

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Development of an Open-Source Data-to-Text System

Nuno Miguel Teixeira Cardoso



Mestrado em Engenharia Informática e Computação

Supervisor: Sérgio Nunes

October 22, 2022

Development of an Open-Source Data-to-Text System

Nuno Miguel Teixeira Cardoso

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Jorge Barbosa

Referee: Prof. Nuno Escudeiro

October 22, 2022

Abstract

The amount of data produced and stored in databases is continuously increasing, leading to new data presentation opportunities that facilitate its interpretation. Structured data can serve as a basis for creating various informative textual contents, such as summaries, reports, or news articles. These may represent a more convenient way for lay users to obtain and showcase information rather than through complex charts, tables, and query tools. With the help of an automatic writing system, the opportunity to report information based on the structured data and present it in an accessible way can be seized. Template-based natural language generation systems can produce news and summaries using data about different entities and events and editable sentence templates. These generators provide various ways to express the same idea or concept and make it possible to describe the data in multiple languages. Throughout the last years, ZOS has been developing the Prosebot system. Prosebot is a natural language generation system that can generate match summaries, given sports data from ZOS' database and pre-defined language templates, in an efficient and autonomous way. The main contribution of the present project is the conversion of Prosebot into an open-source software model, with corresponding code restructuring and domain generalization. Other contributions include finishing the multilingual support and creating a templates validation algorithm for users writing support. To open the system to a larger audience, in an easy-to-use way, a new web platform with a user-friendly interface was also developed to help manage templates. The idea was to embrace the no-code paradigm so that lay users without programming skills could interact with the system. The evaluation of the final product showed promising results. With the validation algorithm, the number of errors made by the *zerozero.pt*'s newsroom journalists, when writing the templates, decreased considerably. In addition, through some user experience interviews, it was possible to obtain feedback and some prospects of improvement in developing the Templates Management Platform. With the use of the SonarQube platform, a thorough code analysis of the generator showed great results in terms of maintainability, security and reliability, and an improvement compared to the initial state of the project. The successful development and publication of this project's work produced a helpful system for content creators and companies to speed up the process of writing news and summaries through the automatic generation of semi-finalized text versions from structured data.

Keywords: Natural Language Generation, No-Code, Open-Source

Resumo

A quantidade de dados produzidos e armazenados em bases de dados está em contínua ascensão, criando novas oportunidades para apresentação da informação de forma a facilitar a sua interpretação. Dados estruturados podem servir de base à criação de vários conteúdos textuais informativos, tais como sínteses, relatórios ou artigos de notícias. Este tipo de conteúdos pode representar uma forma mais conveniente de utilizadores leigos obterem e partilharem informação, do que através de gráficos e tabelas complexos, ou recorrendo a ferramentas de interrogação. Com a ajuda de um sistema de escrita automática, a oportunidade de relatar informação que reside nos dados estruturados, apresentando-a de uma forma acessível, pode ser aproveitada. Sistemas de geração de linguagem natural baseados em *templates* (modelos de frases) são capazes de produzir notícias e sínteses usando dados sobre diferentes entidades e eventos e *templates* de frase editáveis. Estes geradores permitem exprimir a mesma ideia ou conceito de diferentes formas e fornecem ainda a possibilidade de descrever os dados em múltiplos idiomas. Ao longo dos últimos anos, a ZOS tem vindo a desenvolver o sistema Prosebot. Prosebot é um sistema de geração de linguagem natural capaz de gerar sínteses de jogos, de forma eficiente e autónoma, a partir de dados de desporto armazenados na base de dados da ZOS e de *templates* de linguagem predefinidos. A principal contribuição do presente projeto é a conversão do Prosebot para modelo de software open-source, com a respetiva reestruturação do código e generalização do domínio. Outras contribuições incluem a finalização do suporte multilíngue e a criação de um algoritmo de validação de *templates* como suporte à escrita dos utilizadores. De modo a abrir o sistema a um público mais abrangente, uma nova plataforma *web* de interface apelativa e fácil utilização foi desenvolvida para gerir os *templates*. A ideia passa por abraçar o paradigma *no-code*, permitindo que utilizadores leigos, sem conhecimentos de programação, possam também interagir com o sistema. A avaliação do produto final revelou resultados promissores. Com o algoritmo de validação, o número de erros cometidos pelos jornalistas da redação do *zerozero.pt*, ao escrever os *templates*, diminuiu consideravelmente. Além disso, através de algumas entrevistas de experiência do utilizador, foi possível obter *feedback* e algumas perspetivas de melhoria no desenvolvimento da Plataforma de Gestão de Templates. Com a utilização da plataforma SonarQube, uma análise completa do código do gerador demonstrou bons resultados em termos de capacidade de manutenção, segurança e fiabilidade, e uma melhoria em relação ao estado inicial do projeto. O desenvolvimento e publicação deste projecto deram origem a um sistema particularmente útil a criadores de conteúdos e empresas que pretendem acelerar o processo de redação de notícias e sínteses, recorrendo à geração automática de versões semi-finalizadas de texto a partir de dados estruturados.

Keywords: Geração de Linguagem Natural, *No-Code*, *Open-Source*

Acknowledgements

Initially, I would like to thank my supervisor Sérgio Nunes for all the support and guidance throughout this process. I also want to thank *zerozero.pt*'s workers for the friendly environment they received me with, and the working conditions offered during the past months. In particular, I would like to thank Marco Sousa, Pedro Dias, and Vasco Ribeiro for the technical support and guidance, and Daniel Oliveira, Gonçalo Silva, Paulo Mangerotti, Paulo Freitas, and Steve Ramos for the help given in writing professional-quality content for the system.

Finally, I want to thank my friends and family and give a special thank you to my parents, Lurdes Cardoso and Nuno Cardoso, who always supported me throughout the years and provided me with education and great values.

Nuno Cardoso

“A wise man will make more opportunities than he finds.”

Sir Francis Bacon

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	1
1.3	Objectives	2
1.4	Document Structure	2
2	Overview of NLG Systems	3
2.1	Natural Language Generation	3
2.2	Architecture of an NLG System	3
2.3	Commercial NLG Systems	5
2.4	Open-Source NLG Systems	7
2.5	Literature Review Analysis	30
2.6	No-Code Paradigm	35
3	Prosebot Background	37
3.1	The GameRecapper System	37
3.2	Statistical Language Models	39
3.3	Prosebot	40
3.4	Evaluation Metrics	41
3.5	Community-Based Platform and Post-Editing	42
3.6	Placement on the Typical NLG Architecture	43
4	Prosebot Generator	46
4.1	Multilingual Support and Improvements	46
4.2	Templates Validation Algorithm	51
5	Open-Source Refactoring	62
5.1	The Prosebot System	62
5.2	Code Restructuring	62
5.3	Automatic Dictionaries Generation	68
5.4	API Decouple and Base Content Definition	71
5.5	Code Analysis	72
5.6	Architecture	73
5.7	Publishing Process	75
6	Templates Management Platform	77
6.1	API development	77
6.2	Views	78
6.3	Features	78

7	Evaluation	84
7.1	Methodology	84
7.2	Results	85
8	Conclusions and Future Work	93
8.1	Conclusions	93
8.2	Future Work	94
	References	95
A	Open-Source NLG Solutions	103
B	Complete Class Diagram	105
C	Templates Management Platform UX Interview Guide	106

List of Figures

2.1	NLG systems' architecture and task distribution presented by Reiter and Dale [68]. Adapted from Aires [1].	5
2.2	SURGE input-output example. Image from Elhadad and Robin [27, Fig. 1]. . . .	10
2.3	KPML text generation in English, French, Greek, German, Japanese and Dutch. Image from Bateman [6].	11
2.4	Architecture of TG/2. Image from Busemann [9, Fig. 1].	12
2.5	GenI input screen. Image from Kow [45].	13
2.6	GenI debugger. Image from Kow [45].	14
2.7	ASTROGEN architecture. Image from Dalianis [17].	17
2.8	MUG workbench. Image from Reitter [69].	18
2.9	The RNNLG framework for language generation. Image from Wen and Young [85, Fig. 1].	20
2.10	RosaeNLG demo. Image from Stoecklé [77].	21
2.11	The processing stages and sub-stages of NaturalOWL. Image from Androutsopoulos, Lampouras and Galanis [3, Fig. 1].	22
2.12	Modular architecture of PASS. Adapted from Lee, Krahmer and Wubben [82, Fig. 1].	23
2.13	Accelerated Text Document Plan. Image from Navickas [92].	25
2.14	LKB screen dump. Image from Copestake et al. [13, Fig. 1].	27
2.15	General architecture of NLGen 2. Image from Singh et al. [74, Fig. 4].	29
2.16	Open-source NLG solutions over the years.	32
2.17	Open-source NLG solutions popularity.	33
2.18	Language support of open-source NLG solutions.	34
2.19	Open-source NLG solutions licenses.	35
3.1	Example match information from www.zerozero.pt . Image from Aires [1, Fig. 3.1].	39
3.2	User interface. Image from Soares [75, Fig. 3.3].	40
3.3	Prosebot architecture. Adapted from Ribeiro [71, Fig. 3.2].	41
3.4	View after inserting the match ID and loading the event information. Image from Correia [14, Fig. 6.3].	42
3.5	Match summary generated by Prosebot and published on zerozero.pt . Image from https://www.zerozero.pt/news.php?id=352050 (accessed Feb. 10, 2022).	44
3.6	Prosebot generation process example.	45
4.1	NFA diagram - Regular strings and token declarations.	54
4.2	NFA diagram - Token declaration.	55
4.3	NFA diagram - Connector declaration.	55
4.4	NFA diagram - Condition (and = &&; or = ; var = variable name).	57
4.5	User interface for templates validation	59

5.1	UML components diagram of the Prosebot system.	63
5.2	Directories tree of the core files.	66
5.3	Directories tree of a specific context.	67
5.4	Prosebot generator's UML activity diagram.	74
5.5	Prosebot generator's UML class diagram.	75
6.1	Home page.	79
6.2	File import page.	79
6.3	Template file view.	80
6.4	Delete template file pop-up.	81
6.5	Rename template key.	82
6.6	Template edit view.	82
6.7	Template validation.	83
7.1	Number of detected writing errors over time.	87
7.2	Percentage of writing errors over time relative to the number of logs produced.	88
7.3	SonarQube measures report of the version before the project's work.	88
7.4	SonarQube measures report of the final version before revision.	89
7.5	SonarQube measures report of the final version after revision.	89
B.1	Prosebot generator's complete UML class diagram.	105

List of Tables

2.1	Reference NLG companies.	7
2.2	Other NLG vendors.	8
2.3	Availability of open-source NLG solutions.	9
2.4	Top 10 open-source NLG references.	31
2.5	Top 15 authors, according to Google Scholar h-indexes.	31
2.6	Top 15 authors, according to Scopus h-indexes.	32
2.7	Popularity of open-source NLG solutions through analyses crossing.	34
4.1	Previous match variables.	48
4.2	Italian regular connectors.	49
4.3	Variations of “il”.	50
4.4	Variations of “e”.	50
4.5	Italian composite connectors.	50
4.6	Token declaration patterns.	53
4.7	Connector declaration patterns.	53
4.8	Templates validation algorithm rules.	61
6.1	Prosebot component API.	77
6.2	Prosebot Editor component API.	78
7.1	Templates validation logs fields.	84
7.2	SonarQube labeling scale. Source: SonarSource S.A [73].	85
A.1	Open-source NLG solutions.	104

Listings

2.1	Example Chimera RDF triplets input from Moryossef, Goldberg and Dagan [57].	24
4.1	Example template keys.	52
4.2	Unused template warning.	54
4.3	CFG for <i>text</i> productions.	56
4.4	CFG for <i>condition</i> productions.	57
4.5	Error messages formats.	60
4.6	Warning messages formats.	60
5.1	Example grammar and linguistic functions for the Spanish language, before restructuring.	64
5.2	Example list of connectors for the Spanish language, after restructuring.	65
5.3	Example grammar functions for the Italian language, after restructuring.	65
5.4	<i>CompetitionData</i> 's <i>get_entity</i> method, before restructuring.	69
5.5	<i>EntityGetter</i> classes.	70
5.6	<i>MatchData</i> 's list of tokens, after restructuring.	71
7.1	Testing logs of the validation algorithm integration.	86

Abbreviations and Symbols

AI	Artificial Intelligence
API	Application Programming Interface
CFG	Context-Free Grammar
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
NFA	Non-Deterministic Finite Automaton
NLG	Natural Language Generation
UX	User Experience
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

Natural Language Generation (NLG) is a subarea of Artificial Intelligence (AI) that handles the automatic production of textual content [66]. With increasing interest by companies and customers for speeding up the writing of summaries, explanations, news, and descriptions from structured data stored in their databases, NLG has grown to become a fully established commercial category [16].

ZOS¹ is a Portuguese company that provides sports data support for multiple media channels, from news publications to betting companies, and claims to have the world’s largest football database [48]. Considering the daily number of games and sports events and the large amount of data gathered, a new opportunity arises to generate textual content from ZOS’s structured data. Showcasing information through complex charts and tables can turn it challenging for some people to comprehend. Thus, presenting it in a text form can be the best way to increase user engagement.

Throughout the last years, ZOS has been developing Prosebot, alongside their collaboration with FEUP, as part of the *zerozero.pt* project. Prosebot is a template-based natural language generation application that produces mainly matches’ summaries from sports data extracted from their database in an automatic efficient way. Journalists can use it to generate an initial version of the news content to serve as a basis for their work. Additionally, it is already being used by *zerozero.pt*’s advanced users to automatically generate summaries for football matches that range from amateur to professional leagues, from senior footballers to the young football academies [64].

1.2 Motivation

Despite the continuous advancements in the NLG area and the growing demand by companies for these types of technologies and tools to help produce textual information, the number of open-source systems available for public use is still low. Ehud Reiter [67] mentions that developing open-source software is an arduous and time-consuming task. Converting a company’s internal

¹<https://www.zos.pt/>

tool into open-source requires refactoring, reorganizing, and documenting the code, improving readability and comprehension for future users. Flexibility, robustness to failures, maintenance, and support for questions and problems users may encounter are also essential aspects to consider. Although these may represent stopping factors to development, some companies already pay their employees to develop the open-source software they use. With the conversion of Prosebot into open-source software, ZOS intends to make the system known to a broader audience, promote knowledge sharing and expand its functionality through collaboration with the community.

Moreover, a development approach following the no-code paradigm [91] and including multi-lingual support would create new opportunities for lay users to interact with the system, create and manage templates, and take advantage of the overall NLG features, overcoming the lack of programming skills and knowledge. It could change how these projects are approached by opening the development environment to a larger audience.

1.3 Objectives

The main objective of this work is to adapt the existing version of the Prosebot application into an open-source software project, making it freely available for public use. Following the no-code paradigm, implementing a templates management platform with an intuitive interface is also an objective. Other goals include improving the implementation of multilingual support and developing a validation algorithm for evaluating the code used in each text template.

In the end, programmers should be able to reuse the code, expand the tool's functionality, and integrate it into their systems. At the same time, lay users should be provided with resources to execute simple tasks, manage language templates, and create initial versions of descriptions, news, and summaries.

1.4 Document Structure

This document consists of seven main chapters. Chapter 2 conveys an overview of the state of the art in Natural Language Generation in the commercial and open-source fields, a definition of typical architectures, and some examples of available technologies in the area. Chapter 3 describes previous work developed in Prosebot over the years. Chapter 4 presents the improvements made to the NLG module, the inclusion of new features to expand the multilingual support of Prosebot, and the implementation of a templates validation algorithm. Chapter 5 depicts the development process to adapt the system to open-source software. Then, Chapter 6 describes a user-friendly web application developed to help manage templates, following the no-code paradigm. Chapter 7 presents the evaluation results of the use of the templates validation algorithm, the code quality of the generation module, and the use of the templates management platform. Lastly, Chapter 8 concludes the project description and outlines some future work.

Chapter 2

Overview of NLG Systems

This chapter presents a literature review on Natural Language Generation systems and their typical architecture. Then, it describes the commercial state of the art, from the early days to the current development wave, and enumerates some existing commercial tools. Finally, it presents an in-depth review of open-source NLG systems.

2.1 Natural Language Generation

Reiter and Dale define Natural Language Generation as “*the subfield of artificial intelligence and computational linguistics that is concerned with the construction of computer systems that can produce understandable texts in English or other human languages from some underlying non-linguistic representation of information*” [68, p. 1]. Data-to-text systems generate textual content from structured input data and thus perfectly fit the definition. However, text-to-text systems should not be ignored since they also use NLG to translate and summarize texts [14, 29].

2.2 Architecture of an NLG System

2.2.1 Tasks

According to Reiter and Dale [68], an NLG system should execute several tasks in order to progress from input data to the production of output text. The six main tasks agreed within the NLG community are described below.

Content determination: It is essential to start by “*deciding what information should be communicated in the text*” [68, p. 9]. The idea is to filter the input data to choose the relevant information based on the context of the system. This information creates a set of messages and will then be used on the following tasks to generate the appropriate textual content.

Discourse planning: After the content is determined, begins “*the process of imposing ordering and structure over the set of messages to be conveyed*” [68, p. 10]. Discourse planning splits the information through an organized structure that in the simplest form can be just the division into a beginning, a middle, and an end. However, more sophisticated divisions are often witnessed. The result of this process is usually a tree structure with leafs being individual messages and internal nodes describing the relations and grouping between them.

Sentence aggregation: It is “*the process of grouping messages together into sentences*” [68, p. 10]. Although this may be a pivotal task to improve the fluidity and readability of the speech, some messages may not need aggregation as they are complete enough to make a full sentence.

Lexicalization The task of “*deciding which specific words and phrases should be chosen to express the domain concepts and relations which appear in the messages*” [68, p. 11]. This decision is highly context-dependent. Further, it can follow a hard-coded approach, using the same words or expressions for each domain situation, or a more varied approach allowing for multiple options of expressing an idea or concept. This task is particularly beneficial for systems with multilingual support.

Referring expression generation: This task is correlated with lexicalization and deals with “*selecting words or phrases to identify domain entities*” [68, p. 11]. Unlike lexicalization, referring expression generation has to consider the position in the text to distinguish the reported entity. For example, a pronoun identifying an entity can only be used if that entity was mentioned right before in the text, establishing a reference between the two.

Linguistic realisation: Finally, linguistic realisation, also known as surface realisation, applies “*the rules of grammar to produce a text which is syntactically, morphologically, and orthographically correct*” [68, p. 12]. This task encompasses rules such as adding propositions and phrase connectors (syntactic component), fulfilling the concordance between gender, number, and verbal tense (morphological component), and the correct use of capital letters and punctuation (orthographic component), amongst others.

2.2.2 Architectures

Reiter and Dale [68] defend that there are multiple ways to distribute the aforementioned tasks. For example, develop a module for each, or represent the tasks by constraints and axioms. However, the most typical architecture of current NLG systems splits the NLG tasks into three main stages: text planning, sentence planning, and linguistic realisation. Text planning contains the first two tasks: content determination and discourse planning. Sentence planning encompasses sentence aggregation, lexicalization, and referring expression generation. Finally, linguistic realisation includes the NLG task with the same name. Figure 2.1 depicts a diagram of the proposed architecture.

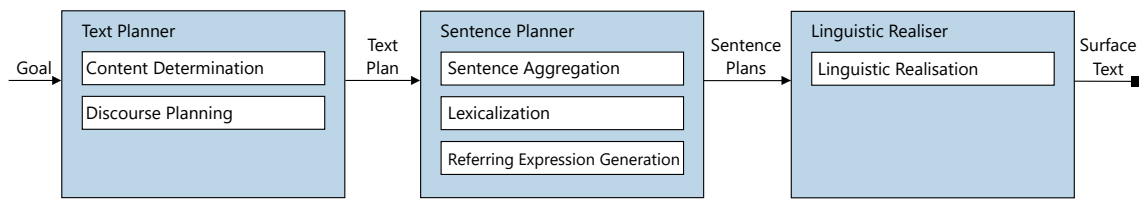


Figure 2.1: NLG systems' architecture and task distribution presented by Reiter and Dale [68]. Adapted from Aires [1].

2.3 Commercial NLG Systems

According to Robert Dale [16], Natural Language Generation was not adequately valued for a long time. One reason that could explain it states that the amount of textual content needing automatic interpretation surpassed on a large scale the quantity of data available for descriptions generation. It was not until more recently that this factor changed. In the 2010s, NLG started to gain relevance through commercial products and increasing customer demand for text generation tools, thus becoming a fully established commercial category in the software market.

2.3.1 The first steps

The first attempts to develop cost-effective commercial NLG applications occurred during the 1990s [16]. The FoG system, designed by CoGenTex, to generate weather reports was the first functional NLG application. CoGenTex¹ is a company founded by Dick Kittredge, Tanya Korlinsky, and Owen Rambow in 1990 that worked on producing bilingual output texts, namely in English and French. Multilingual support is a significant feature of NLG systems, as generating various multilingual outputs from input data provides better results than translating the output after the text generation. On the other hand, some companies like Cognitive Systems, located in the USA, and GSI-Erli, located in France, proposed applications to automatically generate fluent answers to customers' doubts and complaints. Both Cognitive Systems and GSI-Erli have already ceased their activity.

