spurious facts. We do intend to revise this option, devising a way of selecting facts that may be meaningful even without NEs or proper nouns in them.

In Figure 8, we present examples of facts extracted from a sentence, including an erroneous fact identified with an asterisk. Nevertheless, it can be easily acknowledged that facts do summarize the key information bits in the sentence.

4   Conclusions and Future Work

We have presented NLPPORT, a set of tools that provide cross-platform (using Java) end-to-end Portuguese NLP. It borrows from OpenNLP and MaltParser, but includes multiple tweaks and compiled knowledge that allow for improving the output of its tools, considering constructs such as clitics and contractions.

The tools can be used individually or in a pipeline, where the output of one is the input of the next. A major feature is their modularity, allowing for changes and improvements individually in each of them (eventually improving the overall results of the pipeline). As a

---

[12] Actually, objects (OBJ) are defined, for instance, as dative objects (DAT) or direct objects (ACC), or are not objects at all, as is the case of subject complements (SC), but the structure maintains.

```
Mel Blanc, o homem que deu a sua voz a o coelho mais famoso de o mundo, Bugs
Bunny, era alérgico a cenouras.

↦

Fact: {subject=[Bugs_Bunny], predicate=[ser], object=[a sua voz a o coelho
mais famoso de o mundo]}*
+
Fact: {subject=[Mel_Blanc o homem que deu a sua voz a o coelho mais famoso de
o mundo Bugs_Bunny], predicate=[ser], object=[alérgico a cenouras]}
+
Fact: {subject=[Mel_Blanc], predicate=[ser], object=[o homem que deu a sua voz
a o coelho mais famoso de o mundo Bugs_Bunny]}
+
Fact: {[subject=[Mel_Blanc], predicate=[ser], object=[o homem]}
+
Fact: {[subject=[Mel_Blanc], predicate=[ser], object=[alérgico a cenouras]}
```

Figure 8 Example of facts extracted from a sentence.

whole or individually, some of these tools have been used in the past both inside our research group (e.g., [4, 10, 5, 8]) and outside (e.g., [12]).

Although some of the tools have a only narrow margin for improvement, namely the dependent from external ones, others can greatly benefit from increased interest and use, as is the case of FACPORT, which remains to be properly evaluated against existing solutions.

Finally, a missing tool that is being considered for future development and integration in the suite is one for coreference resolution, which could be easily exploited by FACPORT.

The NLPPORT suite of tools is freely available for download, as a whole or each of the tools individually, at GitHub[13].

References

1   Cláudia Freitas, Paulo Rocha, and Eckhard Bick. Floresta Sintá(c)tica: Bigger, thicker and easier. In *8th Conference on Computational Processing of the Portuguese Language (PROPOR)*, pages 216–219, 2008.

2   Pablo Gamallo. An overview of open information extraction. In *3rd Symposium on Languages, Applications and Technologies (SLATE)*, pages 13–16, 2014.

3   Pablo Gamallo and Marcos Garcia. LinguaKit: Uma ferramenta multilingue para a análise linguística e a extracção de informação. *Linguamática*, 9(1):19–28, 2017.

4   Hugo Gonçalo Oliveira. *Onto.PT: Towards the Automatic Construction of a Lexical Ontology for Portuguese.* PhD thesis, University of Coimbra, 2013.

5   Hugo Gonçalo Oliveira, Diogo Costa, and Alexandre Pinto. Automatic generation of Internet Memes from Portuguese news headlines. In *12th Conference on Computational Processing of the Portuguese Language (PROPOR)*, volume 9727, pages 340–346, 2016.

6   Edward Loper and Steven Bird. NLTK: the natural language toolkit. In *Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics (ETMTNLP)*, pages 63–70, 2002.

7   Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *52nd Annual Meeting of the Association for Computational Linguistics*, pages 55–60, 2014.

---

[13] http://github.com/rikarudo/

8   Ana Oliveira Alves, Ricardo Rodrigues, and Hugo Gonçalo Oliveira. ASAPP: Alinhamento
        semântico automático de palavras aplicado ao Português. *Linguamática*, 8(2):43–58, 2016.

9   Lluís Padró, Miquel Collado, Samuel Reese, Marina Lloberes, and Irene Castellón. FreeLing
        2.1: five years of open-source language processing tools. In *7th Language Resources and
        Evaluation Conference (LREC)*, pages 931–936, 2010.

10   Ricardo Rodrigues. *RAPPort: A Fact-Based Question Answering System for Portuguese.*
        PhD thesis, University of Coimbra, 2017.

11   Ricardo Rodrigues, Hugo Gonçalo Oliveira, and Paulo Gomes. LemPORT: a high-accuracy
        cross-platform lemmatizer for Portuguese. In *3rd Symposium on Languages, Applications
        and Technologies (SLATE)*, pages 267–274, 2014.

12   Derry Tanti Wijaya and Tom Mitchell. Mapping verbs in different languages to knowledge
        base relations using web text as interlingua. In *15th Annual Conference of the North
        American Chapter of the Association for Computational Linguistics (NAACL)*, pages 818–
        827, 2016.

Predicting Performance Problems Through Emotional Analysis

Ricardo Martins

Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
ricardo.martins@algoritmi.uminho.pt
https://orcid.org/0000-0003-1993-5343

José João Almeida

Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
jj@di.uminho.pt
https://orcid.org/0000-0002-0722-2031

Pedro Henriques

Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
prh@di.uminho.pt
https://orcid.org/0000-0002-3208-0207

Paulo Novais

Centro Algoritmi / Departamento de Informática
Universidade do Minho, Campus de Gualtar, Braga, Portugal
pjon@di.uminho.pt
https://orcid.org/0000-0002-3549-0754

Abstract

In the cartoons, every time a character is nervous he/she begins to count to ten to keep calm. This is a technique, among hundreds, that helps to control the emotional state. However, what would be the impact if the emotions would not be controlled? Are the emotions important in terms of impairing the ability to perform tasks correctly?

Using a case study of typing text, this paper is about a process to predict the number of writing errors from a person based on the emotional state and some characteristics of the writing process. Using preprocessing techniques, lexicon-based approaches and machine learning, we achieved a percentage of 80% of correct values, when considering the emotional profile on the writing style.

1   Introduction

In any kind of relationship, a golden rule to avoid problems is not taking decisions under emotional pressure. There are several strategies to do this: from counting to ten before responding an unpolished message until taking a break to "refresh the mind" before the decision.

However, there are situations, such as tests, where it is impossible to avoid emotional pressure and its consequences. When in a stressful situation, or under strong emotional conditions, people tend to make mistakes more frequently. This situation happens in any profession, and so being able to predict these errors that are consequences of emotional states is an important approach to plan a strategy to decrease or avoid them. For example, how important would it be for transportation companies to know the drivers' emotional state before travelling, aiming to reduce the risk of accidents? How important would be for a hospital to predict the errors of a doctor, based on his emotions?

Errors differ from profession to profession and also the effect of emotions over the work is different. Different data sets must be collected to identify these errors, and to correlate them with the emotional state of the worker.

As writers usually express their emotions in the texts they produce through the bag of words they use in each situations, and the typing errors they do along an editing session ca be measured, we intend to model the relation between emotions and errors, using the computer as a case of study. The purpose of the study here reported is to analyse a big collection of texts annotated with editing data to demonstrate that the relation between errors and emotions can be identified, and quantified in order to predict undesired situations.

In this paper, we present an approach using Sentiment Analysis and Machine Learning to characterize the impact of emotions in the number of errors during a typing process. After training the model, it will be used to predict new cases in order to assess it.

It is not our intention to claim that this approach is an alternative for predicting errors in all situations, however, we think that the approach will lead further future investigation in this relevant topic.