2.3.2 The turning point

On April 24, 2012, Steven Levy published an article in the Wired magazine entitled “*Can an Algorithm Write a Better News Story Than a Human Reporter?*” [50]. Dale [16] considers this publication to be the turning key point that brought the attention of mainstream media to the capabilities of NLG. This article showed what would be possible to accomplish with the right tools and appropriate data through the work employed by Narrative Science, a company founded in 2010, recently acquired by Salesforce² [31], that produced news stories and summaries of sports

¹<http://cogentex.com/>

²<https://www.salesforce.com/>

events and financial reports. It started by presenting a short example of a paragraph automatically written by Narrative Science’s algorithms, catching the reader’s attention to the fluidity of the text and the imperceptible differences between human and computer writing. According to Narrative Science’s CTO, Kristian Hammond, computer-generated textual content will dominate the future of news and stories publication. However, it will not remove humans from the creation process or take their jobs away. NLG will assume a leading role in reporting information journalists cannot cover. More, it can go beyond journalism and be used to generate descriptions of large data sets, explanations of spreadsheets, complex charts, and tables. And consumers seem to be interested in this type of functionality. As Steven Levy [50] says, “*it turned out that (...) people would pay to convert all that confusing information into a couple of readable paragraphs that hit the key points*”.

2.3.3 The market expansion

Since the early days, some major NLG companies have been founded and continuously evolved their products until they became prominent names. Five of those stand out as the most recognized and established companies in the NLG market: Ax Semantics³, Yseop⁴, Automated Insights⁵, Narrative Science⁶, and Arria NLG⁷ [16]. The products and services provided by the previously mentioned companies usually take two approaches:

1. Development and maintenance of a customized NLG tool solution, ordered by a customer to satisfy a particular need;
2. Self-service toolkits for customers to implement their applications, following a Software as a Service (SaaS) model. This approach usually emerges naturally due to the experience acquired during the first one.

In addition, some NLG vendors also integrate their applications into recognized platforms with already established user bases as a strategy to catch users. For example, Automated Insights and Yseop had plug-ins for Excel for their Wordsmith and Savvy products, respectively. Despite some distinguishable features and specificities of each of the products provided, they seem to be very similar in accessibility and utilization. In fact, for the usual use cases, the currently available NLG tools share a big part of the characteristics of their predecessors from the 1990s.

The advancements in the NLG market and companies’ production led to a point in which usability represents a key factor to a product’s success. The end-user should be abstracted from the theoretical notions NLG comprises and be provided with an interface to interact with the system. Dale mentions that “*much more important in terms of the success of the tool is the quality and ease of use of its user interface*” [16, p. 4].

³<https://en.ax-semantics.com/>

⁴<https://yseop.com/>

⁵<https://automatedinsights.com>

⁶<https://narrativescience.com>

⁷<https://www.arria.com/>

2.3.4 Commercial solutions

As stated before, companies tend to follow two approaches to the products and services they provide. According to Dale [16], Ax Semantics opted for the self-service toolkit approach since the beginning. It offers plug-ins and integration with many available platforms and has focused its motivation on improving the generation of product descriptions from structured data. Automated Insights has its Wordsmith NLG platform and offered a free trial period for its users in 2014, and Arria presented its NLG Studio⁸ tool around 2017. Narrative Science had Quill⁹, which generated stories for dashboards, and then developed Lexio¹⁰, an application for generating business news-feeds from Salesforce input data. As for Yseop, it had Savvy¹¹, a data-to-text application with multilingual capabilities. Table 2.1 shows the most recognized NLG companies in the market, and Table 2.2 shows a list of other NLG vendors and their contributions, based on Dale [16, pp. 4-5].

Table 2.1: Reference NLG companies.

Name	Foundation	Country
AX Semantics	2001	Germany
Automated Insights	2007	USA
Yseop	2007	France
Narrative Science	2010	USA
Arria NLG	2013	UK

2.4 Open-Source NLG Systems

In a publication in his blog in 2017, Ehud Reiter [67] stated that there are still too few open-source NLG systems available. Reiter [67] declared that creating or adapting an existing software into an open-source model requires a great effort and is time-consuming. So does its maintenance and continuous support for users of the system. There is still the need to bring more commercial developers into the NLG area, which would encourage the development of more open-source systems and promote knowledge sharing amongst the community. According to Reiter, the “*NLG community is pretty bad at letting its members know about software resources*” [67]. The present section aims to disclose the state of the art in open-source NLG, splitting the existing systems in line with the architecture proposed by Reiter and Dale [68].

A search methodology was followed to find open-source solutions with natural language generation capabilities. Ehud Reiter’s blog [67] was used as a starting point for finding some systems. Then, Google search permitted finding more open-source solutions and relevant documentation using selected keywords. The keywords chosen were “natural language generation”, “open-source NLG”, “no-code NLG”, “NLG systems”, and “NLG state of the art”. The search led to the list of “Downloadable NLG systems” from the ACL wiki [30]. A more in-depth search for articles,

⁸<https://www.arria.com/nlg-studio/>

⁹<https://narrativescience.com/quill/>

¹⁰<https://narrativescience.com/lexio/>

¹¹<http://yseop.github.io/savvy-api/>

Table 2.2: Other NLG vendors.

Name	Foundation	Country	Focus
Linguastat ^a	2005	USA	Product descriptions.
Retresco ^b	2008	Germany	Product description, real estate, sports reporting, traffic news, and stock market reporting.
Infosentience ^c	2011	USA	Sports reporting.
Phrasetech ^d	2013	Israel	NLG with theoretical bases.
2txt ^e	2013	Germany	Product descriptions.
Textual Relations ^f	2014	Sweden	Product descriptions and support for 16 languages.
Narrativa ^g	2015	Spain	Financial services, e-commerce, healthcare and telecommunications.
VPhrase ^h	2015	India	Case studies for industries and multilingual support.

^a<http://www.linguastat.com>

^b<https://www.retresco.de/>

^c<https://infosentience.com>

^d<https://www.phrasetech.com>

^e<https://2txt.de>

^f<https://textual.ai>

^g<https://www.narrativa.com>

^h<https://www.vphrase.com>

GitHub repositories, and web pages' content was done afterwards for each solution found. In addition, from those articles, new systems were pursued through references.

The following subsections are an enumeration of open-source NLG solutions and their characterisation according to their features, time of creation and development, license, and position inside the architecture of NLG systems proposed by Reiter and Dale [68]. Subsections 2.4.1 to 2.4.7 depict linguistic realisers, 2.4.8 to 2.4.13 enclose solutions that are both sentence planners and linguistic realisers, and 2.4.14 to 2.4.18 describe open-source solutions that cover all stages. Finally, in solutions of Subsections 2.4.19 to 2.4.24, neither the authors place them in the architecture, nor it was possible to deduce their position from the information found. Table 2.3 summarises the solutions, and Table A.1 details their characteristics. The solutions marked with a “*” are no longer available.

2.4.1 FUF/SURGE

Functional Unification Formalism Interpreter (FUF) [25, 26] implements the FUG formalism [43], a recognized valuable formalism on developing syntactic realisation grammars for generation, and expands it. The interpreter was written in Common Lisp. In his thesis, Michael Elhadad [26] enumerates some of the contributions of the FUF implementation in contrast to past FUG systems:

- More efficient and can sustain larger grammars;
- It is used for syntactic realisation, lexical choice, and content determination;

Table 2.3: Availability of open-source NLG solutions.

Name	Creation	Last Update
DAYDREAMER*	1983-88	-
LKB	1991	2019
FUF/SURGE	1992-96	2017
KPML	1993	2020
ASTROGEN	1996-99	2005
TG/2*	1998	2005
CLINT*	1999-2000	-
MUG	2002	2004
Suregen-2	2002	-
STANDUP	2003	2007
OpenCCG	2003	2019
NaturalOWL	2007	2008
GenI	2007	2017
NLGen & NLGen2	2009	2010 & 2009
SimpleNLG	2009	2021
PHP-NLGen	2011	2020
TGen	2014	2021
JSrealB	2015	2022
RNNLG	2016	2017
PASS	2017	2021
Syntax Maker	2018	2021
RosaeNLG	2018	2022
Chimera	2019	2020
Elvex	2019	2021
Accelerated Text	2020	2022

- Supports inheritance, making the formalism more concise and expressive;
- Includes mechanisms to treat grammars in a modular way and to handle exchange with external knowledge sources, thus improving readability and robustness in their development.

Systemic Unification Realization Grammar of English (SURGE) [25, 27] was written in FUF by Michael Elhadad and Jacques Robin between 1992-1996 and distributed as a package accompanied by a FUF interpreter. It is “*a syntactic realization front-end for natural language generations systems*” [27]. Figure 2.2 shows an example of an output to a given input to the SURGE system.

According to Essers and Dale, KPML [6] 2.4.2 and FUF [25, 26] are the “*two most well known realisers*” [28, p. 1], alongside their English grammars, namely Nigel [52] and SURGE [25, 27], respectively. Their study comparing the two systems concluded that “*both systems are excellent resources for anyone who needs to incorporate a **surface realiser** into a natural language generation system*” [28, p. 11]. KPML/Nigel requires more abstract semantically oriented input, therefore, the micro-planner can undertake a more relaxed sentence planning. On the other hand, FUF/SURGE deals with more syntactic input, thus, the micro-planner must be aware of the syntactic possibilities and be capable of mapping them with document elements.

Input Specification (I_1):

cat	clause							
	process	<table> <tr> <td>type</td> <td>composite</td> </tr> <tr> <td>relation</td> <td>possessive</td> </tr> <tr> <td>lex</td> <td>"hand"</td> </tr> </table>	type	composite	relation	possessive	lex	"hand"
type	composite							
relation	possessive							
lex	"hand"							
partic	agent	<table> <tr> <td>cat</td> <td>pers_pro</td> </tr> <tr> <td>gender</td> <td>feminine</td> </tr> </table>	cat	pers_pro	gender	feminine		
	cat	pers_pro						
	gender	feminine						
	affected	<div>1</div> <table> <tr> <td>cat</td> <td>np</td> </tr> <tr> <td>lex</td> <td>"editor"</td> </tr> </table>	cat	np	lex	"editor"		
cat	np							
lex	"editor"							
possessor	<div>1</div>							
possessed	<table> <tr> <td>cat</td> <td>np</td> </tr> <tr> <td>lex</td> <td>"draft"</td> </tr> </table>	cat	np	lex	"draft"			
cat	np							
lex	"draft"							

Output Sentence (S_1): "She hands the draft to the editor"

Figure 2.2: SURGE input-output example. Image from Elhadad and Robin [27, Fig. 1].

The source code of FUF/SURGE can be seen in a public GitHub repository¹² and is licensed under GNU General Public License 2.0. More information on the package and documentation can be found on the FUF/SURGE website¹³. There is also a web page¹⁴ explaining how to install FUF/SURGE.

2.4.2 KPML

Komet-Penman Multilingual (KPML) [6] is a platform for large-scale multilingual grammar development, including construction and maintenance, to provide resources for generation applications dealing with the flexibility of expression and speed of generation. According to Reiter and Dale, the system is "a *linguistic realiser based of systemic grammar*" [68, p. 27]. Systemic grammar is an approach that uses a series of choices on linguistic realisation to characterize the sentence being generated, rather than the standard sets of grammar rules that usually translate the input to output [68]. Bateman [7] enumerates the following features KPML offers when integrated into other natural language generation projects:

- A set of standardized linguistic resources and continuously expanding;
- A generation engine to handle the resources;

¹²<https://github.com/melhadad/fuf>

¹³<https://www.cs.bgu.ac.il/~elhadad/surge>

¹⁴<https://www.cs.bgu.ac.il/~elhadad/install-fuf.html>

Common Lisp and CLIM in Unix and Windows¹⁵. Currently, two system versions are available: 4.1 is the most recent, and 3.2 is the legacy one.

2.4.3 TG/2

TG/2 [9, 74] is a template-based generator written in Lisp by Stephan Busemann in 1998. It is a shallow verbalizer that combines canned text, templates, and context-free grammar rules into a single formalism to produce textual and tabular outputs. In addition, it can be integrated with deep generation, it can “reuse the generated substrings for additional solutions” [9, p. 2] and “can be parameterized according to linguistic properties (regarding style, grammar, fine-grained rhetorics etc.)” [9, p. 2]. The generation rules used by TG/2 are condition-action pairs, and they are treated independently from their interpreter. The interpreter is composed of a three-step cycle [74, 10] that starts by defining a set of applicable rules (matching), then proceeds by choosing one of them (conflict resolution), executing it (firing), and repeating the cycle for the extracted sub-structure of the input. Figure 2.4 depicts the architecture of the TG/2 system.

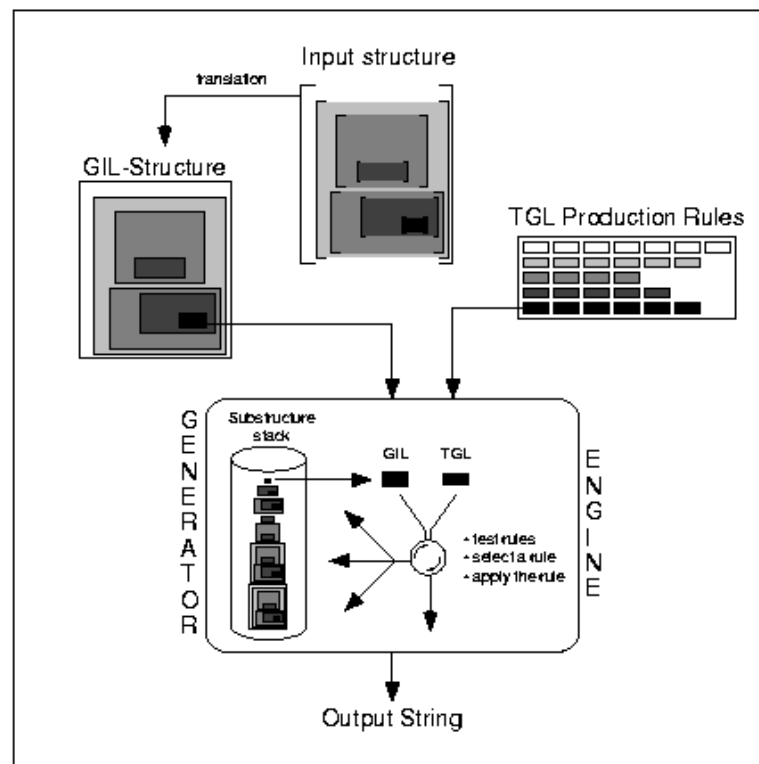


Figure 2.4: Architecture of TG/2. Image from Busemann [9, Fig. 1].

TG/2 web page¹⁶ refers to a hyperlink from which it would be possible to visualize a multilingual demo at work in Chinese, French, English, German, Japanese, and Portuguese of the application. However, the demo is no longer available.

¹⁶<https://www.dfki.de/~busemann/more-tg2.html>

2.4.4 GenI

GenI [46, 44] is a Haskell **surface realiser** developed by Carlos Areces and Claire Gardent in 2007 within the TALARIS project [15]. It takes as input a Feature Based Lexicalized Tree Adjoining Grammar [46] and a set of first-order terms and generates sentences through the semantics of the grammar. The system includes a batch generation mode and a graphical user interface for development and debugging [44]. Figure 2.5 demonstrates an example of the input screen, and Figure 2.6 shows the graphical debugger. According to the ACL wiki entry of GenI [30], some basic example grammars are provided for English and French, and a complete version for the French grammar is in development.

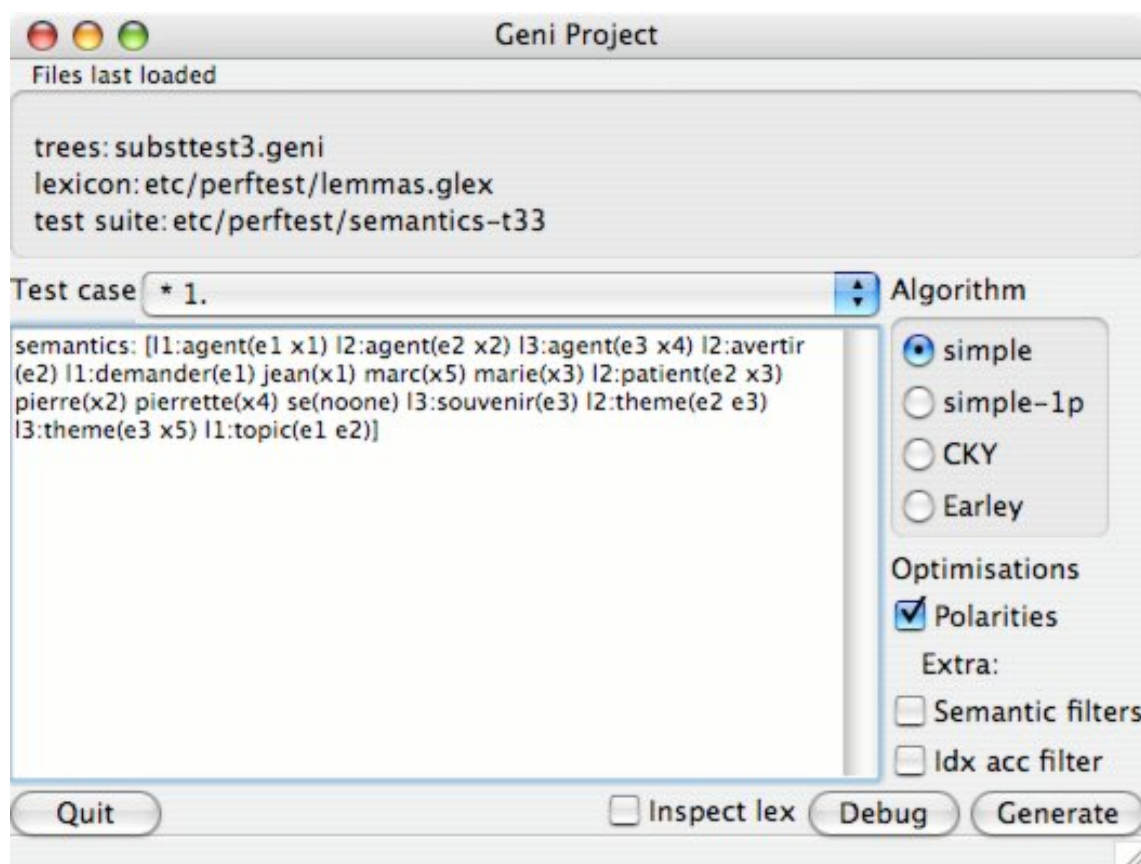


Figure 2.5: GenI input screen. Image from Kow [45].

The source code of GenI is freely available in Hackage, the Haskell community’s package archive of open-source software¹⁷ and licensed under GNU General Public License 2.0.

2.4.5 SimpleNLG

SimpleNLG [34, 35] is a Java library that offers an API to control the **realisation** process programmatically. It was created by Ehud Reiter [34], Professor at the University of Aberdeen’s

¹⁷<https://hackage.haskell.org/package/GenI>

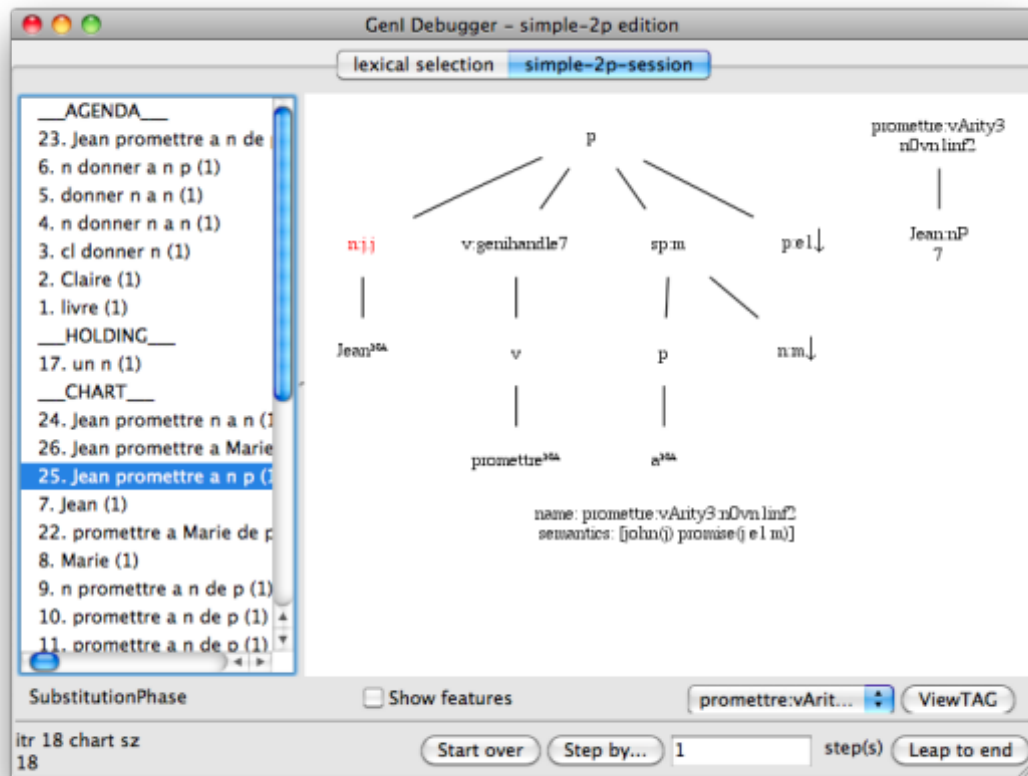


Figure 2.6: GenI debugger. Image from Kow [45].

Department of Computing Science, UK, and co-founder of Arria NLG, in 2009. The current official released version of SimpleNLG only offers support for English. However, several other adjacent projects are in development covering other languages, such as French, Italian, Brazilian Portuguese, German, Spanish and Dutch [35].

SimpleNLG features [35]:

- A lexicon/morphology system. The “*morphological rules (...) cover the full range of English inflection, including regular and irregular forms*” [34, p. 2];
- A realiser for producing natural language from a syntactic form, with limited grammatical coverage in comparison to other known realisers, such as KPML (2.4.2) or FUF/SURGE (2.4.1);
- Basic **micro-planning**, through aggregation, despite not being the system’s focus.

According to Gatt and Reiter [34], SimpleNLG has been well-accepted by the NLG community and used in many projects with different purposes. It can be used as part of a complete project where realisation is not the main focus, as a natural language generator for systems working in other research areas, or ultimately as a teaching tool.

The source code of SimpleNLG and the list of adjacent projects for other languages are freely available in a public GitHub repository¹⁸ and licensed under Mozilla Public License 2.0.

2.4.6 JSrealB

JSrealB is a bilingual **text realiser** for English and French. Paul Molins and Guy Lapalme [55, 4] developed it at the University of Montreal, written in JavaScript to facilitate the integration into web applications.

JSrealB results from the combination of SimpleNLG-EnFr [84], a bilingual version of SimpleNLG (2.4.5) [34] with support for English and French, and JSreal [19], a JavaScript web realiser for French. It can be used individually as a standalone application or as part of another generation system. To start using it integrated with a web application, one needs to import the program and write the command to initiate the loader of resources. JSrealB can produce syntactically correct expressions and sentences and deals with morphology, declension of word syntactic forms, and verb conjugation. Declensions cover gender and number agreement for every grammatical category.

JSrealB follows equivalent procedures to SimpleNLG to execute the generation:

1. Data structures are created to represent the various parts of the sentence to be generated. Usually, these are represented in a tree structure format;
2. Then, the tree is traversed, and a list of the tokens of the sentence is created;
3. The lexicons of JSrealB are based on SimpleNLG-EnFr and can be edited if the user intends to add other domain-specific vocabularies. Each word of the lexicon has grammatical properties and a pointer to an inflection table;
4. JSrealB has stated inflection tables for “*nouns, adjectives, verbs, determiners and pronouns in both English and French*” [55, p. 2].

According to Molins and Lapalme [55], their objective was to develop a text realiser for English and French yet scalable to other languages with less effort. This premise is opposed to SimpleNLG-EnFr and JSreal systems, as their lexicons encompass many irregular forms that hinder the expansion.

The source code of jsRealB is freely available in a public GitHub repository¹⁹ and licensed under Apache 2.0 license.

2.4.7 Syntax Maker

Syntax Maker [40] is an open-source library written in Python that generates sentences in Finnish. According to Hämäläinen and Rueter [40], it focuses mainly on the **linguistic realisation/surface**

¹⁸<https://github.com/simplenlg/simplenlg>

¹⁹<https://github.com/rali-udem/jsRealB>

generation stage of the NLG architecture proposed by Reiter and Dale [68]. It is currently integrated into the Poem Machine [39] system, taking part in the generation of Finnish poetry. By doing so, the tool aims to solve the issue of generating “*novel, grammatical sentences not tackled by the previous Finnish poem generators*” [40, p. 3].

Syntax Maker takes the abstract linguistic structure of a sentence in JSON format as input, that is, specific phrases of a speech nested under each other such that the highest root of the tree is a verb phrase, and noun phrases are assigned to slots of the verb phrase. Then, it evaluates the verb’s valency and characterizes it as transitive, ditransitive, or intransitive. In addition, it analyzes the agreement and government writing rules of a sentence. By resolving the latest, it can then modify verb phrases to produce more complex sentences, such as negations, interrogations, converting to passive voice, and even handling mood and tense.

The source code of Syntax Maker is available in a public GitHub repository²⁰ and licensed under Apache 2.0 license.

2.4.8 ASTROGEN

Aggregated Deep and Surface Natural Language Generator (ASTROGEN) [18, 17] is an NLG system written in Prolog, designed by Hercules Dalianis between 1996-1999 [74].

ASTROGEN is composed of two main modules: the **Deep and the Surface generator**. The Deep generator comprises several sub-modules dealing with **sentence planning** and specializes in **aggregation**. This process depicts the suppression of redundancies from the generated discourse, thus improving readability and fluency [17]. The Surface generator module is a surface grammar of Definite Clause Grammars, using a lexicon of lexical items as terminals. Following a pipeline architecture, as depicted in Figure 2.7, the system receives as input formal specifications, such as the STEP/ EXPRESS standard [18], and uses the previously mentioned modules to translate them to English text. According to Dalianis et al. [18], the ASTROGEN system is still missing however a translator to convert the EXPRESS specifications to the Prolog format and a separate text planner.

The source code of ASTROGEN is freely available on its documentation website²¹, with a copyright Hercules Dalianis [17] notice forbidding the use of ASTROGEN in commercial applications without a license.

2.4.9 MUG

In 2002, David Reitter started the development of the Multimodal Functional Unification Grammar (MUG) system in Prolog [74, 70]. The core idea behind MUG was to achieve a graphical debugging environment and toolset to support the development of functional unification grammars. The system uses a grammar formalism and a hybrid generation algorithm to measure the

²⁰<https://github.com/mikahama/syntaxmaker>

²¹<https://people.dsv.su.se/~hercules/ASTROGEN/ASTROGEN.html>

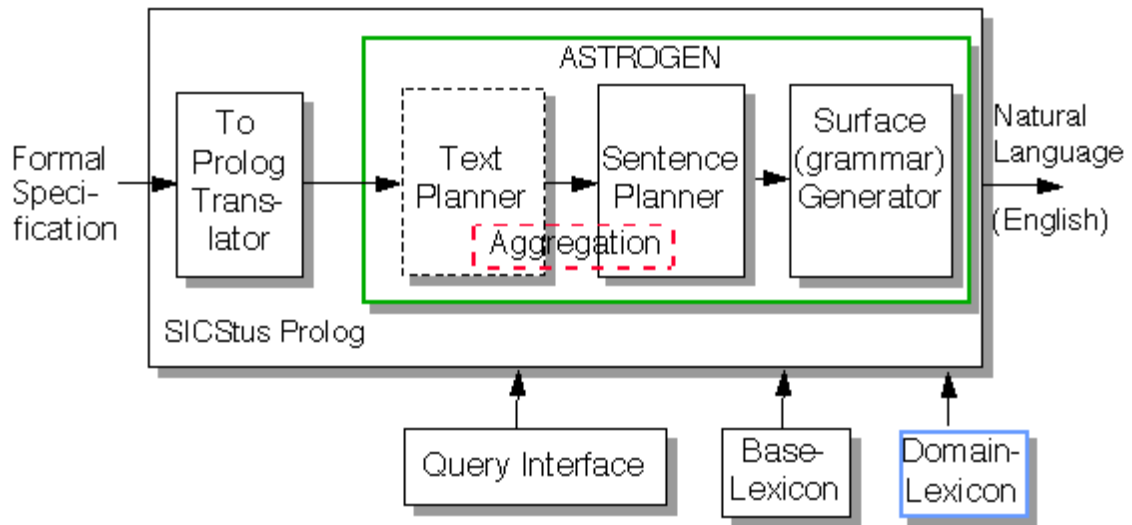


Figure 2.7: ASTROGEN architecture. Image from Dalianis [17].

output’s efficiency by combining soft and restrictive constraints. Besides, it possesses a knowledge base and a **realiser** to “*optimize different constraints using iterative-deepening branch and bound, depth-first search algorithm*” [74].

MUG features include:

- Multimodal fission [38] and arrangement of output’s structure;
- **Sentence planning** of the information to include in the produced utterances;
- Generation of natural language through a graphical user interface with different visualization options. Figure 2.8 shows a view of the MUG workbench.

The zip file with the source code of MUG can be downloaded from David Reitter’s website²², and it is licensed under GNU General Public License 2.0, as stated in the COPYRIGHT file.

2.4.10 OpenCCG

OpenCCG [90, 8] is a system written in Java that uses the Combinatory Categorical Grammar (CCG) formalism [76] for syntax and hybrid logic dependency semantics to parse and generate textual content. In the approach led by Moore et al. [56] to present information in spoken dialogues, the authors used the OpenCCG system for **sentence planning** and **realisation**. In more recent developments undertaken by Michael White and Jason Baldridge [89, 88, 87, 86], the authors tried to continue adapting the realiser utilization into dialogue systems, thus “*improving (...) the grammar development process*” [8, p. 3].

²²<http://david-reitter.com/compling/mug/index.html#dwn>

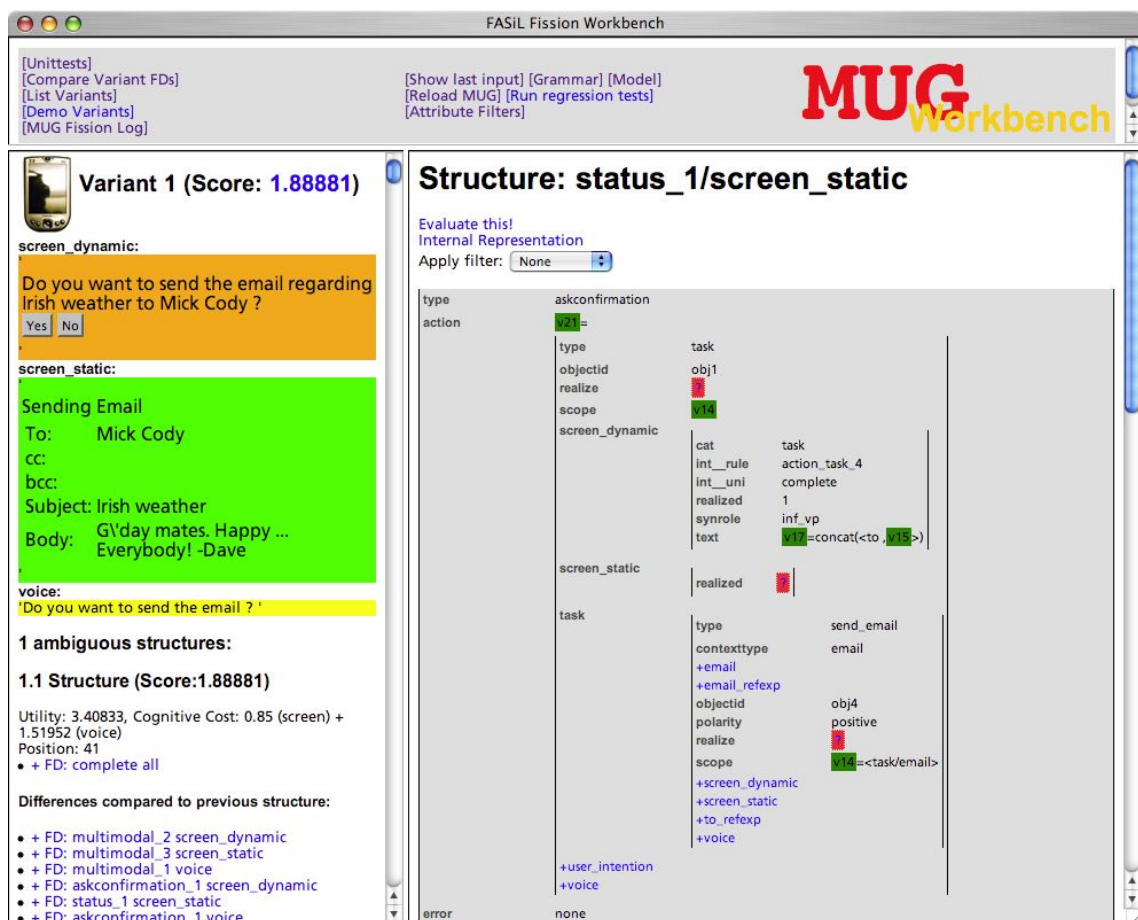


Figure 2.8: MUG workbench. Image from Reitter [69].

The source code of OpenCCG is freely available in a public GitHub repository²³ and licensed under GNU Lesser General Public License 2.1.

2.4.11 TGen

Ondrej Dušek and Filip Jurčiček [23] present TGen, a sequence-to-sequence natural language generator with a recurrent neural network architecture [22] that produces deep syntax trees and regular sentences. It was developed in the Python programming language and had copyright 2014-2019 Institute of Formal and Applied Linguistics, Charles University, Prague.

TGen deals with spoken dialogue systems, that is, the conversion of meaning representations into sentences in English. According to Dušek and Jurčiček [23], the previously mentioned conversion is usually carried out through **sentence planning** and **surface realisation**, either following a two-steps approach, one for each, or a single-step strategy. In contrast to other systems, TGen can execute in both modes, creating an opportunity to compare the two methods. The system can plan and produce sentences directly or generate deep syntax trees to serve as input to an external

²³<https://github.com/OpenCCG/openccg>

surface realiser. In order to achieve that, TGen needs to receive first as input dialogue acts [51] composed of an action type, like *inform* or *request*, and attributes with the corresponding values.

The results of the study led by Dušek and Jurčiček [23] showed that both modes offered proper performance, despite producing different quality outputs. The direct approach was more promising, reflecting higher n-gram based scores with identical semantic errors. It was also concluded that TGen needed fewer amounts of training data to generate high-quality content than other recurrent neural network systems usually do.

The source code of TGen is available in a public GitHub repository²⁴ and licensed under Apache 2.0 license.

2.4.12 RNNLG

The Recurrent Neural Network Language Generation (RNNLG) framework [85] is a Python natural language generator that produces utterances from dialogue acts. It is based on the Recurrent Neural Network Language Model (RNNLM) [54] and on the idea that RNNLM models tend to generate adequate surface realisations when properly conditioned during the training phase [85]. According to Novikova et al. [61], RNNLG approaches the two stages of **sentence planning** and **surface realisation** together, resorting to a Long Short-term Memory (LSTM) [42] network.

As mentioned before, the system receives as input a dialog act that includes a *type*, for example, “*inform, request, confirm, etc*” [85, p. 4] and a set of slot-value pairs, and then produces the appropriate surface realisation. Figure 2.9 depicts the execution flow of the RNNLG system.

The source code of RNNLG, licensed under Apache 2.0 license, and a slides presentation with more in-depth descriptions and hands-on guides can be seen in a public GitHub repository²⁵.

2.4.13 RosaeNLG

RosaeNLG [77] was created by Ludan Stoecklé and became the first NLG open-source software provided by a company [2]. Its initial version goes back to 2018, according to the history of versions provided in the documentation [77]. Ludan Stoecklé is also known for being a former CTO of Yseop [2], a French company founded in 2007 and part of the most recognized NLG companies in the market.

Initially called FreeNLG, RosaeNLG got its current name from Latin, considering that “Rosae” means “Roses” in English. RosaeNLG is a template-based natural language generation library for Node.js and browser integration (client-side), based on JavaScript and the Pug template engine. It provides complete multilingual support for English, French, German, Italian, and Spanish, yet quite a few features for other languages. The system takes input domain data in JSON-like formats, and through templates written with the Pug syntax, it generates text content in HTML format. In addition, it gives the possibility of introducing variations in the sentences. This feature is accomplished with the definition of synonyms, the introduction of conditional texts, where boolean

²⁴<https://github.com/UFAL-DSG/tgen>

²⁵<https://github.com/shawnwun/RNNLG>

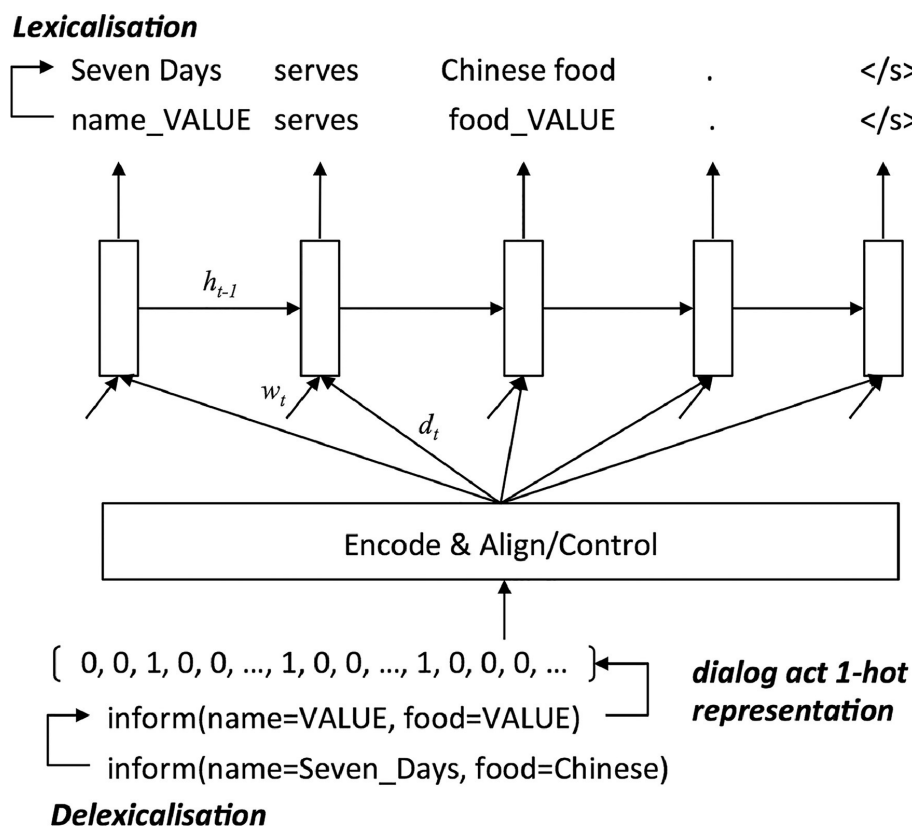


Figure 2.9: The RNNLG framework for language generation. Image from Wen and Young [85, Fig. 1].

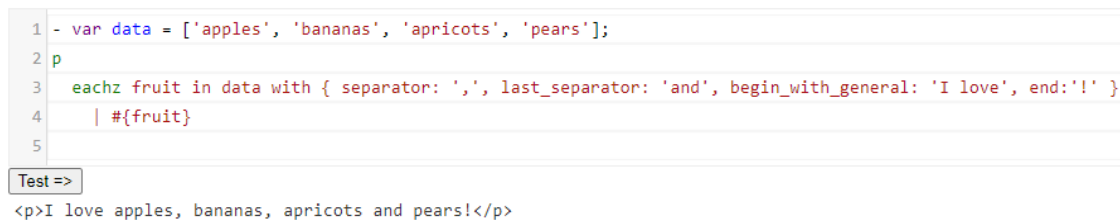
expressions define which text part is used in the generation, and through pronoun replacement. These fall directly into the **Sentence Planner** stage [68], mainly focused on the **lexicalization** and the **referring expression generation** [78]. Finally, it can handle word spacing and automatic capitalization, thus implementing **surface realisation** [78]. Figure 2.10 demonstrates the use of RosaeNLG integrated into a browser.

The documentation of RosaeNLG is freely available online²⁶, as well as some tutorials and demos to get started with the system quickly. The source code is available in a public GitHub repository²⁷, where is also maintained an issues tracker to which the users can report.

Most of the RosaeNLG code is licensed under Apache 2.0 license, and the documentation is licensed under Creative Commons Attribution 4.0 International (CC-BY-4.0) license. Some elements, however, are licensed under different open-source licenses; for example, “*english-ordinals and rosaenlg-cli modules remain under MIT*” [77]. Each linguistic resource package is also licensed, and those can be seen in detail in the documentation of RosaeNLG.