The remainder of the paper is as follows: Section 2 introduces the concept of basic emotions, which is a well-known theory used in Sentiment Analysis. Section 3 discusses related work concerned with the detection of emotions from a text that inspired this work, while Section 4 describes the creation of a dataset for our tests. Section 5 presents our proposal, explains the steps used in the analysis, discusses the results obtained from a set of tests performed. The paper ends in Section 6 with the conclusions and future work.

2   Theory of Basic Emotions

Basic emotion theorists explain that every human emotion is composed of a set of discrete basic emotions [2, 4, 11].

Many researchers have identified some basic universal emotions. One of the first attempts is a study by Paul Ekman [3] which concluded that there are six basic emotions are *Dislike*, *Happiness*, *Sadness*, *Anger*, *Fear* and *Surprise*. His work is based on the theory that human faces can represent this basic emotion as universal pictures.

For Plutchik [11], every sentiment is composed of a set of 8 basic emotions: *Anger*, *Anticipation*, *Disgust*, *Fear*, *Joy*, *Sadness*, *Surprise* and *Trust*, represented as a "wheel of emotions". Furthermore, the combination of basic emotions results in *dyads*. Plutchik created rules for building the *dyads*, defining the primary dyads emotions as the sum of two adjacent basic emotions, as *Optimism = Anticipation + Joy*. Meanwhile, secondary dyads

emotions are composed of emotions that are one step apart on the "emotion wheel", as *Unbelief = Surprise + Disgust.* The tertiary emotions are generated from emotions that are two steps apart on the wheel, as *Outrage = Surprise + Anger.*

Other well-known emotions model is the Five Factor Model (as known as Big Five), introduced by McCrae [8] which suggests that the personality is composed of 5 independent factors:

Openness to experience: People with high scores like news and tend to be creative. At the other end of the scale are the conventional and orderly, those who like the routine and have a keen sense of right and wrong;

Conscientiousness: It measures the level of concentration. Those with high scores are highly motivated, disciplined, committed and trustworthy. Those with low results are undisciplined and easily distracted;

Extroversion: It measures the sense of well-being, the level of energy, and the ability in interpersonal relationships. High scores mean affability, sociability, and ability to impose oneself. Lows indicate introversion, reservation, and submission;

Agreeableness: It refers to how we relate to others. Many points indicate a compassionate, friendly and warm person. At the other end are the withdrawn, critical and egocentric;

Neuroticism: It measures emotional instability. People with high scores on this scale are anxious, inhibited, melancholic and have low self-esteem. Those that get low scores are easy to deal with, optimistic and well-liked with themselves.

In this paper, we adopt the Plutchik's model to represent emotions because we consider more realistic, easy to use and this model allows to represent several different emotions through dyads emotion. Moreover, there are some libraries and lexicons used in this work which represent and process the emotions according to this model.

3 Related work

There are several works using sentiment analysis for diversified objectives, and some are more relevant for the present paper as they have been used as a source for the idea proposed. In this section, we survey these inspiring works. However, predicting typing errors from an emotional analysis is an unexplored field and we have not found specific previous works to reference.

The usage of emotional labels for predictions was inspired by the work of Martins et al. [7], which uses emotional labels to improve the authorship identification. This is made using Facebook posts from personalities known and a hybrid approach containing lexicon and machine learning approaches.

Moreover, Thewall et al. [12] have presented a work to extract sentiment strength from the informal English text, using new methods to exploit the de facto grammars and spelling styles of cyberspace, which contributed with the idea of extract sentiment polarities from text.

Finally, the work presented by Meier et al. [9] contributed to the idea of predicting writing performance using affective variables to relate to efficacy expectations.