²⁶<https://rosaenlg.org/rosaenlg/3.2.2/index.html>

²⁷<https://github.com/RosaeNLG/rosaenlg>



```

1 - var data = ['apples', 'bananas', 'apricots', 'pears'];
2 p
3 eachz fruit in data with { separator: ',', last_separator: 'and', begin_with_general: 'I love', end: '!' }
4   | #{fruit}
5
Test =>
<p>I love apples, bananas, apricots and pears!</p>

```

Figure 2.10: RosaeNLG demo. Image from Stoecklé [77].

2.4.14 Suregen-2

Suregen-2 [47, 74] is an ontology-based domain-focused system that generates medical reports, namely “*findings, surgical reports or referral letters*” [5], with support for German and English. The project started development in 2002 with Dirk Hüske-Kraus and was written in Allegro Common Lisp [74].

Suregen-2 is composed of several modules distributed according to a pipeline architecture. It has a base ontology for the medicine domain, open to the addition of new concepts relevant to the specific domain of the application. The ontology functions as an organization of the knowledge base, and it conveys filtered information to the generation phase, acting directly in **text planning**. The system also includes a module specialized in conceptual and **sentence aggregation** and modules to handle semantic functions and refer to common medical terminology, phrases, and findings, thus performing **lexicalization** and **referring expression generation**. Finally, Suregen-2 contains a primary **surface generator** yet [47, 5].

The source code and the executable file of Suregen-2 can be downloaded from its website²⁸.

2.4.15 NaturalOWL

NaturalOWL [3, 33, 32] is an open-source natural language generator for Web Ontology Language (OWL) ontologies, written in Java, at the Athens University of Economics and Business, Greece. It features support for English and Greek and is mainly used in Protégé²⁹. It generates descriptions of instances and classes given an ontology as input.

According to Androutsopoulos et al. [3], NaturalOWL follows the typical pipeline architecture described by Reiter and Dale [68], consisting of three stages: **document planning**, **micro-planning**, and **surface realisation**. In NaturalOWL, the document planning stage includes two sub-stages, namely **content selection**, and **text planning**. During content selection, the system starts by considering the ontology and retrieving all the relevant OWL statements that describe a class or entity. Then, it transforms them into message triples and chooses the ones to be used. After, the text planner takes the message triples and orders them. The micro-planning main stage comprises three sub-stages: **lexicalization**, **sentence aggregation**, and **generation of referring**

²⁸<http://www.suregen.de/00023.html>

²⁹<https://protege.stanford.edu/>

expressions. During lexicalization, for each property of the ontology and each language supported, the author behind the domain can come up with some “*template-like sentence plans to indicate how message triples involving that property can be expressed*” [3, p. 19]. Sentence aggregation picks up on the triples created and ordered in the previous step, which corresponds to an organization of the corresponding sentences and links them to improve readability. In terms of referring expression generation, NaturalOWL possesses a restricted number of referring expressions, including Natural Language names, pronouns, and noun phrases. Finally, the surface realisation process converts the internal sentence specifications into text at the last stage. Figure 2.11 shows the complete pipeline stage flow of the NaturalOWL system.

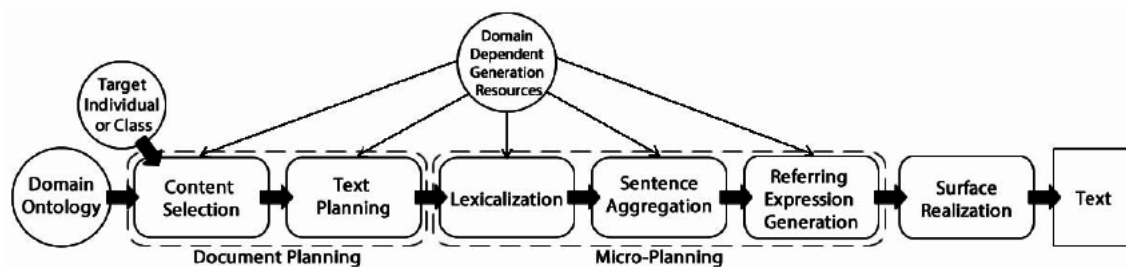


Figure 2.11: The processing stages and sub-stages of NaturalOWL. Image from Androutsopoulos, Lampouras and Galanis [3, Fig. 1].

Currently, two versions of NaturalOWL are available in the Protégé wiki³⁰: 1.0 and 1.1, both compatible with Protege-OWL 3.3.1. A tutorial on how to install and use NaturalOWL is also available online³¹. NaturalOWL is licensed under GNU General Public License 2.0.

2.4.16 PASS

Personalized Automated Soccer texts System (PASS) [82] was developed in 2017 by Chris van der Lee, Emiel Krahmer, and Sander Wubben at Tilburg University, The Netherlands. It is a Python data-to-text system for Dutch football matches reports and statistics, following a re-implementation of the GoalGetter [79] system. It uses input data scrapped from *Goal.com*³² in an XML format to generate professional football matches reports with the possibility of tone switch and adapting to the supporters of each team. The reports produced are divided into four parts represented by separate paragraphs:

- Title – result (win, lose or tie) and score of the match;
- Introduction – a preview of the expectations and most relevant results;
- Game course – a timeline of the match events, linked by the subjective discourse that provides the emotional relatedness;

³⁰<https://protegewiki.stanford.edu/wiki/NaturalOWL>

³¹<http://www.ling.helsinki.fi/kit/2008s/clt310gen/docs/NaturalOWL-README.pdf>

³²<http://www.goal.com>

- Debriefing – the consequences to the teams that come from the results and an overview of the next matches.

PASS [82] has been designed following a modular approach so that each module could be operated, enhanced, or swapped separately. The generation is similar for each of the four text parts instantiated above. However, while the order of the topics is fixed for the title and introduction parts – “*title, win/tie/loss information and the final score*” [82, p. 5], it is not the case for the game course and debriefing, which depend on the match events. So, the topic collection module was added to extract and order the topics. Having sorted the topics, the system then follows a set of steps. Firstly, a topic is chosen to be handled. After, the lookup module analyzes and provides the template categories and respective sets of templates regarding the current topic. While some categories are general-purpose, some can only be used when specific conditions are fulfilled. The ruleset module is responsible for checking whether the conditions are met for each category, returning “true” in case of success and “false” otherwise. For each successful category, its templates are moved to a list of possible templates. Then, the template selection module follows a weighted selection approach and picks a template. The template filler module substitutes the empty entrances in the template with data. When all the parts have their corresponding text generated, the text collection module is responsible for combining them in the correct order. At this point, the system is fully operational to produce text, but some other modules were added to increase variety. The information variety module guarantees that some pieces of information are not repeated. The reference variety module goes throughout the text and substitutes repeated referents with new ways to address entities. Finally, the between-text variety module retains track of previously used templates for reports generation and ensures they are not repeated in future reports. Figure 2.12 shows a sequential diagram of the modular architecture of PASS.

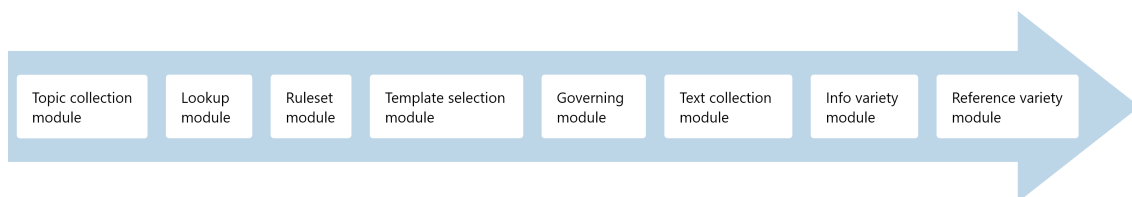


Figure 2.12: Modular architecture of PASS. Adapted from Lee, Krahmer and Wubben [82, Fig. 1].

The source code of PASS is freely available in a public GitHub repository³³ and licensed under GNU General Public License version 3.0.

2.4.17 Chimera

Amit Moryossef et al. refer to data-to-text generation as two steps, namely “*ordering and structuring the information (planning), and generating fluent language describing the information (realization)*” [57, p. 1]. In contrast with the modern neural generators that merge the two steps into

³³<https://github.com/TallChris91/PASS>

a single end-to-end architecture, Chimera proposes a different approach, splitting generation into two distinct tasks: a symbolic **text-planning** stage and a neural generation stage for **linguistic realisation** [57]. In addition, Chimera can also perform **referring expression generation**, which fits between planning and realisation.

Chimera is a recent project published in 2019 and written in Python. It receives RDF triplets as input describing facts of entities and their intertwined relations and generates fluent text content that matches those facts. Example RDF triplets are depicted in Listing 2.1.

```
1   John , birthPlace , London
2   John , employer , IBM
```

Listing 2.1: Example Chimera RDF triplets input from Moryossef, Goldberg and Dagan [57].

Example sentences generated (source: Moryossef, Goldberg and Dagan [57, p. 1]):

1. *“John, who works for IBM, was born in London”.*
2. *“John works for IBM and was born in London”.*
3. *“London is the birthplace of John, who works for IBM”.*
4. *“John works for IBM. John was born in London”.*

Amit Moryossef et al. [57] evaluated it under the WebNLG benchmark. The study concluded that splitting text planning and neural realisation improved the reliability and adequacy of the system without spoiling the fluency of the output.

The source code and the training and development data sets of Chimera are freely available in a public GitHub repository³⁴ and licensed under MIT license.

2.4.18 Accelerated Text

Accelerated Text [80] had its first public release in 2020 by TokenMill UAB in JavaScript and Clojure programming languages. It is a data-to-text engine for generating data descriptions, with a visual interface of building blocks, following the no-code paradigm [93] in a Google Blockly³⁵ environment. It allows the variation of the content’s wording and structure by switching the blocks inside the root, “Document plan”, as depicted in Figure 2.13. In addition, it has support for multiple languages, like English, German, Russian and Spanish.

According to the documentation [81], the data used by Accelerated Text can come from many different origins, such as business and financial metrics, customer interaction, and product attributes. Its main features include:

1. A document plan editor to define the information being reported about the data;

³⁴<https://github.com/AmitMY/chimera>

³⁵<https://developers.google.com/blockly/>

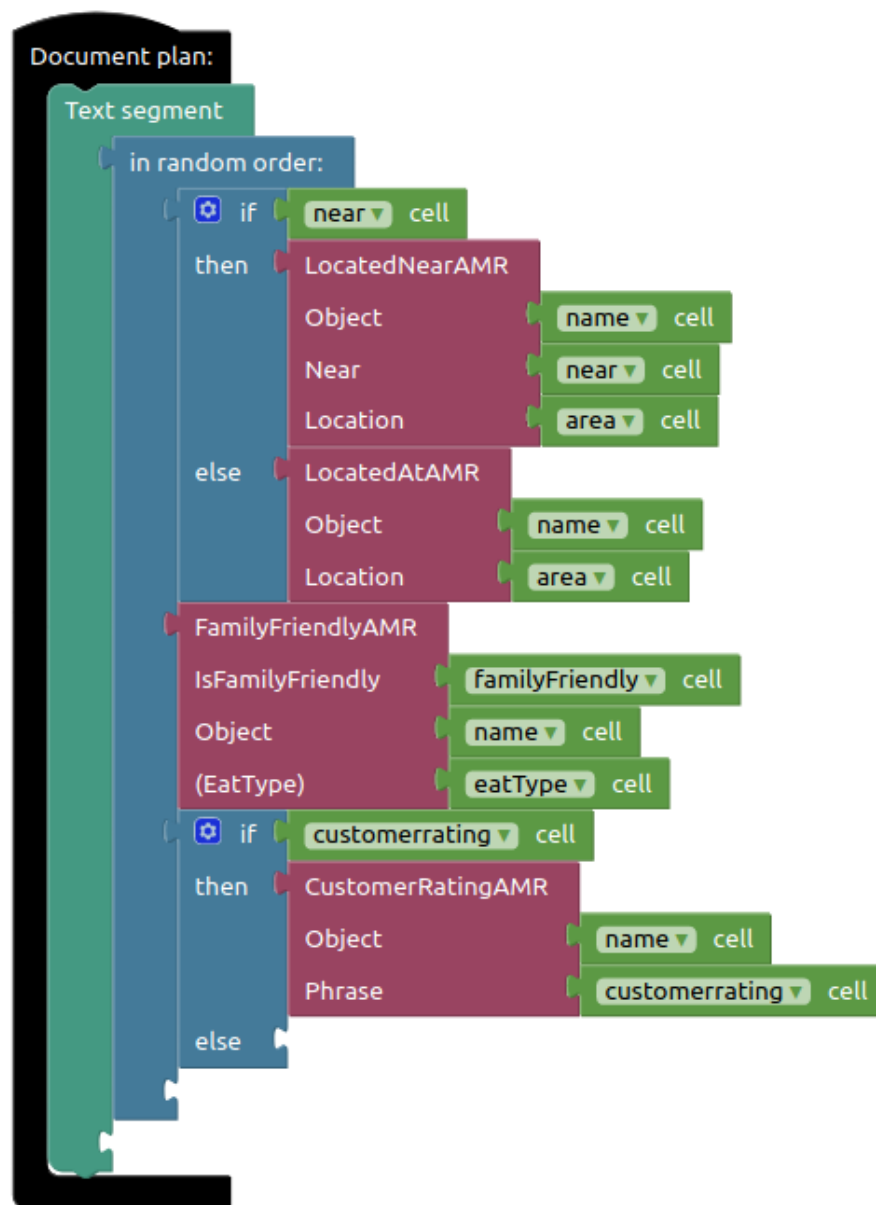


Figure 2.13: Accelerated Text Document Plan. Image from Navickas [92].

2. Data samples in the form of CSV files for input;
3. Text structure variations to stimulate readability and variety into the generated texts;
4. Language and vocabulary control to adapt to the reader's specific features;
5. A built-in rule engine to control "what is said based on the different values of data points" [81];
6. Live preview of the generated text.

The source code of Accelerated Text is publicly available in its GitHub repository³⁶ and licensed under Apache 2.0 license.

2.4.19 DAYDREAMER

Mueller and Dyer define daydreaming as “*the spontaneous human activity of recalling or imagining personal or vicarious experiences in the past or future*” [59], and enumerate some of its practical usages:

1. Planning for the future – It is the idea that when we drift off in thought about different scenarios and possible ways for events to happen, we also envision adequate responses to those situations. Thus daydreaming enhances the ability to make decisions.
2. Learning from successes and failures – The concept is that when daydreaming, by reflecting on already occurred successful or failed events and coming up with alternative actions, we are enhancing the ability to plan strategies. Since it is a thinking process that happens after the event, it can be done with newly acquired knowledge.
3. Creativity – Daydreaming promotes the search for new, outside the box ideas and solutions for problems, with continuous revision and reformulation. This process can also increase the motivation to undertake those solutions.
4. Emotion regulation – The process of thinking of different possible outcomes to events and changing the perception according to that rationalization. For example, “*Fear associated with a future event may be reduced if one daydreams about effective plans to succeed in that event, or increased if daydreams of likely failure result*” [59].

The DAYDREAMER [59] program was developed in 1983-1988 [58] using GATE [60], a development environment for the T language [65], a dialect of Scheme. It aimed to test the theory of daydreaming through computational means. The program receives as input “*simple situational descriptions*” [59], or casual events, and generates two outputs: suggested actions for the present scenario and daydreams, only in the English language. DAYDREAMER has five main components, each exchanging information with another. The scenario generator uses experiences provided by the dynamic episodic memory and is guided by the collection of goals. The dynamic episodic memory is long-term and encloses the complete daydreams, the planned actions, and the strategies created from daydreaming. Goals can be personal, that is, all basic needs, such as, “*health, food, sex, friendship, love, possessions, self esteem, social esteem, enjoyment, and achievement*” [59] or control goals. They are ordered in a tree structure regarding importance in time. Then the scenario generator chooses the path that meets the goal with the highest importance. The emotion component initiates the daydreams and manages the emotional states that result from the goal outcomes. Finally, the last component encompasses the domain knowledge of interpersonal relations and daily mundane events.

³⁶<https://github.com/accelerated-text/accelerated-text>

Despite some claims that the DAYDREAMER system is still available at the CMU AI Repository³⁷ and licensed under GNU General Public License version 2.0 [20], neither the source code nor the executable file was found.

2.4.20 LKB

Linguistic Knowledge Builder (LKB) [13] was originally developed in 1991, and it is a system for grammar development written in Common Lisp and distributed as part of the open-source LinGO tools [12].

According to Copestake et al. [13], LKB was the development environment of many grammars with different sizes and languages. LKB features also make it a fit in teaching due to its efficiency and availability as open-source, taking part in formal syntax, computational linguistics, and grammar engineering courses [13]. The system is composed of “a parser, generator, support for large-scale inheritance hierarchies (...), various tools for manipulating semantic representations, a rich set of graphical tools for analyzing and debugging grammars, and extensive on-line documentation³⁸” [13, p. 1]. Figure 2.14 shows a screen dump view of the LKB system.

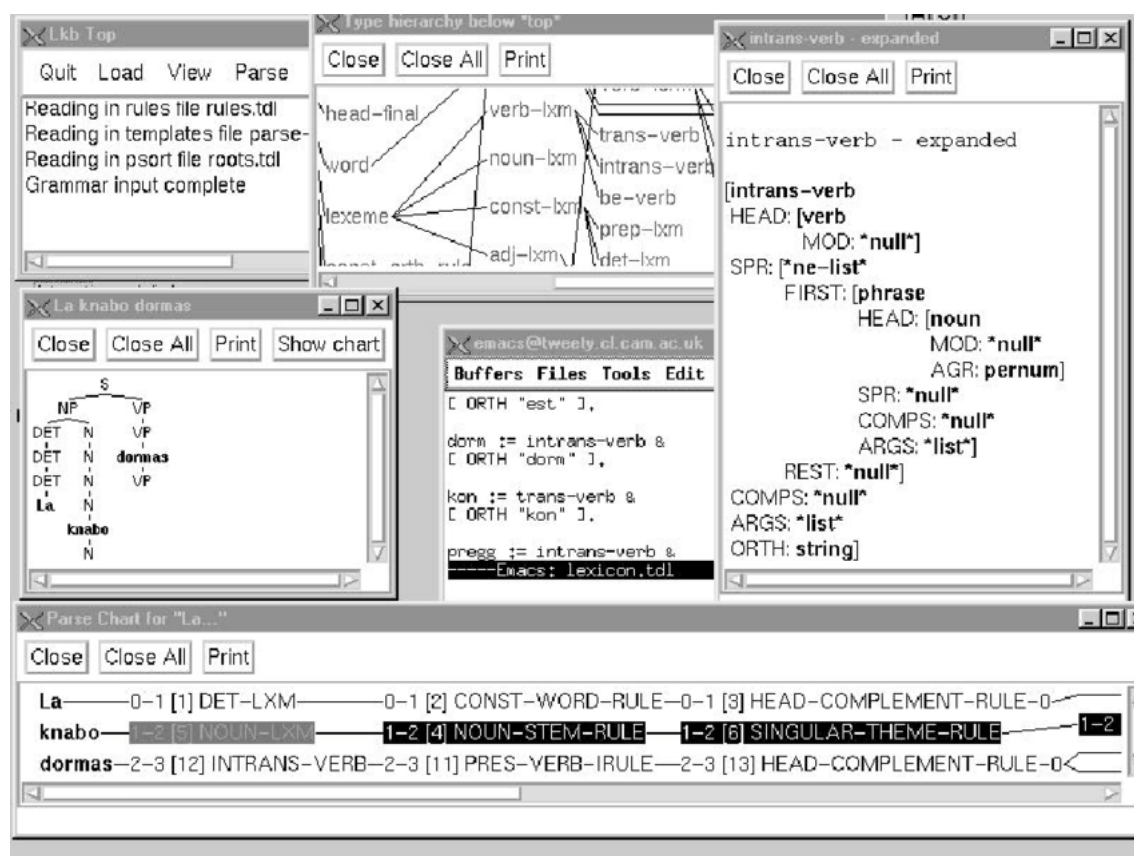


Figure 2.14: LKB screen dump. Image from Copestake et al. [13, Fig. 1].

³⁷<https://www.cs.cmu.edu/Groups/AI/0.html>

³⁸<https://github.com/delph-in/docs/wiki/LkbTop>

The source code of LKB is freely available in a public GitHub repository³⁹ and licensed under MIT license.

2.4.21 CLINT

CLINT [24, 74] was designed by Rinat Gedalia and Michael Elhadad between 1999-2000 in C++ and ran under Microsoft Windows 3.1. It is a hybrid text generator, combining template-based and word-based generation to create text content. CLINT is composed of four components:

- A template development system;
- A problem definition system that distributes templates following a decision tree structure;
- A run-time generator that inquires the user based on the decision tree and obtains the values to assign to the parameters;
- A noun-phrase generator that receives the parameter values and chooses the adequate noun-phrase form according to the discourse context at each point.

CLINT web page⁴⁰ refers to a hyperlink from which it would be possible to download a zip file containing an executable *clint.exe* and an example grammar template. However, this hyperlink redirects to a not found page.

2.4.22 STANDUP

System To Augment Non-speakers' Dialogue Using Puns (STANDUP) [74, 62] is a Java project created by Manurung et al. [53] in October 2003 and finished support in March 2007. It is a jokes generation system in English, and its "*humor-creating behaviour reflects the current state of the art in computational humour*" [53, p. 2]. The main goal of STANDUP is to promote an exploring environment and friendly user-interface children with complex communication needs can use to develop their language skills by creating puns in a question-answer pair. An example of a generated joke: "*What kind of tree is nauseated? A sick-amore*" [53, p. 2].

In December 2010, a new version of the system was made available, called STANDUP 2, with slight differences from its predecessor.

The source code of STANDUP and STANDUP 2 can be downloaded from the University of Aberdeen website⁴¹, with a copyright notice to the University of Aberdeen, University of Dundee, and the University of Edinburgh.

2.4.23 NLGen, NLGen2

NLGen [74, 37] is a Java system designed by Ben Goertzel in 2009 that generates sentences in English from ReLex semantic relationships using the SegSim algorithm⁴². NLGen is based on

³⁹<https://github.com/jingtaozf/lkb>

⁴⁰<https://www.cs.bgu.ac.il/~elhadad/clint.html>

⁴¹<https://www.abdn.ac.uk/ncs/departments/computing-science/software-480.php>

⁴²https://wiki.opencog.org/w/Natural_language_generation#SegSim

the idea that “most language generation may be done by reversing previously executed language comprehension processes” [37, p. 2]. RelEx is a natural language comprehension engine that “takes sentences and maps them into abstract logical relations which can be represented in the OpenCog Atomspace” [37, p. 2]. NLGen iterates through the predicates of the Atoms and creates a graph for each, representing the RelEx semantic relationships. Then, by confronting these graphs with sentences and semantic interpretations stored in memory, NLGen can merge the selected predicates and generate new sentences.

NLGen 2 [49, 74] was written by Blake Lemoine, also in Java, in 2009. While NLGen could not generate complex sentences in a timely manner, NLGen 2 could overcome that scaling limitation “by using a psychologically realistic generation strategy that proceeds through symbolic stages from concept to surface form” [49]. In addition, while NLGen was dependent on a corpus, NLGen 2 used a Link Parser’s dictionary instead as its knowledge source. Figure 2.15 depicts the architecture of NLGen 2. Both NLGen and NLGen 2 are part of the OpenCog [41] project, a software development framework designed to merge multiple Artificial Intelligence methodologies.

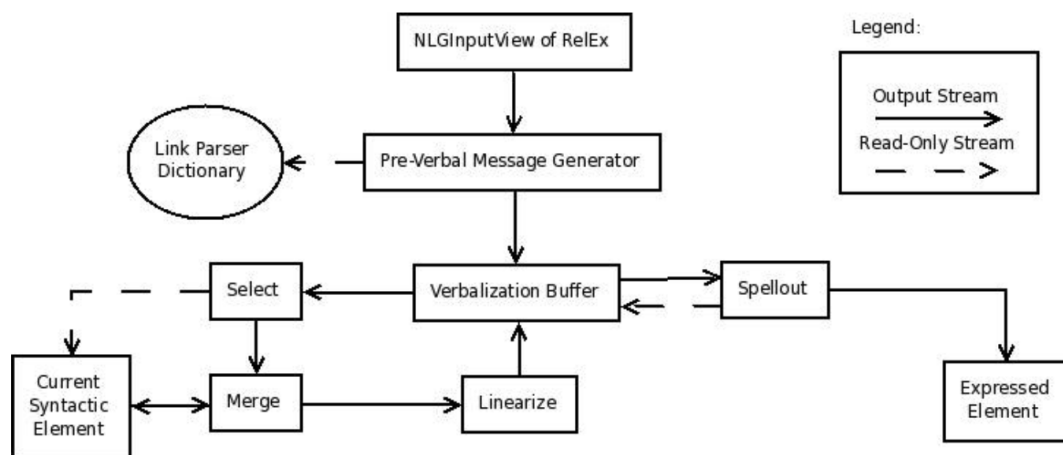


Figure 2.15: General architecture of NLGen 2. Image from Singh et al. [74, Fig. 4].

The source code of both NLGen⁴³ and NLGen 2⁴⁴ can be browsed and downloaded from the Launchpad website and licensed under Apache 2.0 license.

2.4.24 PHP-NLGen

PHP-NLGen [21] is a framework of PHP classes developed by Pablo Duboue. It includes a generator, an ontology container, and a lexicon container of lexical entries, which can create recursive-descent natural language generators. In the recursive-descent generation, each grammar rule is

⁴³<https://launchpad.net/nlgen>

⁴⁴<https://launchpad.net/nlgen2>

represented in a function with two return values: the generated string and the corresponding dictionary of semantic info. Duboue [20] refers to the DAYDREAMER system (2.4.19) [59] as a precursor to the development of such generators.

PHP-NLGen had its first version published in August 2011 in a public GitHub repository⁴⁵, where the source code of the latest version is still available and licensed under MIT license. Some tutorials and basic examples are provided, and a slide presentation with demos in English and French.

2.4.25 Elvex

Elvex [11] is a Natural Text Generator in C++, with support for English and French, developed by Lionel Clément at Bordeaux University, France. The system uses as parameters a handwritten lexicon and grammar. It also receives a *concept* as input, a meaning or elements of speech, such as language level, linguistic utterance acts, and enunciative properties. Then, it produces text content that represents the *concept*. Unlike similar systems, the concepts do not have to depend on previously made lexical choices.

The source code of Elvex is freely available in a public GitHub repository⁴⁶ and licensed under GNU General Public License 3.0.

2.5 Literature Review Analysis

2.5.1 Citation analysis

A citation analysis of the bibliography for the open-source NLG solutions was done to evaluate the references' quality. "*Citation analysis involves counting the number of times an article is cited by other works to measure the impact of a publication or author*" [63]. Google Scholar⁴⁷ and Scopus⁴⁸ were used to count the number of articles, proceedings, books and thesis citations. Table 2.4 shows the top 10 references, sorted according to the average number of citations between Google Scholar and Scopus counts. Following the same approach, the number of citations and *h-index* were considered to evaluate the impact of each author. An author's *h-index* represents the maximum number of papers (*h*) of that author with at least *h* citations each [63]. Table 2.5 and Table 2.6 show the top 15 authors sorted according to the Google Scholar and Scopus *h-indexes*, respectively. The values were obtained on August 6, 2022.

⁴⁵<https://github.com/DrDub/php-nlgen>

⁴⁶<https://github.com/lionelclement/Elvex>

⁴⁷<https://scholar.google.com>

⁴⁸<https://www.scopus.com>

Table 2.4: Top 10 open-source NLG references.

Reference	Type	Scopus	Google Scholar	Average
Hochreiter and Schmidhuber [42]	article	41,360	69,308	55,334
Wen and Young [85]	proceedings	3,223	6,474	4,848.5
Reiter and Dale [68]	article	264	2,823	1,543.5
Gatt and Reiter [34]	proceedings	240	476	358
Novikova et al. [61]	article	143	324	233.5
Bateman [7]	article	112	337	224.5
Dušek and Jurcicek [23]	proceedings	99	175	137
Mairesse et al. [51]	proceedings	87	164	125.5
Elhadad and Robin [27]	proceedings	60	185	122.5
White [88]	proceedings	82	143	112.5

Table 2.5: Top 15 authors, according to Google Scholar h-indexes.

Reference	Citations	H-Index
Jürgen Schmidhuber	171,599	110
Steve J. Young	39,068	91
Sanjeev Khudanpur	28,769	61
Ido Dagan	19,355	59
Yoav Goldberg	23,005	58
Johanna D. Moore	12,698	57
Lukás Burget	25,988	57
Ehud Reiter	12,629	54
Sepp Hochreiter	99,080	52
John Carroll	9,601	52
Emiel Krahmer	11079	50
Tomás Mikolov	122,129	48
Robert Dale	11,684	47
Anne Copestake	9,653	46
Oliver Lemon	7,311	45

2.5.2 Vendors

Years between 2000 and 2015 comprised the foundation of many NLG vendors worldwide that are still active today, including the total of the reference NLG companies. Considering the companies described in Subsection 2.3.4, Europe was the foundation stage of more than 50% of those companies, with Germany founding three out of seven. The USA also contributed four, two of which are current reference NLG companies: Automated Insights and Narrative Science. Asian countries, namely Israel and India, launched two companies.

2.5.3 Open-source solutions

As described in the bar chart of Figure 2.16 and considering the systems found, the number of open-source NLG solutions had been following a trend of increase, with more solutions being created in each decade: 1 from 1980 to 1990, 6 from 1990 to 2000, 8 from 2000 to 2010, 9 from

Table 2.6: Top 15 authors, according to Scopus h-indexes.

Reference	Citations	H-Index
Jürgen Schmidhuber	87,971	71
Steve J. Young	10,933	54
Sanjeev Khudanpur	15,445	46
Lukás Burget	10,660	42
Yoav Goldberg	6,832	35
Ido Dagan	4,935	34
Sepp Hochreiter	51,857	33
Kai Yu	4,365	33
Emiel Krahmer	4,218	33
Johanna D. Moore	2,787	30
Martin Karafiát	6,423	28
Oliver Lemon	2,530	28
Tomás Mikolov	47,569	27
Milica Gašić	3,826	27
Ehud Reiter	3,314	27

2010 to 2020, and 1 in the last two years. Of these, almost 90% are still available for public use, and around 64% had updates during the last decade. These results can depict a growing interest by companies and universities in open-source NLG systems.

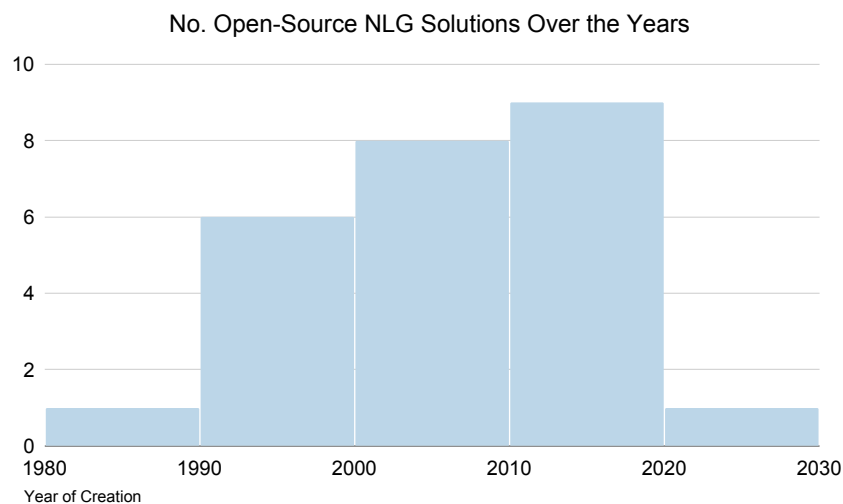


Figure 2.16: Open-source NLG solutions over the years.

Regarding the systems currently available at GitHub, the number of stars and forks made to the repositories by GitHub's users can be considered as a criterion of popularity. Figure 2.17 presents a bar chart with the values of those measures by the 3rd of August 2022, sorted in order of increasing number of stars. Of the solutions considered, SimpleNLG [34, 35] stands out as the most popular with 765 stars and 181 forks, followed by Accelerated Text [80] with 607 stars and

33 forks, and RNNLG [85] with 409 stars and 126 forks.

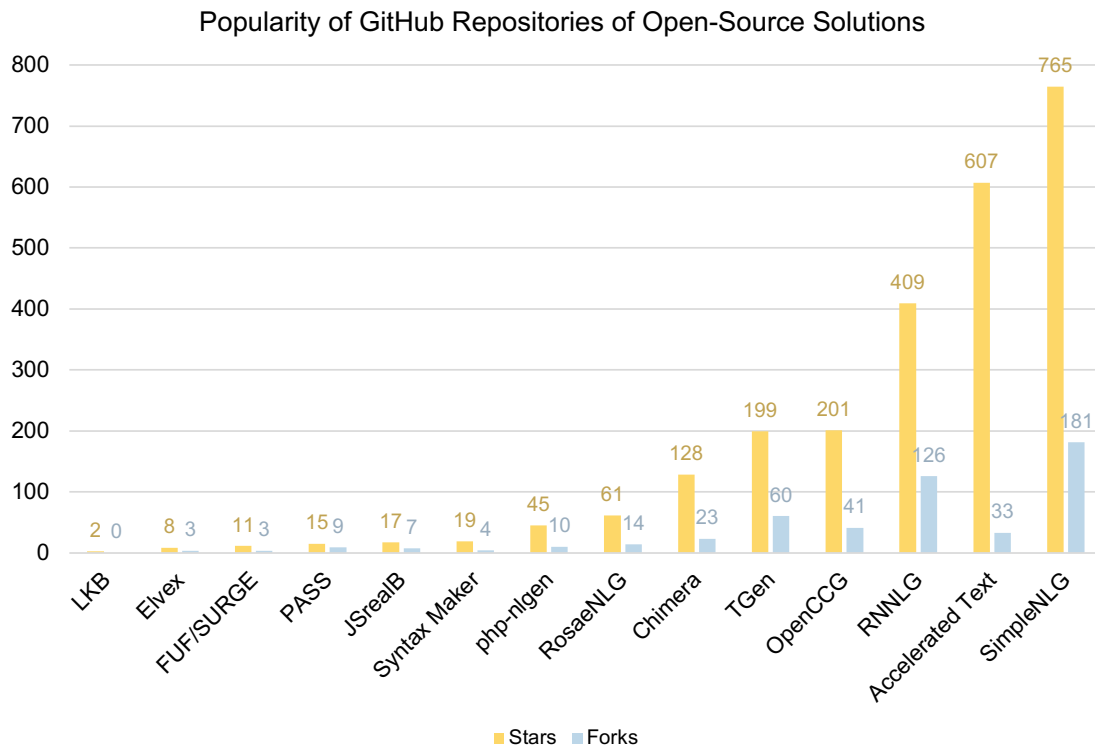


Figure 2.17: Open-source NLG solutions popularity.

Attempting to find a relation between the GitHub factors of popularity and citation analysis numbers, a new analysis was made by crossing the two methods. The number of stars and forks, the average *h-index* of the most popular author and the average number of citations of the most popular reference regarding an open-source solution were considered. Each value was then weighted to a number between 0 and 14, considering the 14 open-source solutions analysed and the values equal to zero for some measures, and then were summed to achieve the popularity value. The final result depicts a slight change in the order of popularity of the solutions, with RNNLG being the most popular, as described in Table 2.7.

In terms of languages, the open-source NLG systems tend to find English the most regular choice for the generation, with 22 solutions out of 25 (88%) supporting it. French and German follow English with 32% and 24%, respectively. Figure 2.18 shows a bar chart of the percentage of solutions with support for a set of languages. 16% of the systems have yet generation for other not so commonly supported languages, for example, Brazilian Portuguese, Czech and Finnish. Turning the focus to the number of languages supported, 13 of the systems found are monolingual, with 11 for English, PASS [82] for Dutch and Syntax Maker [40] for Finnish. Bilingual support is also usual, especially a combination of English and French. Multilingual support is a widespread way of approaching open-source NLG, representing 24% of the systems when counting only the ones with more than two languages but corresponding to 48% when including the bilingual ones.

Table 2.7: Popularity of open-source NLG solutions through analyses crossing.

Name	Stars	Forks	Average h-index of the most popular author	Average no. citations of the most popular reference	Popularity
RNNLG	409 (12)	126 (13)	90.5 (14)	55,334 (14)	1° (53)
SimpleNLG	765 (14)	181 (14)	40.5 (10)	358 (13)	2° (51)
TGen	199 (10)	60 (12)	72.5 (13)	137 (12)	3° (47)
Chimera	128 (09)	23 (09)	46.5 (12)	87 (09)	4° (39)
OpenCCG	201 (11)	41 (11)	27.5 (06)	112.5 (10)	5° (38)
PASS	15 (04)	9 (06)	41.5 (11)	34.5 (08)	6° (29)
FUF/SURGE	11 (03)	3 (03)	35 (08)	122.5 (11)	7° (25)
JSrealB	17 (05)	7 (05)	28 (07)	15 (07)	8° (24)
Accelerated Text	607 (13)	33 (10)	0 (00)	0 (00)	9° (23)
Syntax Maker	19 (06)	4 (04)	9.5 (05)	10 (06)	10° (21)
RosaeNLG	61 (08)	14 (08)	0 (00)	0 (00)	11° (16)
LKB	2 (01)	0 (00)	37.5 (09)	0.5 (05)	12° (15)
php-nlgen	45 (07)	10 (07)	0 (00)	0 (00)	13° (14)
Elvex	8 (02)	3 (03)	0 (00)	0 (00)	14° (05)

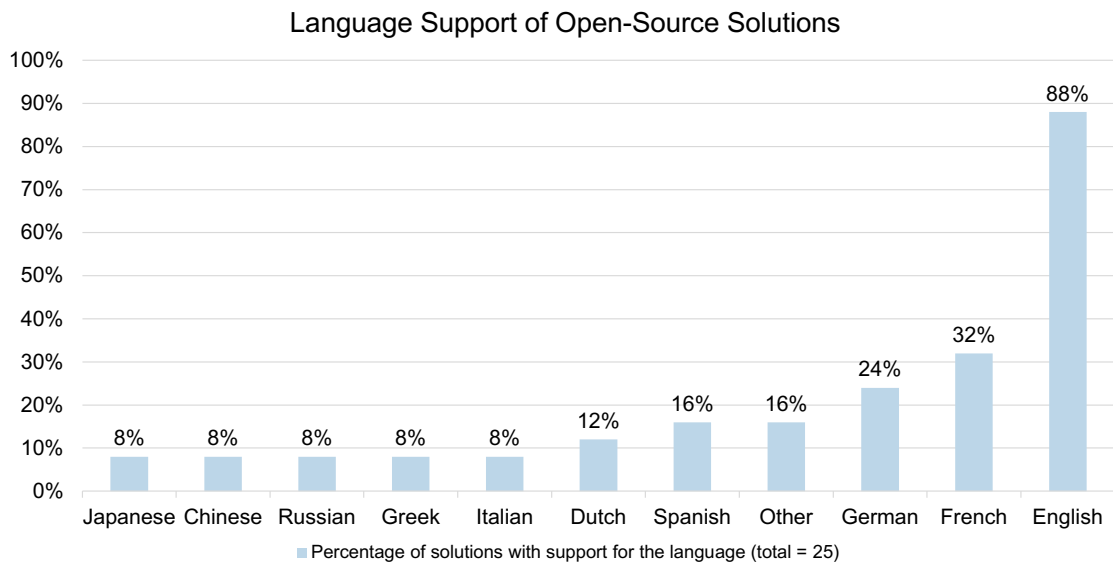


Figure 2.18: Language support of open-source NLG solutions.

When comparing programming languages, Python, Java and Lisp are the most used, with five systems written in each. JavaScript, alone or together with Clojure or Pug, was used to write three of the systems, and Prolog and C++ were used to develop two systems each. Other less used programming languages include Haskell, T and PHP.

Finally, considering only the 20 licensed open-source solutions, there is a general preference for the GNU, Apache and MIT licenses in this order, respectively, 40%, 35% and 15% of the

solutions. Figure 2.19 depicts a pie chart with the distribution of licenses through the solutions. It is essential to notice that GNU GPL v2.0, GPL v3.0 and LGPL v2.1 were joined into a single category.

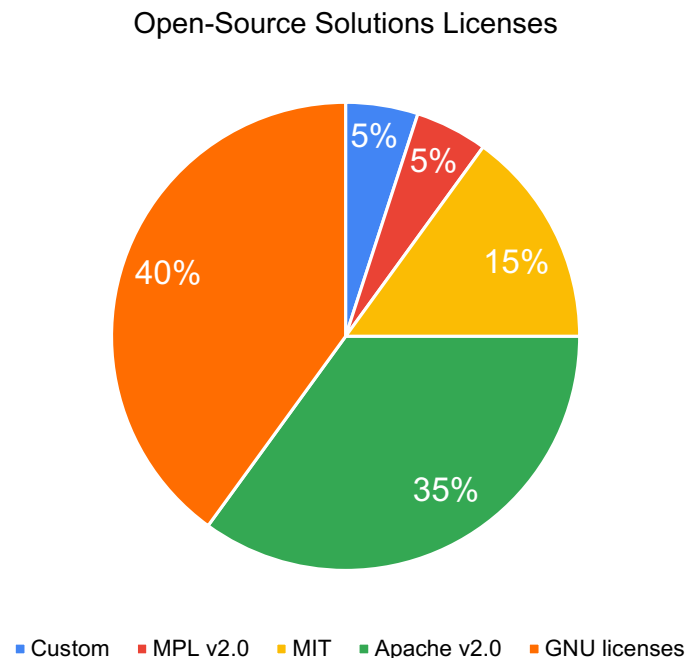


Figure 2.19: Open-source NLG solutions licenses.