4 Data creation

In order to analyse the impact of the emotions during the text writing process, it was necessary to analyse texts containing emotional load and meta-information about its creation. For this purpose, the dataset provided by Banerjee et al [1] containing keystroke logs for
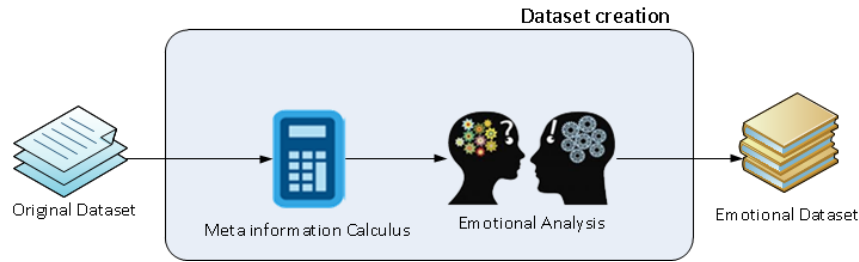
Figure 1 Dataset creation process.

opinion texts about gun control[1] was used as the basis for a new dataset creation. To perform the intended analysis we actually needed a text repository with emotional load and editing numerical data for each written piece. The option of a keystroke log is justified by the necessity of gather information about the text creation process. So, in our study all texts are considered written "from beginning to end", i.e., the first typing step without a posterior text revision phase. This is important for levelling possible errors and editing in a same identifiable pattern.

The process of the new datasets creation is performed in 2 steps: Meta-information Creation and Emotional Analysis, as presented in Figure 1.

4.1 Meta-information

The Meta-information Creation step analyses the keystroke log and calculates metrics about the text creation. For analysis purposes, we defined some metrics considered important in this study. These metrics are:

- **TimeText:** It represents the time spent during the text writing. It is the amount of time span in milliseconds between press and release for each character and white space key in the keyboard. Punctuations, numbers, and others are discarded;
- **AveragePerWord:** Is the average between the total words in the text and the time spent during typing process;
- **AmountErrors:** It is the number of errors during the typing process. It is important to emphasize that due to dataset limitations that do not store mouse movements or selections, it is impossible to detect all forms of removing characters (for example, single character or block removing). For convenience of this study, it is considered an *error* each *backspace occurrence*[2];
- **AverageErrors:** It is the average of the total words in the text and the number of errors during typing process;
- **TimeBetweenKeys:** It is the average time span between the keyboard press;
- **RepeatedCharacterFrequence:** It is the frequency of a character is repeated in the text. A repeated character is considered the same character those have been pressed immediately before and the time between them is at least 15% lower than TimeBetweenKeys. This is important to detect situations when a key is pressed for a long time, repeating the character.

---

[1] This is a hot, sensible, topic provoking emotive reactions on commenters.
[2] We are aware that counting backspaces is not the more adequate way to count errors, because other reasons can lead the writer to backspace and delete characters, and also many errors are made without being detected and corrected. Anyway we feel that there is a clear relation between both.
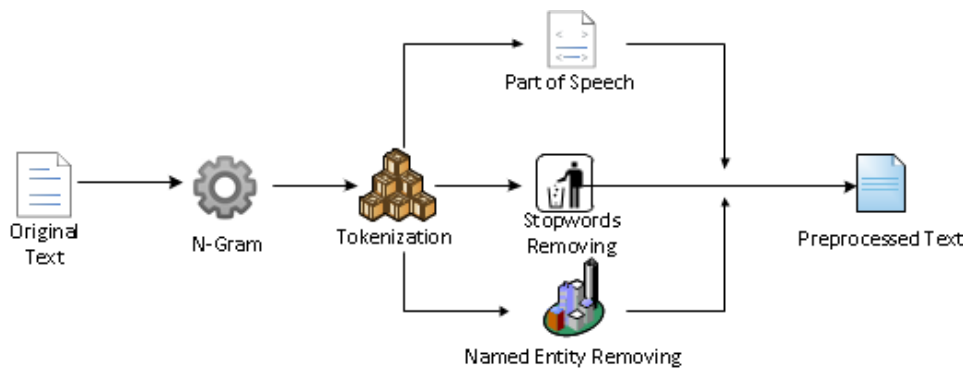
Figure 2 Preprocessing pipeline.