2.6 No-Code Paradigm

The low/no-code [91] paradigm defines an approach to software development based on user interfaces to facilitate the creation and interaction with applications so that lay users with low or no programming skills can also enter the development environment. According to an investigation conducted by Richardson and Rymer in 2016 [72], when interviewing companies regarding their low/no-code platforms, the “vast majority (...) reported that their low-code platforms helped them accelerate development by five to 10 times” [72, p. 5].

According to Medelis [93], co-founder and CEO at TokenMill⁴⁹ and Accelerated Text⁵⁰, no-code is “a perfect fit for Natural Language Generation”. He believes an NLG system must enhance the author’s capabilities instead of completely substituting the human author in the writing process. In addition, he declares that the no-code approach can tackle some of the complexity NLG comprises by giving easier ways to manage the production of text content, acknowledge and adapt to business requirements, and generate text that relates directly to the input data.

⁴⁹<https://www.tokenmill.ai/>

⁵⁰<https://www.acceleratedtext.com/>

Following the thoughts of Yan [91], some benefits of the low/no-code approach can be enumerated:

- **Swiftness:** the use of interfaces to visually manage and prototype new applications speeds up the development process, providing a direct path to continuous testing, user requirements' refining, and showcasing;
- **Citizen development:** the introduction into the development and delivery processes of lay users with no programming skills, fully aware of the business requirements and domain knowledge, has the potential to increase the meet of customers' needs;
- **Security:** *"If each component or building block in low/no-code platforms is secure, reusable, and optimized, the applications built with them should automatically be secure and optimized"* [91, p. 3];
- **Maintainability:** the existence of a single and centralized development environment platform for all collaborators leads to lower maintenance costs.

On the other hand, the low/no-code approach may fall into some restrictions that reside mainly in the products' scalability and level of closure. Yan [91] describes the following limitations:

- **Customizability/Flexibility:** the components used to build and interact with the application are usually predefined and fixed, resulting in low levels of customization of the products' features;
- **Scalability:** low/no-code platforms have limited scalability and thus are rarely used to build large-scale applications;
- **Security:** having low levels of customizability/flexibility and high levels of abstraction, the users must accept that the system is bug-proof and will not originate vulnerabilities with its continuous use;
- **Vendor lock-in:** companies and organizations that focus too much on a specific low/no-code platform increase their dependency on the vendor and become more averse to switching platforms.

Ultimately, the no-code paradigm is a powerful approach with the potential to include domain-knowledgeable users in development. However, it should be taken cautiously depending on the platform's intended use and to avoid creating too much dependency on a single system. The best approach may be to address the no-code paradigm for specific system parts that need direct interaction with lay users, leaving more central aspects of the development to programmers.

Chapter 3

Prosebot Background

This chapter briefly describes previous contributions to the development of Prosebot in chronological order. The final section of the chapter gives an overview of the Prosebot system and attempts to place it in the architecture proposed by Reiter and Dale [68].

3.1 The GameRecapper System

Before even Prosebot had its actual name, Aires [1] started developing the GameRecapper system in collaboration with ZOS. GameRecapper was a template-based data-to-text system that generated Portuguese football match summaries from structured sports events and games data.

The generation module of GameRecapper was based on the GoalGetter [79] system’s module. To generate the summaries, the module processed input data retrieved from *zerozero.pt*’s API in the form of a JavaScript Object Notation (JSON) tree structure file constructed from the information of a specific game. Likewise, in order to produce sentences with some variation, domain data with additional information of the teams was processed, and a set of language templates was defined to be handled by the module. These templates were manually annotated accordingly to the typical structure of the news pieces summarizing the first 21 rounds of Liga NOS 2015/2016 – Introduction, Game Events, and Conclusion. Moreover, the generation module had yet to handle grammatical functions that ensured the correction of the content by number (singular or plural) and gender (masculine or feminine), linguistic functions to parse numeric data into words, and specific information of each team. An example summary generated by GameRecapper, and its translation, for the match information shown in Figure 3.1, can be seen below, as presented by Aires [1, p. 36]:

- **Original:**

“O Arouca bateu a Académica por 3-2, este sábado à tarde, no Estádio Municipal de Arouca.

Pedro Nuno, aos 11 minutos, abriu o ativo para a equipa visitante.

Aos 18 minutos, Jubal Júnior devolveu a igualdade ao encontro.

Aos 39 minutos, o Arouca confirmou a reviravolta no marcador, com Lucas Lima a ser o marcador de serviço.

O Arouca ampliou a vantagem aos 43 minutos, quando Artur Moreira colocou o resultado em 3-1, após passe de Adílson Tavares.

A equipa dos estudantes fixaria o resultado final em 3-2 com um golo de Gonçalo Paciência, depois de uma assistência de Rafa Soares, aos 62 minutos.

Com este resultado, o Arouca mantém o 5º lugar e passa a somar 44 pontos. Já a equipa de Filipe Gouveia continua com 23 pontos e mantém o 17º lugar.”

- **Translated:**

“Arouca beat Académica with a 3-2 victory, this saturday afternoon, in the Estádio Municipal de Arouca.

Pedro Nuno, at the 11th minute, opened the score for the away team.

At the 18th minute, Jubal Júnior equalized the score.

At the 39th minute, Arouca completed the comeback, with a goal by Lucas Lima.

Arouca extended their lead at the 43rd minute, when Artur Moreira put the scoresheet at 3-1, after an assist by Adílson Tavares. The students team fixed the final result in 3-2 with a goal by Gonçalo Paciência, after an assist by Rafa Soares, at the minute 62.

With this result, Arouca keeps the 5th position and has now 44 points. On the other side, Filipe Gouveia’s team continues with 23 points and remains at the 17th position.”

Two types of manual evaluation methods were applied to rate the quality of the generated summaries by GameRecapper. To test the **intelligibility and fluidity**, Aires [1] generated 44 match summaries for rounds 25 to 29 of Liga NOS 2015/2016. Then, he divided them into three surveys accordingly to the match final result so that each survey had a maximum number of different final results. Five evaluators for each survey were picked from the *zerozero.pt* newsroom to rate the two criteria on a scale from 1 to 5. The system achieved high intelligibility and fluidity scores on the generated summaries according to the results obtained. However, an inverse “*correlation between the fluidity of a text and the number of goals scored in a game as well as for the goal difference between the winning team and the losing team*” [1, p. 56] was noted. This is explained by GameRecapper’s lack of game events aggregation, generating extensive summaries in eventful games. Another survey was conducted to test users’ perception of **human-written versus GameRecapper generated** match summaries. Ten random matches were selected for the survey, where five were human-authored, and the remaining were generated. Forty-six evaluators answered the survey regarding the summaries’ “*accuracy/completeness and (...) readiness to be published online*” [1, p. 50], without knowing their provenience. The results obtained showed average completeness and readiness scores of 80.42% and 73.04%, respectively, for the GameRecapper’s summaries against 87.66% and 81.30% for the human-written ones. Aires concludes that “*despite the evaluators recognized the human-authored summaries as more complete/accurate and more ready to be uploaded online, the difference between them is considerably low*” [1, p. 57].



Figure 3.1: Example match information from www.zerozero.pt. Image from Aires [1, Fig. 3.1].

The development approach and the structure definition undertaken for the GameRecapper were the basis for the Prosebot application and can still be seen in its present state.

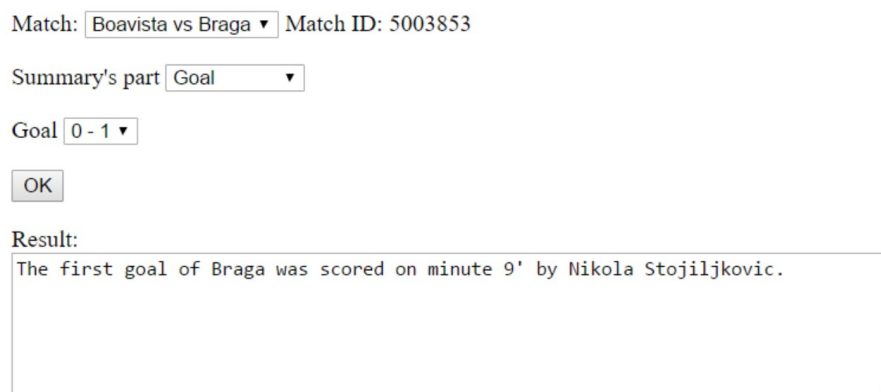
3.2 Statistical Language Models

Soares [75] carried out a different approach to the development of the data-to-text system. Instead of a template-based methodology, match summaries generation was achieved using statistical language models and learning methods.

Soares [75] started by extracting sample sentences from a dataset of summaries from the season 2015/2016 of the Italian championship and manually replaced some lexicons with tokens. Some patterns began to pop up, as in all the summaries firstly “*the journalist introduces the game, then he talks about the goals and sent-offs if there were any and finalize it with a conclusion*” [75, p. 31]. Thus, the sentences of the summaries were discriminated into four categories: introduction,

goals, sent-offs, and conclusion, and from that, the categories were even split into more specific subtypes. The SRILM¹ toolkit was used to train the model with the resulting sentences.

Moreover, as depicted in Figure 3.2, a user interface was designed to select categories of a match. Then, the system could automatically identify the subtype for the chosen category and generate a sentence by replacing the tokens with the match data extracted from *zerozero.pt*'s database through its API.



Match: Boavista vs Braga ▼ Match ID: 5003853

Summary's part: Goal ▼

Goal: 0 - 1 ▼

OK

Result:

The first goal of Braga was scored on minute 9' by Nikola Stojiljkovic.

Figure 3.2: User interface. Image from Soares [75, Fig. 3.3].

To evaluate the system, Soares [75] distributed surveys with rating questions for the intelligibility and completeness of the sentences generated on a scale from 1 to 5. The results achieved proved to be not as promising as expected, as the system could produce both high and poor quality content.

3.3 Prosebot

Prosebot² is a template-based data-to-text application that had its origin in the GameRecapper system. While GameRecapper was implemented in Python, Prosebot was adapted to PHP. At its core, Prosebot has the same objective to automatically generate football match summaries from structured data provided by *zerozero.pt*'s API.

Ribeiro [71] employed some enhancements to the Prosebot application. His contribution comprised adding new fields to the summaries, and a refactor of the structure, following the organization of a regular news piece: title, subtitle, abstract, and body. The body section resembles the one presented by GameRecapper, with an introduction, game events, and conclusion. Ribeiro [71] also stimulated the diversity of the generated contents by inserting some new match events, such as red cards, substitutions, and missed penalties. In addition, he initiated the development of multilingual support for Brazilian Portuguese, Spanish, and English, while maintaining the already existing support for European Portuguese.

¹<http://www.speech.sri.com/projects/srilm/>

²<https://www.zerozero.pt/prosebot.php>

The generation module of Prosebot follows the same methodology as GameRecapper’s, including the parse of JSON match data, domain data and language templates, and the use of grammatical and linguistic functions. Moreover, a new input field was added to accommodate the newly implemented multilingual support, that is, the language of the summaries to choose the correct templates’ files. A new feature was also added to the generation module to analyze and score the generated texts’ quality on a scale from 0 to 10, considering lexical diversity and sentence size. Prosebot’s architecture diagram of this period can be seen in Figure 3.3.

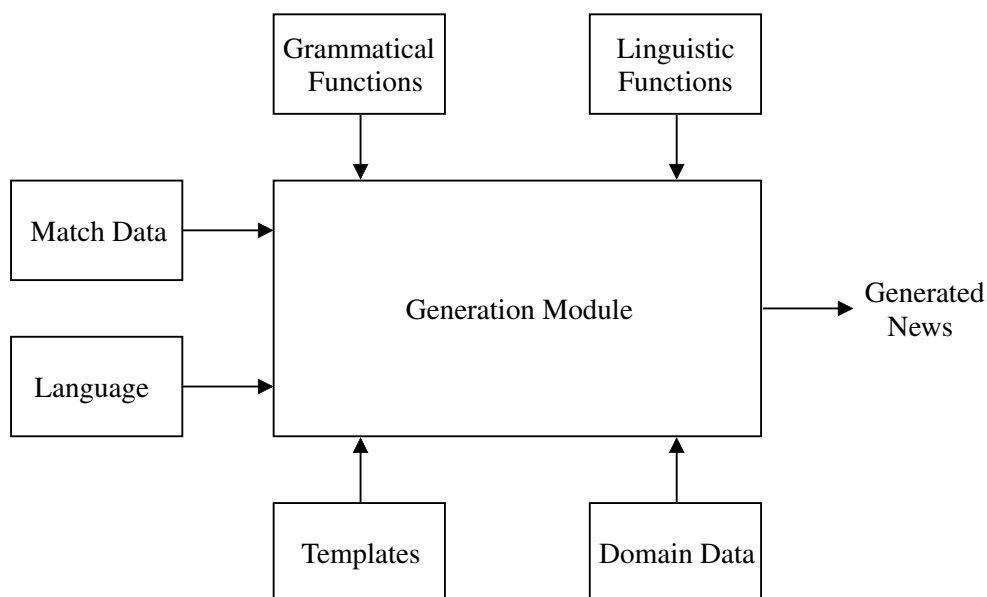


Figure 3.3: Prosebot architecture. Adapted from Ribeiro [71, Fig. 3.2].

3.4 Evaluation Metrics

The main objective of Correia’s [14] work was to develop a system to evaluate the quality of computer-generated and human-written texts using evaluation metrics. A literature review on the state of the art of NLG evaluation methods was conducted, leading to the creation of a metrics system queried with a developed API and composed of three distinct modules: one for scoring the computer-generated reports according to some automatic metrics, one to showcase textual features and scores according to readability indicators and the third one for entity recognition and Part-Of-Speech tagging (token classification).

Another accomplishment was the development of an interface to facilitate the automatic creation of football match reports by journalists. The interface allowed to pick match events and obtain initial versions of Prosebot’s generated text that the user could then edit and improve until it reaches a state when it is ready to be published. Figure 3.4 shows a view of the user interface after picking a match.

The image shows a web interface for generating news articles from football match data. On the left, a sidebar displays match information: ID do jogo 6942190, Borussia Dortmund vs Schalke 04, 2020-05-16 14:30:00, 1. Bundesliga 19/20. It also lists 'Últimos 5 resultados', 'Classificação pré-jogo', 'Golos' (Erling Haaland, Raphaël Guerreiro, Thorgan Hazard, Raphaël Guerreiro), 'Cartões Vermelhos', and 'Jogadores Chave' (Raphaël Guerreiro, Julian Brandt). At the bottom of the sidebar is a 'CRIAR NOTÍCIA' button. The main area on the right is a form titled 'RELATÓRIO DE MÉTRICAS' with sections for 'Title' (Borussia Dortmund com controlo total), 'Subtitle' (Goleada da equipa da casa em chuva de golos), 'Introduction' (O Borussia Dortmund goleou Schalke 04 no Sábado, 4-0. A equipa de Dortmund marcou por Raphael Guerreiro 2x, Erling Haaland e Thorgan Hazard.), and 'Body' (O Borussia Dortmund cilindrou o Schalke 04 numa vitória, 4-0, em jogo referente à jornada 26.).

Figure 3.4: View after inserting the match ID and loading the event information. Image from Correia [14, Fig. 6.3].

3.5 Community-Based Platform and Post-Editing

Until this point, Prosebot had only been used and tested internally by ZOS co-workers. Fernandes [29] had the task of publishing the application and evaluating it with the target audience: *zerozero.pt* users.

The project’s main goal was to speed up the writing process and the number of news generated for the various matches happening every day that are stored in ZOS’s database. The launching of Prosebot followed a community-based approach so that *zerozero.pt*’s advanced users could use it to generate summaries for a broader range of football matches, from amateur to professional leagues, for every age group of the athletes. This increased the number of collaborators in the writing process and the number of generated summaries. Besides, journalists could still use Prosebot to create initial versions of the articles and post-edit them. By spending less time analyzing sports data and statistics, journalists can focus more on creativity and increase their writings’ quality, fluidity, and diversity.

A survey was conducted with journalists from inside and outside *zerozero.pt*’s newsroom, addressing their opinion on the platform and the impact of this type of technology in the future of their jobs. While the feedback was majorly positive from *zerozero.pt*’s journalists, the acceptance by the outside ones was not that great, as they believed the introduction of such technology in the press world would result in journalists losing their jobs, leading to a decrease in the quality of the content created. Fernandes [29, p. 59] subscribes to their opinion and states that “*it is fundamental*

that they are included in the development (...) so that we can better understand how to use these tools to their advantage”.

Finally, some improvements to Prosebot were made, such as bug fixes and the addition of new templates and information to the generated texts. Figure 3.5 depicts an example of a match summary generated by Prosebot, published on *zerozero.pt* with the corresponding disclaimer referencing Prosebot and the editor of the summary.

3.6 Placement on the Typical NLG Architecture

The previously declared contributions led Prosebot to become a complete Natural Language Generation system that covers all three stages of the typical architecture presented by Reiter and Dale [68]: text planning, sentence planning, and linguistic realisation.

During **content determination** and **discourse planning**, Prosebot fetches and filters specific fields from the input data and uses template files to order the messages communicated in the text. In the Prosebot system, each template file corresponds to a section of the complete summary. Then, the generation module filters the templates that fulfill the conditions and traces a path in the template file for the sentences to be written and aggregated. This task is known as **sentence aggregation**.

After choosing which templates to use, the generation module produces **lexicalization** and **referring expression generation** by replacing the tokens with values and variable expressions for referencing entities and using cached memory to handle variability. Moreover, Prosebot is composed of several linguistic functions to translate numeric data into words.

Lastly, the system uses some grammatical functions to ensure agreement in number (singular or plural) and gender (masculine or feminine), include phrase connectors, and produce grammatically correct summaries. Figure 3.6 shows the Prosebot generation process of part of the introduction section of a made-up match summary.

VITÓRIA TANGENCIAL NO TERRENO DA EQUIPA DE FILIPE MARTINS

AF Lisboa Jun.B Honra: Santa Iria bateu UDR Santa Maria

Esta síntese foi criada automaticamente a partir de um [algoritmo](#) idealizado pelo zerozero.pt com alterações feitas pelo colaborador [Paulo_Lola](#)

2022/02/09 23:28

0



Em jogo referente à décima sétima jornada, o Santa Iria triunfou sobre o UDR Santa Maria, 1-0, no domingo. À chegada desta jornada, a equipa de Santa Iria de Azoia chegava de uma derrota, e a equipa de Odivelas vinha de uma vitória.

A primeira parte chegou ao fim sem golos.

[Rafael Mouralinho](#) marcou o golo da vitória no sexagésimo minuto de jogo.

Depois deste resultado o UDR Santa Maria ocupa o sexto lugar na [classificação geral](#), 29 pontos, com o Santa Iria a encontrar-se na décima sétima posição, 12 pontos.

Quanto à próxima jornada, a equipa de Santa Iria de Azoia [joga em casa do Oriental](#).

Por sua vez, a equipa de Odivelas [joga em casa frente ao GS Loures](#).



Esta síntese foi criada automaticamente a partir de um [algoritmo](#) idealizado pelo zerozero.pt com alterações feitas pelo colaborador [Paulo_Lola](#)

Achei relevante



Achei que tem qualidade



Figure 3.5: Match summary generated by Prosebot and published on [zerozero.pt](#). Image from <https://www.zerozero.pt/news.php?id=352050> (accessed Feb. 10, 2022).