4.2 Emotional Analysis

The Emotional Analysis step is responsible for identification of each basic emotion according to Plutchik's model [11]. This model was chosen because there are many libraries to process information according to it and lexicons which contain the basic emotions for each word. To achieve this objective, all sentences are analysed using the EmoLex [10] lexicon.

For this analysis, all texts have been submitted to a preprocessing pipeline; at the end of this phase, only the relevant information remained.

This pipeline was composed of n-Gram identification, tokenization, stopwords removal, part of speech tagging and named entity removal, as presented in Figure 2.

Using the Stanford Core NLP toolkit [6] for these tasks, the preprocessing is divided into 3 parallel tasks. This is important because both Part of Speech Tagging and Named Entity Recognition need the text in the original format in order to identify the information.

The preprocessing begins with the N-Gram identification, where a predefined set of n-grams are identified in the text and labelled to be interpreted as a single word. Later, the tokenizer splits the text in a list of words (tokens) and these tokens are syntactically analysed in Part of Speech, where the nouns, verbs, adverbs, and adjectives are identified and stored for future purposes. In parallel, the tokens identified in a predefined stopwords list are removed and the tokens in named entity process are analysed in order to identify names (persons, locations or organizations) and discard them.

Later, the common tokens in these processes are stored and the emotions from each preprocessed text are identified through a process in R which queries the NRC lexicon [10] and identifies the basic emotions.

Finally, all information is stored in a new dataset containing the opinion and preprocessed text (from the original dataset), the 6 metrics created in Meta-Information Creation step, 8 basic emotions percentages and 2 polarities identified in the Emotional Analysis step.

5 Data analysis

The objective of this analysis is to find some evidence that emotions influence the writing process. In order to achieve this objective, some experiments were performed to relate emotions and writing patterns.

■ **Table 1** Correlations between *AmountErrors* and emotions.

Emotion	r^2	Emotion	r^2	Emotion	r^2	Emotion	r^2
Anger	0.35	Optimism	0.30	Pessimism	0.42	Trust	0.36
Anticipation	0.31	Hope	0.39	Awe	0.38	Curiosity	0.37
Disgust	0.26	Anxiety	0.52	Despair	0.38	Pride	0.38
Fear	0.38	Love	0.35	Shame	0.38	Surprise	0.19
Joy	0.23	Guilt	0.41	Disapproval	0.30	Remorse	0.31
Sadness	0.30	Delight	0.25	Unbelief	0.27	Outrage	0.34

5.1 Emotional correlations

As initial step, the values for some Plutchik's basic emotions and defined dyad emotions[11] were calculated in order to provide more sources of information to analyse the data. To calculate these emotions, we used the package Syuzhet [5] in R, which analyses the text provided and returns the values of each basic emotion contained into the text, according to the NRC lexicon [10]. The dyad emotions were calculated according to the formula below:

- Optimism = Anticipation + Joy;
- Disapproval = Surprise + Sadness;
- Hope = Anticipation + Trust;
- Unbelief = Surprise + Disgust;
- Anxiety = Anticipation + Fear;
- Outrage = Surprise + Anger;
- Love = Joy + Trust;
- Remorse = Sadness + Disgust;

- Guilt= Joy + Fear;
- Delight = Joy + Surprise;
- Pessimism = Sadness + Anticipation;
- Curiosity = Trust + Surprise;
- Awe = Fear + Surprise;
- Despair = Fear + Sadness;
- Pride = Anger + Joy;
- Shame = Fear + Disgust;

Later, the Pearson correlation (r^2) was applied to both each basic emotion and dyad emotions, to obtaining the correlation between the number of backspaces in the writing process[3] (*AmountErrors* as presented in subsection 4.1) and the emotions, according to Table 1.

Despite no single emotion having a strong correlation, it is possible to identify that among all emotions analysed, anxiety is the most relevant for the number of errors during typing, having a moderate correlation.

5.2 Machine learning predictions

A machine learning analysis was applied to determine the influence of the meta-information and emotional labels on the number of errors prediction. For this purpose, 5 different scenarios were considered:

- Scenario A: Only text and opinion – no meta-information neither emotional labels;
- Scenario B: Only meta-information and emotional labels;
- Scenario C: Only meta-information;
- Scenario D: Only emotional labels;
- Scenario E: All dataset information – text, opinion, meta-information and emotional labels.

---

[3] Remember that we are using this measure to compute the errors number.

■ **Table 2** Algorithms correlations and their Root Mean Squared Error (RMSE).

Scenario	Linear Regression	RMSE	SVM	RMSE	Random Forest	RMSE	Decision Table	RMSE
Scenario A	0.171	391.28	0.347	208.62	0.429	145.36	0.321	104.78
Scenario B	0.774	99.62	0.769	103.32	0.791	96.67	0.739	106.37
Scenario C	0.777	99.14	0.770	103.08	0.780	98.66	0.739	106.37
Scenario D	0.525	134.10	0.528	137.73	0.492	141.70	0.426	143.54
Scenario E	0.320	437.12	0.696	131.79	0.586	127.88	0.591	129.18

■ **Table 3** Ranges of *AmountErrors* per emotions.

AmountErrors	Emotions							
	Anger	Anticipation	Disgust	Fear	Joy	Sadness	Surprise	Trust
0.0-204.5	2.5-3.5	0.5-1.5	0.5-1.5	0.0-4.5	0.0-0.5	0.0-1.5	0.0-0.5	2.5-4.5
204.5-256.5	3.5-5.5	1.5-2.5	0.0-0.5	5.5-7.5	1.5-2.5	2.5-3.5	0.5-1.5	4.5-∞
256.5-340	3.5-5.5	0.5-1.5	1.5-2.5	5.5-7.5	0.5-1.5	3.5-∞	0.0-1	2.5-4.5
340-∞	5.5-∞	2.5-∞	2.5-∞	7.5-∞	2.5-∞	3.5-∞	2.5-∞	4.5-∞

The dataset used for both training and validation is the same created in subsection 4.2. In this analysis, the meta-information *AverageErrors* was discarded because it has strong correlation with *AmountErrors*, induced by *AmountErrors ≈ AverageErrors ∗ TimeText*.

For each scenario, the following machine learning algorithms were applied: Linear Regression, SVM, Random Forest and Decision Table.

All tests were performed using a 10-fold cross-validation in Weka; the correlation coefficients obtained with each algorithm between AmountErrors and the dimensions analysed in each scenario are shown in Table 2.

After the tests, the best correlation coefficient for predicting the number of errors was obtained with the Random Forest algorithm for Scenario B (only meta-information and emotional labels).

Once identified that the meta-information and emotional labels are an adequate to predict the number of errors, the next step was to identify the patterns for these predictions. For this purpose, all values were discretized into 4 groups and a K-Means algorithm was used to cluster the data (meta-information and emotional labels) into 4 different clusters representing respectively 39%, 17%, 23% and 21% of the information available.

In a preliminary analysis, all meta-information was identified with the same value range, and for this reason, it was considered irrelevant for this objective and removed from the visualization. Also, as the purpose of this analysis is to measure the emotional influence in the *AmountErrors* values, the dimensions *Positive* and *Negative* were removed too. Then the relevant information remaining was grouped by *AmountErrors* and is presented in Table 3, where the range in *AmountErros* refers to the number of errors identified, while the range for each emotion refers to the number of words containing the emotion in the text.

Having in mind the results in Table 3, it is possible to conclude that in general, as higher the emotions are, higher is the impact on the *AmountErrors* number.