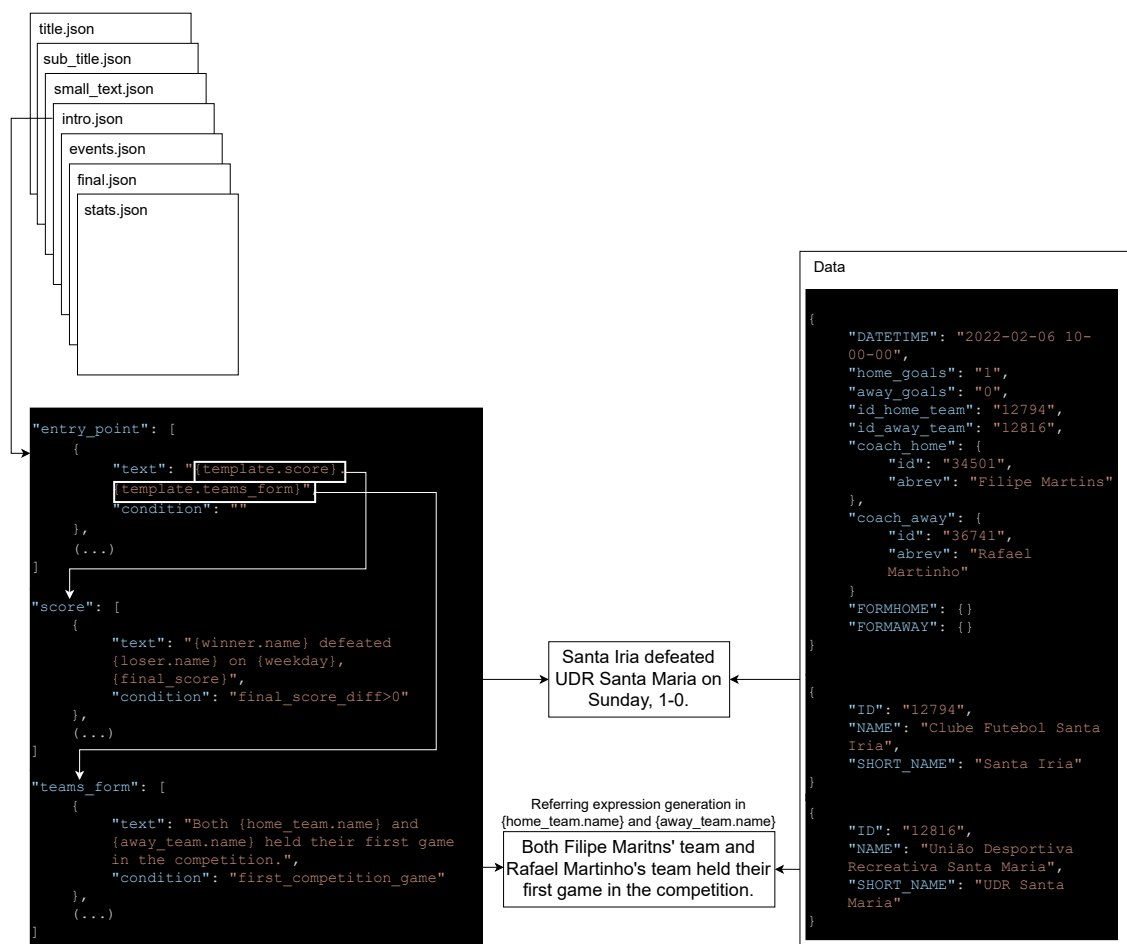


Figure 3.6: Prosebot generation process example.

Chapter 4

Prosebot Generator

This chapter describes improvements to the Prosebot generation module and the multilingual support refinement. It also presents the development process of a templates validation algorithm with a methodology to meet future integration in a templates management platform following the no-code paradigm.

4.1 Multilingual Support and Improvements

As mentioned before, one of the project's goals was to improve the multilingual functionality of Prosebot to generate production-quality content in every supported language: Brazilian Portuguese, English, European Portuguese, and Spanish. In collaboration with journalists from *zerzero.pt*'s newsroom, the templates of each language were continuously and cyclically revised, rewritten, and tested through summaries generation with Prosebot. From this process, some opportunities for improvement arose. The following subsections describe in detail the most significant enhancements employed in the generation module, divided by application areas.

4.1.1 Templates writing correction

The Prosebot templates in each language originated from the European Portuguese version. Thus, some writing issues were found and fixed regarding missing translations, wrongly-placed connector forms, unused or wrongly-placed template keys, use of undefined variables in conditions, and not closed hyperlink tags (e.g., `<a>`). The development of the templates validation algorithm described in Section 4.2 proved to be very useful for finding and fixing these syntactic errors. Moreover, new templates were written from scratch in specific languages, beyond simple translations. Lastly, each template file was cleaned from encoding issues that introduced special Unicode character forms (e.g., `\u003C`) and HTML tags in the text of the templates.

4.1.2 Language specific expressions

New expressions and ways of referring to teams were included for the Brazilian Portuguese, such as, “equipe do técnico {*coach_name*}”, “equipe comandada por {*coach_name*}”, “time do técnico {*coach_name*}”, and “time treinado por {*coach_name*}”. A critical aspect of this process was the gender of each expression. Before, there were only feminine forms, but with the inclusion of “o time”, the system should now also consider masculine forms.

In addition, versions of the competitions’ and teams’ names for each language were added. This was particularly relevant in national teams, as while clubs tend to keep their name independently of the language, national teams are more prone to change. For example, the Sweden national team name: *Suécia* (Portuguese), *Suecia* (Spanish), and *Sweden* (English).

4.1.3 Grammatical and linguistic functions

As stated before, Prosebot includes grammatical and linguistic functions to handle the correct production of varied text content. In order to ensure agreement in gender, feminine versions of the players’ positions in the field were added, and the corresponding gender selector was changed from hard-coded masculine to being functionally handled. Due to the lack of data mentioning players’ gender, if it is a masculine competition match, the players are assigned the masculine gender. Otherwise, they are assigned the feminine gender.

Furthermore, a neutral form was added to each connector’s existing feminine and masculine forms, and a new linguistic function was implemented. In contrast to the already existing functions that produce fully written numbers, the new function, *ordinal_num*, can generate numeric ordinal forms, for example, “1^a or 1^o”.

4.1.4 General bugs correction

Through Prosebot utilization, several generic bugs were found and fixed. The following list enumerates some:

1. In many matches of the league or group phase of a competition, the generated summaries did not mention the teams’ position in the classification;
2. In matches of a knockout competition, the final paragraph of the summary showed a fragmented sentence which was wrongly attempting to mention the next game of the team that was knocked out;
3. In specific cases, the fixture of the match was printed as the number zero instead of the correct number;
4. When a player from the bench received a red card, the summary would sometimes tell that the team was left playing with minus one player. This error was fixed by adding a condition to verify if the player entered the field;

5. Finally, as ZOS intended to use Prosebot in each of its web domains for different countries, the hyperlinks of the matches, competitions, players, and teams needed to be changed according to the language.

4.1.5 New features

The previous subsections described changes, improvements, and corrections to already implemented features. Besides those, some new variables and conditions were introduced to Prosebot to expand its functionality and provide new information in the generated texts.

A new *decisive_player* entity variable was created to refer to players who scored a decisive goal during the match, a goal after the eighty-fifth minute that granted the team a victory or a tie. To ensure Prosebot uses the entity correctly, a new variable was defined, called *has_decisive_player*. Every template that uses *decisive_player* in the text should include *has_decisive_player* in the condition.

When analysing generated summaries of knockout competitions, which could be in multiple hands, it missed mentioning which team went through to the next round. Until then, summaries only expressed the final result of a single match and had no reference to past games, despite the system receiving such data from the ZOS database. So, for example, in a two-handed knockout round, if a team won the match but lost the round, the summary would still express positive feelings by focusing on the victory. The variables in Table 4.1 were introduced to fill this gap, all considering a previous match between the same teams for the same competition but in the other team's stadium. The created variables made it possible to write new template sentences, like “despite the defeat, {*knockouts_winner.name*} knocks out {*knockouts_loser.name*}”.

Table 4.1: Previous match variables.

Variable	Description
<i>prev_match_final_score_diff</i>	The difference of goals in the final score of the previous match
<i>has_previous_match</i>	Whether there was a previous match between the two teams
<i>has_two_hands</i>	Whether it is a two-handed knockout round
<i>loser_moves_on</i>	Whether the team that lost the current match, still managed to win the knockout round
<i>winner_moves_on</i>	Whether the team that won the current match also won the knockout round
<i>prev_match_score</i>	The final score of the previous match in an NxN format
<i>prev_match_winner</i>	The winning team of the previous match
<i>prev_match_loser</i>	The losing team of the previous match
<i>TeamData.prev_match_goals</i>	The number of goals scored by the team in the previous match

One concern expressed by the journalists of *zerozero.pt* newsroom was the lack of a token variable they could use on templates to express the position in the field of a player, which was

solved with the inclusion of position as a new player property. Another problem was the continuous misuse of definite articles before a player's name in the Portuguese and Spanish texts. When referencing a player, Prosebot manages to increase sentence fluidity by switching between calling a player by the name or by the position in the field, for example, “João Cancelo” or “o defesa (the defender)”. In order to stop using definite articles when mentioning the player's name while keeping them for the position, a neutral gender form was assigned to the name part. Furthermore, a separate *PlayerData.name_gender* variable was introduced to accommodate situations where the writer wants to enforce the use of definite articles. The example presented below shows the difference between using *player.name* and *player.name_gender*, in Portuguese.

- “{o%player.name} {player.name} foi expulso ao ver o cartao vermelho.”
Produces:
(1) “João Cancelo foi expulso ao ver o cartão vermelho.”
(2) “o defesa foi expulso ao ver o cartão vermelho.”
- “{o%player.name_gender} {player.name} foi expulso ao ver o cartao vermelho.”
Produces:
(1) “o João Cancelo foi expulso ao ver o cartão vermelho.”

Keeping in mind ZOS's web domains for other countries, a new feature was considered to assign time zones to languages, thus converting Portugal's time zone of the sample data to the time zone of the destination country in a mutable and programmatic way. Moreover, two new standards were defined for every language. The nicknames of the teams and players started being written in italics, and matches' scores were changed from *N-N* to *NxN* format, where *N* is the number of goals.

4.1.6 Italian language

During the development phase, there was the opportunity to expand the multilingual support by adding a new language from scratch: Italian. Following the same approach as other languages, the European Portuguese template files were translated into Italian. New connectors needed to be defined from this process, and new versions of the linguistic functions were implemented so that Prosebot could correctly generate written cardinal and ordinal Italian numbers. Table 4.2 shows the connectors defined and their appropriate form according to gender and number.

Table 4.2: Italian regular connectors.

Name	Singular		Neutral	Plural	
	Masculine	Feminine		Masculine	Feminine
a	al	alla	a	ai	alle
di	del	della	di	dei	delle
da	dal	dalla	da	dai	dalle
in	nel	nella	in	nei	nelle

Following the grammatical rules of Italian, the “il” and “e” connectors had to be handled separately, as their variations depended not just on the gender and number but also the first characters of the word after them. Tables 4.3 and 4.4 present the correct form of the “il” and “e” connectors according to each scenario.

Table 4.3: Variations of “il”.

“il” variations	Scenario	Variation
Singular	Beginning in vowel	l’
	Feminine beginning in consonant	la
	Masculine beginning in “s” or “z” + consonant	lo
	Other masculine words	il
Plural	Feminine	le
	Masculine beginning in vowel or in “s” or “z” + consonant	gli
	Other masculine words	i

Table 4.4: Variations of “e”.

Scenario	Variation
Beginning in “i” or “e”	ed
Other words	e

In the same way, composite connectors were implemented after, comprising a set of connectors followed by an “il” variation. These connectors included “a, da, di, e, in, su” plus “il”. The correct variations’ construction handled by the generator can be seen in Table 4.5.

Table 4.5: Italian composite connectors.

Name	Connector	il	lo	l’	la	i	gli	le
a_il	a	al	allo	all’	alla	ai	agli	alle
da_il	da	dal	dallo	dall’	dalla	dai	dagli	dalle
di_il	di	del	dello	dell’	della	dei	degli	delle
e_il	e	ed il	e lo	e l’	e la	ed i	e gli	e le
in_il	in	nel	nello	nell’	nella	nei	negli	nelle
su_il	su	sul	sullo	sull’	sulla	sui	sugli	sulle

Lastly, new referring expressions were included for teams, such as “squadra di {coach_name}”, “squadra {di_il%city} {city}”, and “nazionale {di_il%country} {country}”, and week’s days and player positions in the field were translated. An example summary generated in Italian is presented below for a match between the national teams of Portugal and Azerbaijan for the World Cup Qualifiers (UEFA) 2022:

Title: “Il Portogallo ha battuto l’Azerbaijan 1x0”

Sub-title: “Vittoria di misura degli ospiti”

Small text: *“Il Portogallo ha vinto l’Azerbaigian in una vittoria serrata mercoledì, 1x0. Tra i titolari è stato Cristiano Ronaldo.”*

Summary: *“Il Portogallo ha vinto l’Azerbaigian in una vittoria di misura, mercoledì, 1x0, nella prima settimana di gare della Qualificazioni ai mondiali (UEFA). Sia i "portugueses", che la nazionale dell’Azerbaigian giocavano la sua prima sfida nella competizione. Cristiano Ronaldo è stato titolare nella partita.*

Al trentaseiesimo minuto della partita, Maksim Medvedev ha avuto la sfortuna di segnare nella propria porta.

Dopo questo risultato il Portogallo si incontra nel secondo posto in classifica generale, 17 punti, trovandosi l’Azerbaigian al quinto posto, 1 punto. Quanto alle prossime sfide della competizione, la squadra di Fernando Santos si trasferisce a casa della Serbia, mentre la squadra di Gianni De Biasi accoglie la Serbia.”

And the corresponding translation:

Title: *“Portugal beat Azerbaijan 1x0”*

Sub-title: *“A narrow victory for the guests”*

Small text: *“Portugal defeated Azerbaijan in a tight victory on Wednesday, 1x0. Among the starters was Cristiano Ronaldo.”*

Summary: *“Portugal overcame Azerbaijan in a narrow victory on Wednesday, 1x0, in the first week of World Cup Qualifiers (UEFA) matches. Both the "portugueses" and the Azerbaijan national team were playing their first challenge in the competition. Cristiano Ronaldo was a starter in the match.*

In the 36th minute of the match, Maksim Medvedev had the misfortune to score in his own goal.

After this result, Portugal is in second place in the overall standings, 17 points, with Azerbaijan in fifth place, 1 point. As for the next challenges in the competition, Fernando Santos’ team travels to the home of Serbia, while Gianni De Biasi’s team welcomes Serbia.”

4.2 Templates Validation Algorithm

This section describes an algorithm developed to validate the writing correction of Prosebot’s templates. Besides being a valuable resource to help the users validate and fix common mistakes, it is also a key feature to follow the no-code paradigm in the management and edition of templates by enforcing and automating writing rules.

4.2.1 Search for writing patterns

Since the objective was to automate the validation of syntactic rules for writing templates, the already existing templates' JSON files were analyzed to find patterns. Initially, Prosebot had support for four languages: Brazilian Portuguese, English, European Portuguese, and Spanish, with seven template JSON files each. Thus, twenty-eight files were taken into account. Later, seven new files were introduced with the addition of the Italian language and keeping the same writing rules. A template JSON file is a set of template keys composed of arrays of elements in a *text-condition* pair format. Listing 4.1 presents an example of two template keys: *best_player* and *best_player_contribution*.

```
1 "best_player": [  
2   {  
3     "text": "was the star {template.best_player_contribution}",  
4     "condition": "has_best_player"  
5   }  
6 ],  
7 "best_player_contribution": [  
8   {  
9     "text": "with {best_player.goals} goals",  
10    "condition": "has_best_player"  
11  }  
12 ]
```

Listing 4.1: Example template keys.

The patterns extracted from analysing the JSON files were the following:

- **Tree structure vs. Sparse keys:**

Five (title, sub_title, small_text, intro, final) out of the seven template files had a tree structure where each template key node redirected to template key leaves. In the example Listing 4.1, the *{template.best_player_contribution}* token in *best_player* redirects to the *best_player_contribution* key. The remaining two files (events, stats) did not have this hierarchic structure. Instead, they present independent key declarations.

- **Text-Condition pairs:**

Each template is depicted as a *text-condition* pair, in which *text* is the content, or rather a set of regular strings and tokens, and *condition* is the boolean expression that should be fulfilled so that *text* can be used.

- **Regular string:**

A regular string is any set of characters besides tokens, opening and closing brackets, that form words, expressions, or sentences.

- **Token declaration:**

A token declaration is any chunk of text that begins in an opening bracket and ends in a

closing bracket. Tokens are the parts of the *text* that variable values will replace during the generation. Table 4.6 shows the different writing patterns of tokens and their corresponding use cases.

Table 4.6: Token declaration patterns.

Pattern	Use case	Example
template.template_name	Traverse templates keys tree structure	{template.normal_title}
template.template_name%entity	Traverse templates keys tree structure and pass <i>entity</i> as argument	{template.team%away_team}
entity.property	Property of an entity	{loser.name}
property	Property of the main entity (match)	{stadium}
connector%token	Number and gender adaptable expressions	{de%scorer}
link_name string ^a	Hyperlink on the given string	{edition_link}table

^aLater, an “@” was added before the *link_name* so that the validation algorithm could identify this particular pattern (e.g. {*@edition_link*}table)

• Connector declaration:

A connector declaration is the part of a token that appears before a “%” signal (except on a *template.template_name%entity* pattern). It allows the expression to vary according to the number (singular or plural) and gender (feminine, masculine or neutral) of the entity residing after the “%” signal. A connector declaration in the templates is ultimately linked to the grammatical and linguistic functions of the system. Table 4.7 depicts the different writing patterns of connectors and their use cases.

Table 4.7: Connector declaration patterns.

Pattern	Use case	Example
connector_name	Expressions connector	{de%competition.name}
s:[string] p:string	Singular and plural forms	{s:amplioulp:ampliaram%team.name} {s:lp:m%#arg.name}
f:string m:string ^a	Feminine and masculine forms	{f:decisivalm:decisivo%decisive_player.name}

^aLater, an optional neutral form was added in *n:string* format

The search for patterns in the templates led to the definition of rules for the algorithm to validate. Each pattern was directly translated into a rule, and others were included later as the algorithm was improved. Table 4.8 summarises the rules used by the templates validation algorithm.

4.2.2 JSON files validation

Following the previously described patterns and rules, a JSON file validation was initially implemented with an input checkbox to guarantee the distinction between tree-structured and sparse keys files. When checked, the validation algorithm would traverse the tree of keys in-depth, leaving the isolated keys out of the validation. These were pushed into an unused templates list, resulting in a warning as depicted in Listing 4.2.

```
1 Warning: "$unused_template" defined but never used
```

Listing 4.2: Unused template warning.

On the other hand, if the checkbox were left unchecked, the validation would consider every key, and no warnings would be printed. The algorithm would after parse every array element inside a template key. It would verify first if the *text* and *condition* keys were set and then validate each separately.

4.2.3 Text content validation

The *text* side of a template *text-condition* pair was validated using regular expression techniques to catch wrong token constructions. Initially, the validation verifies whether the link declarations were correctly closed by `` tags. Then, the algorithm splits the *text* string through *{token}* chunks into an array of regular strings and tokens and validates each separately, following the patterns described before. The regular strings validation checks the existence of opening or closing brackets (`{}`) and throws an error in a positive case. In turn, token declarations verification must be more thorough as they are the core aspect and the reason for developing a validator. The validation algorithm has the critical task of ensuring that the token writing standards mentioned above are met.

To better understand and analyze the token declaration patterns and make it easier to develop validation functions that follow a modular approach, a Non-Deterministic Finite Automaton diagram (NFA) was designed and converted to the corresponding grammar definition. Due to the great inherent complexity of the production and visualization of such a diagram as a whole, this was divided into simpler diagram parts to improve readability and comprehension, both during production and for the reader. Figure 4.1 shows the NFA diagram in the highest level of abstraction, with the distinction between regular strings and token declarations, being *S* both the initial and the goal state.

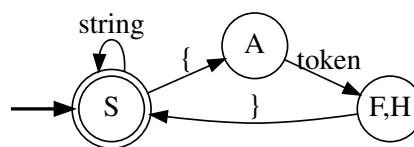


Figure 4.1: NFA diagram - Regular strings and token declarations.

Figure 4.2 depicts the NFA diagram that dives inside the token declaration, where the states' names mark it as a continuation of the higher level NFA presented before. Each path in the NFA represents one of the token's patterns.

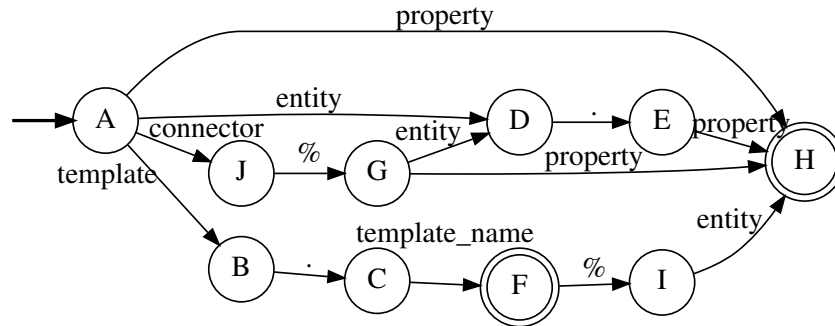


Figure 4.2: NFA diagram - Token declaration.

In the same way, Figure 4.3 goes through the connector declaration in-depth, showing the path for the number, gender, and basic connector name formats. The connector declaration is enclosed in the transition from state A to state J.

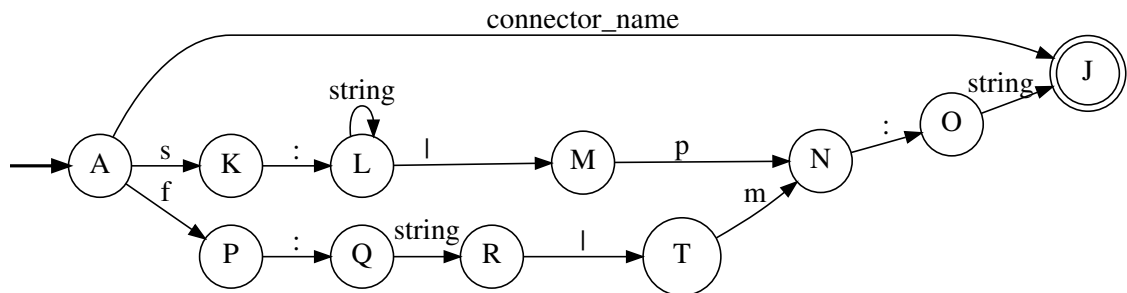


Figure 4.3: NFA diagram - Connector declaration.