6   Conclusion

A typist certainly will have fewer errors than a normal person when typing. However, even this typist will do more mistakes if he is, for instance, anxious or feeling guilty.

As a first step in a research direction we want to further explore – *the sentiment analysis in different tasks to understand the effect of the worker's emotional state on his performance, to reduce mistakes* – this paper presents a combination of lexicon-based and machine learning approaches to correlate the number of typing errors based on the emotional labels and metrics associated with the text creation (text first typing/editing). That model can be used to predict errors based on the information of the emotional state; in this way we will have a rigorous criterion to recommend people to stop doing some task under some personal states to avoid dangerous faults.

Everyone has particular characteristics of expressing himself and these personal characteristics can of course influence that prediction. Maybe on account of that, the study results so far obtained were a bit surprising, and the measured influence of emotions on users tendency to make mistakes is *moderate* (we were expecting bigger values). In our tests, the best approaches for predicting errors based on human behaviour were obtained using emotional information (emotions inferred from the text lexical analysis)and meta-information (metrics evaluated based on the text creation process) collected during text typing. Clustering the data revealed how the emotions can affect the number of errors. It is a promising result.

Despite the final outcomes, this work is the beginning of a research line, and some premises were assumed (like, calculate typing errors by counting the backspace-key strokes) in order to get objective data to produce numerical, trustable, results. However, when advancing this research line, these premises will be revisited and new detection/measuring approaches will be considered.

As future work, it is planned to increase the accuracy of the system to handle with character block removing (i.e. removing several characters at once with a mouse text selection and one backspace press). Moreover, it is planned to detect and handle typos as a different category of errors, as well as to determine on an individual scale, how the emotions affect people in their usual routine, increasing the risks of errors in their activities. Also, it is planned to analyse different texts from different authors. It is important to avoid bias in themes and identify "stressive patterns" which increase the number of errors in different authors and themes.

References

1 Ritwik Banerjee, Song Feng, Jun Seok Kang, and Yejin Choi. Keystroke patterns as prosody in digital writings: A case study with deceptive reviews and essays. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1469–1473, 2014.

2 Paul Ekman. An argument for basic emotions. *Cognition & emotion*, 6(3-4):169–200, 1992.

3 Paul Ekman and Richard J. Davidson. *The nature of emotion: Fundamental questions.* Oxford University Press, 1994.

4 Carroll E. Izard. *Human emotions.* Springer Science & Business Media, 2013.

5 Matthew L. Jockers. *Syuzhet: Extract Sentiment and Plot Arcs from Text*, 2015. URL: https://github.com/mjockers/syuzhet.

6 Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *52nd Annual Meeting of the Association for Computational Linguistics*, pages 55–60, 2014.

7 Ricardo Martins, José João Dias de Almeida, Pedro Rangel Henriques, and Paulo Novais. Increasing authorship identification through emotional analysis. In *Trends and Advances in Information Systems and Technologies*, volume 745, pages 763–772. Springer International Publishing, 2018. `doi:10.1007/978-3-319-77703-0_76`.

8 Robert R. McCrae and Oliver P. John. An introduction to the five-factor model and its applications. *Journal of personality*, 60(2):175–215, 1992.

9 Scott Meier, Patricia R. McCarthy, and Ronald R. Schmeck. Validity of self-efficacy as a predictor of writing performance. *Cognitive therapy and research*, 8(2):107–120, 1984.

10 Saif Mohammad and Peter D. Turney. Crowdsourcing a word-emotion association lexicon. *Computational Intelligence*, 29(3):436–465, 2013. `doi:10.1111/j.1467-8640.2012.00460.x`.

11 Robert Plutchik. *Emotion: A psychoevolutionary synthesis*. Harpercollins College Division, 1980.

12 Mike Thelwall, Kevan Buckley, Georgios Paltoglou, Di Cai, and Arvid Kappas. Sentiment strength detection in short informal text. *Journal of the Association for Information Science and Technology*, 61(12):2544–2558, 2010.