Later, the NFA diagrams were converted into a single Context-Free Grammar (CFG) describing each writing production of *text*. After simplification, the established grammar is presented in Listing 4.3, accompanied by an example *text* production. The validation algorithm ensures that each production rule is obeyed.

```

1 S -> string S | { A | epsilon
2 A -> "template".template_name F | entity.property} S | property} S |
   connector_name% G | s: L | f:string vertical_bar m:string% G
3 F -> %entity} S | } S
4 G -> property} S | entity.property} S
5 L -> string L | vertical_bar p:string% G
6 vertical_bar = |
7
8 /* Example: "{o%scorer} {scorer.name} fez o golo da partida aos {template.time
   }"
9 S -> { A
10    -> {connector_name% G
11    -> {o%property} S
12    -> {o%scorer} { A
13    -> {o%scorer} {entity.property} S
14    -> {o%scorer} {scorer.name} string S
15    -> {o%scorer} {scorer.name} fez o golo da partida aos { A
16    -> {o%scorer} {scorer.name} fez o golo da partida aos {"template".
       template_name F
17    -> {o%scorer} {scorer.name} fez o golo da partida aos {template.time} */

```

Listing 4.3: CFG for *text* productions.

4.2.4 Condition validation

The validation of *condition* is the parsing of a string representing a boolean expression. Originally, only simple expressions were considered, including equations, inequations, conjunctions, disjunctions, and their combinations, as regular expressions verification could solve them:

- **Equation:**

left_hand == right_hand

left_hand != right_hand

- **Inequation:**

left_hand >= right_hand

left_hand > right_hand

left_hand <= right_hand

left_hand < right_hand

- **Conjunction:**

left_hand && right_hand

- **Disjunction:**

left_hand || right_hand

Following the same procedure, an NFA diagram of the simple conditions was built to clarify the writing of a condition, as shown in Figure 4.4.

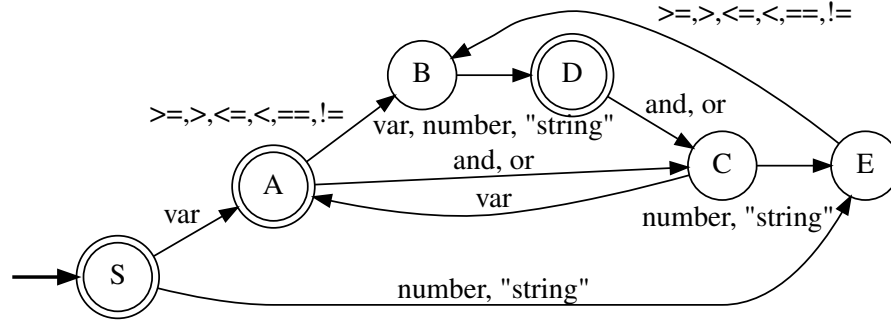


Figure 4.4: NFA diagram - Condition (and = &&; or = ||; var = variable name).

After simple conditions validation was operational, the opportunity to start analyzing more complex boolean expressions emerged. Thus, parentheses were introduced, allowing for composite expressions. The Context-Free Grammar, which depicts the *condition* writing rules, is delivered in Listing 4.4, accompanied by an example *condition* production. Again, the validation algorithm ensures that each rule is fulfilled.

```

1 S -> C | left_parentheses S right_parentheses A | epsilon
2 A -> E | D
3 B -> variable D | !variable D | number D | "string" D
4 C -> variable A | !variable A | number E | "string" E
5 D -> and C | or C | epsilon
6 E -> >= B | > B | <= B | < B | == B | != B
7 left_parentheses = (
8 right_parentheses = )
9
10 /* Example for "extra_time != "PEN" ":
11 S -> C
12   -> variable A
13   -> extra_time E
14   -> extra_time != B
15   -> extra_time != "string" D
16   -> extra_time != "PEN" */

```

Listing 4.4: CFG for *condition* productions.

4.2.5 Vocabulary definition

Up until this point, only syntactic rules were assessed. Another feature was introduced to the validation algorithm that considered a dictionary of valid variable names, the *vars.json* file. Any variable name used on the template files and not included in the dictionary would cast an undefined variable error.

The dictionary is a JSON file composed of properties, entities, and connectors. The properties are divided into two types: properties for *text* and properties for *condition*. The entities object encloses the valid entities variables' names and corresponding inner properties list. As for the connectors object, it was composed of all connectors names divided by language. Finally, the variables' names had to follow some rules, such as do not have white spaces, do not start with numbers or special characters, except for the particular case of *#arg* (used to pass an entity as an argument of the next template in the hierarchy).

4.2.6 Execution methods, settings and actions

As mentioned before, the validation algorithm represents a powerful tool in the context of the no-code paradigm and speeds up the fix of writing issues. In order to take full advantage of the algorithm's capabilities, some new ways of executing it were thought of that could meet further user needs. The three methods of execution that were made available were:

1. **Full validation** - Validate templates' syntactic rules and check if the entities used are defined in the dictionary of variables' names;
2. **No entities definition check** - Validate syntactic rules without checking if the entities used are defined;
3. **Get entities names** - Validate syntactic rules, do not check if the entities used are defined and print the names of the entities separated by their type: property, entity, property of an entity, and connector (divided by language).

In addition to the different methods of execution, some extra settings with defaults were made available to tune the validation. These include:

1. **Context** switch - The contexts available are a core complete *Football* domain and a *Weather* dummy example.
2. **Languages** switch - The available languages are Brazilian Portuguese, English, European Portuguese and Spanish. This setting is needed due to the division of connectors' names by language.
3. **Hierarchy** checkbox. When checked, the algorithm traverses the tree hierarchy, leaves out of the validation the templates which are not mentioned by parent nodes and prints warnings if a template key is defined but not traversed. When not checked, the algorithm validates all template nodes independently.

Finally, the available actions make use of the methods and settings presented before and perform the validation accordingly:

1. **Validate input file** - Choose a template JSON file to validate.

2. **Text-Condition** pair - Validate a single *text* and/or *condition*. It does not check if a *template_name* is defined.
3. **Validate all** - Validate all templates of a chosen language.
4. **Validate and get status** - Validate a *text-condition* pair and get status. Status options include [1, “Success”] and [0, \$Error], where \$Error is in one of the formats presented in Subsection 4.2.7. This action, in particular, is not available in the interface, only through the definition of its function.
5. **Generate dictionary** - Later in the development, another action was added to generate automatically the *vars.json* dictionary file used on the validation. This action will be described more in detail in Chapter 5.

Figure 4.5 shows the user interface for validating Prosebot’s templates, with distinct field sets for the execution methods, settings, and actions.

[Back](#)

Validate Templates

Execution Settings

Football
Português
Hierarchy

Execution Methods

Full
Without checking entities definition
Get entities

Actions

Validate input file:

Escolher ficheiro
nenhum ficheiro selecionado
Validate

Validate text-condition pair:

Text
Condition
Validate

Validate all template files for context and language:
Validate

Generate dictionary for context:
Generate

Figure 4.5: User interface for templates validation

4.2.7 Error handling

Whenever a typo is found in a template that would cause a mistake in Prosebot’s writing, the algorithm casts an exception and stops execution. The exceptions thrown include a variable \$path

with the place in the JSON tree where the error occurred, an explanatory message, and, depending on the setting used, the name of the template file. They obey one of the formats depicted in Listing 4.5.

```

1 Error: [$path] - Missing "text" key on element
2 Error: [$path] - Missing "condition" key on element
3 Error: [$path] - Missing "$bracket" on chunk "$chunk"
4 Error: [$path] - Expected "$expected", found "$found"
5 Error: [$path] - Wrong token construction: "{$token}", $explanation
6 Error: [$path] - Wrong construction: "{$construct}", $explanation
7 Error: [$path] - Wrong condition construction: "{$condition}", $explanation
8 Error: [$path] - Invalid variable name: "$variable"
9 Error: [$path] - Wrong boolean expression construction: "$expression"
10 Error: [$path] - Wrong link construction: "$link", $explanation
11 JSON validation errors (errors in the intrinsic construction of the JSON file)

```

Listing 4.5: Error messages formats.

In some cases, it is not worth stopping execution and throwing an error since, despite being a typo, it will not compromise the text production. Instead, a warning message is printed, letting the validation keep going. The warning messages obey the formats presented in Listing 4.6.

```

1 Warning: Directory is empty
2 Warning: "$unused_key" defined but never used
3 Warning: $path - Invalid variable name: "$variable"
4 Warning: $path - Wrong token construction: "{$template}", template not defined

```

Listing 4.6: Warning messages formats.

“*Invalid variable name*” appears both in errors and warnings. The algorithm should cast an exception when the variable is an entity, property, property of an entity or connector name. However, the program should not stop its execution in the case of template names used as keys in the JSON hierarchy.

Table 4.8: Templates validation algorithm rules.

JSON level	Rule
Root	<ul style="list-style-type: none"> • JSON has to be valid, following JSON syntax rules; • A template key should have a valid variable name: do not have spaces, do not start with numbers or special characters (w); • A template key represents a template name; • A template key has an array of elements as corresponding value.
Element	<ul style="list-style-type: none"> • Each element must be a <i>text-condition</i> pair.
Text	<ul style="list-style-type: none"> • <i>text</i> can be just a string, including the empty string (“”); • <i>text</i> is a set of regular strings and tokens; • Regular strings in <i>text</i> cannot have “{” or “}”; • <i>text</i> can include tokens in <i>{token}</i> format.
Token	<ul style="list-style-type: none"> • A token should obey the formats presented in Table 4.6 and cannot be null; • Each entity, property, template name, and link name should have a valid variable name. In addition, the entity can also be <i>#arg</i>.
Connector	<ul style="list-style-type: none"> • A connector should obey the formats presented in Table 4.7 and cannot be null; • Each connector name should have a valid variable name; • Regular strings in connector declarations do not have validation.
Condition	<ul style="list-style-type: none"> • A condition must be a boolean expression or the empty string (= no condition); • A condition has equal numbers of opening “(” and closing parentheses “)”, and a closing parentheses “)” cannot be immediately followed by an opening parentheses “(”; • At each place of the condition, the number of appearances of closing parentheses “)” should always be less or equal to the number of appearances of opening parentheses “(”; • A condition cannot start or end with “&&, , >=, <=, <, >, == or !=” (symbols), and symbols cannot be immediately followed by other symbols; • Boolean expressions can be equations, inequations, or variables in <i>[[!]<i>entity</i>.]<i>property</i></i> format, linked by boolean operators “&&” (and) and “ ” (or).

Chapter 5

Open-Source Refactoring

5.1 The Prosebot System

The main objective of this work was to convert the existing Prosebot application into an open-source software project that could be freely used and expanded by the community with less effort. The generator's source code was restructured, and new components were included to meet the no-code paradigm and make it possible for lay users and programmers to interact with the system. The components architecture of the new Prosebot system is described in the following subsection.

5.1.1 Components

The Prosebot system was idealized to be composed of three main components and corresponding interfaces: Prosebot (generator), Prosebot Editor, and Templates Management Platform. The components diagram in Figure 5.1 shows an overview of the entire system.

Prosebot is the natural language generator described in the past chapters, with the corresponding code refactoring. Besides, a new API was developed to support the Templates Management Platform with templates validation and other supporting methods.

Prosebot Editor is an auxiliary component written in PHP that provides a copy of the template files and an API for their management. With this extra component, it is possible to preserve the templates stored in the Prosebot system while using the Templates Management Platform.

The **Templates Management Platform** is a React JS¹ platform that provides a user-friendly interface for helping with creating, editing, deleting, and validating templates.

5.2 Code Restructuring

As stated, the generator component suffered a refactoring process to fit the new Prosebot system. While reorganizing the source code, attention was always focused on future third-party utilization, and thus, it needed to be as comprehensible and prone to change as possible.

¹<https://reactjs.org>

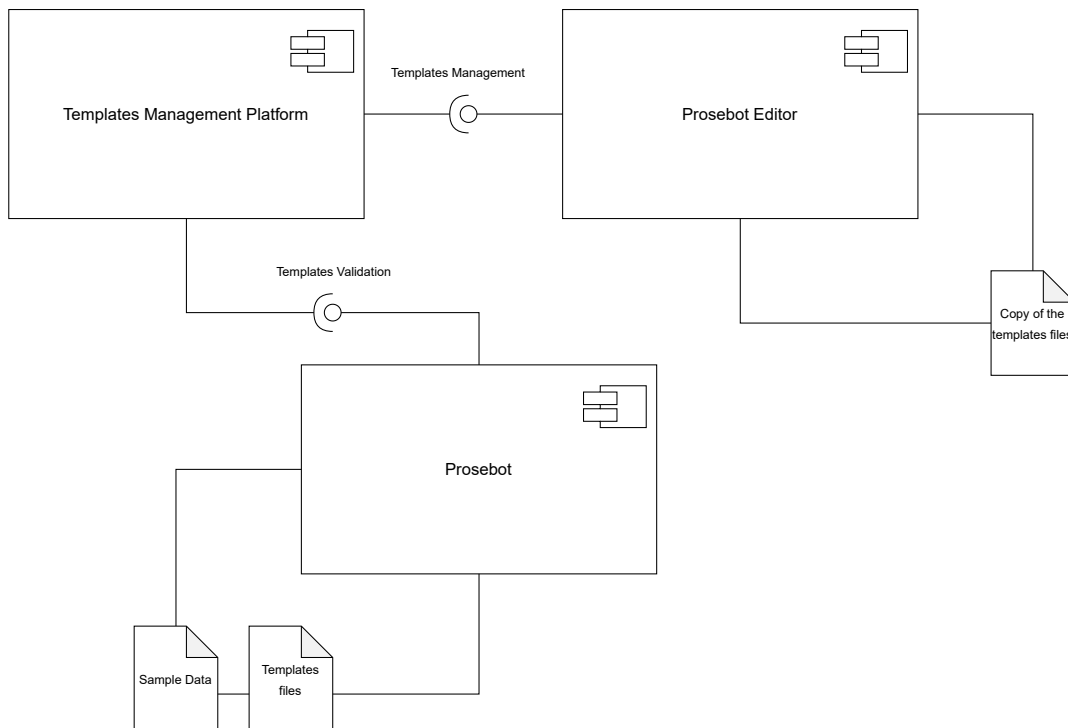


Figure 5.1: UML components diagram of the Prosebot system.

The first step was to restructure the entities' class files. Due to the inherent characteristics of PHP programming language and the support of dynamic variable declarations, the code found did not distinguish between class properties and methods, and many variables had public access without being necessary. In many files, it was even tough to identify the properties of a class. To solve this issue, a class attribute was created with the corresponding access modifier for each dynamic variable declaration. Then, a clear separation between properties and methods, and a further separation between *getters*, *setters*, and other functions, was undertaken. In addition, all source code was adequately documented according to the PHP standards. Prosebot's source code featured an *EntityData* class from which some entities' classes were sub-classes. In order to enforce normalization, the source code was adapted so that all classes representing entities extended the *EntityData* class, thus inheriting its properties and methods. These include *id*, *name*, *link*, *__toString()* to translate the entity into a reference expression, and *get_entity()* to get the value of an entity, amongst others.

A second task featured a reorganization of generic functions and conditions-related classes. All functions with generic purpose and global usability were passed to an already defined *Utils* class. All variables with a global scope, namely the lists of contexts, languages, time zones and classes of the main entities, were compiled in a list to avoid code repetition. Moreover, the *Properties* and the *Templates* classes were separated into different files to improve readability, and a *PropertiesManager* class was created to handle the construction of properties for templates' conditions.

The next step consisted in reorganizing the grammar files and so the grammar and linguistic functions of Prosebot. Initially, the methods of each child's grammar class (*GrammarBR*, *GrammarEN*, *GrammarES*, *GrammarPT*, and *GrammarIT*) were revised, and shared functions were modified and moved to the higher abstract *Grammar* class. Moreover, some domain-related functions were moved from the grammar classes to other scopes, such as the entities manager class. Then, the grammar and linguistic functions were reformulated to improve legibility. Before, both types were represented as function declarations whose return value would be used to write the text, as shown in Listing 5.1.

```

1 private static function de($gender, $number, $text) {
2     $result = "de";
3     if ($gender === NameGender::MALE) {
4         if ($number === NameNumber::PLURAL) {
5             $result = "de los";
6         }
7         else {
8             $result = "del";
9         }
10    }
11    if ($gender === NameGender::FEMALE) {
12        $result = "de la";
13        if ($number === NameNumber::PLURAL) {
14            $result .= "s";
15        }
16    }
17    return $result;
18 }
19
20 private static function a($gender, $number, $text){}
21 private static function el($gender, $number, $text){}
22 private static function cardinal($gender, $number, $text){}
23 private static function cardinal_fem($gender, $number, $text){}
24 public static function ordinal($gender, $number, $text){}
25 private static function ordinal_fem($gender, $number, $text){}

```

Listing 5.1: Example grammar and linguistic functions for the Spanish language, before restructuring.

During the restructuring, a list of connectors was defined, like the one in Listing 5.2 for the Spanish language. It depicted a clear distinction between grammar functions that depended on gender and number, other grammar functions that depended on more concrete scenarios, and linguistic functions to produce cardinal and ordinal numbers' forms. The generation module handles the *\$connectors* list and executes different operations according to the type of the assessed element. Each element has the connector's name as a key, which is the one written in the templates. The element's value can be either an array of connector forms sorted in *Singular Male*, *Singular*

Female, Neutral, Plural Male, Plural Female order or the name of the grammar function or the linguistic function to call. If the element is an array of connector forms, the generation module writes the correct number-gender form according to context. Otherwise, it calls the specified function and writes the returning value. This change even repaired the lack of some grammar functions in the different languages. As another example, Listing 5.3 shows the grammar functions of the Italian language that produce the correct forms for the connectors defined in Subsection 4.1.6.

```

1 private static $connectors = array(
2     // Name => [Singular Male, Singular Female, Neutral, Plural Male, Plural
3     //           Female]
4     "a" => ["al", "a", "a", "a los", "a las"],
5     "de" => ["del", "de la", "de", "de los", "de las"],
6     "el" => ["el", "la", "", "los", "las"],
7     // Name => linguistic_function
8     "cardinal" => "cardinal",
9     "cardinal_fem" => "cardinal_fem",
10    "ordinal" => "ordinal",
11    "ordinal_fem" => "ordinal_fem",
12    "ordinal_fem_num" => "ordinal_fem_num",
13    "ordinal_num" => "ordinal_num"
14 );

```

Listing 5.2: Example list of connectors for the Spanish language, after restructuring.

```

1 private static $connectors = array(
2     (...)
3     // Name => grammar_function
4     "il" => "il",
5     "e" => "e",
6     "a_il" => "a_il",
7     "da_il" => "da_il",
8     "di_il" => "di_il",
9     "in_il" => "in_il",
10    "su_il" => "su_il",
11    (...)
12 );

```

Listing 5.3: Example grammar functions for the Italian language, after restructuring.

5.2.1 Context generalization

One of the major concerns during source code refactoring was the possibility of opening the Prosebot generator to more domains than just football matches summaries generation. The basic idea was to reformulate the source code so that domain-specific elements were separated from the

core linguistic, grammar and management features. New abstractions were created to reinforce classes' coding rules and guide the future programmer. The code of every entity, manager, fetcher, and grammar class considered generic was abstracted into parent classes, and new extending ones were defined, enclosing the remaining domain-specific code.

Before restructuring, the source code files of Prosebot were all located at the project's root, except for a grammars directory and a templates directory. Despite not being ideal, this disposition was acceptable when dealing with just one domain. For multiple domains, it becomes essential to re-dispose the files in a directory tree structure to show a better visual presentation. Figure 5.2 shows a high-level representation after generalization of the directories tree of the core files of Prosebot, and Figure 5.3 shows the tree inside of a specific *<context>*.



Figure 5.2: Directories tree of the core files.

Analysing each core directory and file individually, starting with the **index.php** files, the one located at the root was changed so the user could choose between the available domains and go to the corresponding generator's page. The **validator** and **grammars** directories include the code of the templates validation algorithm and the grammar and linguistic functions of each language, respectively. The **properties.php** and **propertiesmanager.php** files include the *Property* class and corresponding manager to handle the template condition variables construction. The **templates.php** and **templatesmanager.php** feature a *Template* class and corresponding manager responsible for the production of the summaries. The **entities.php** file defines the *EntityData* parent class and a new *MainEntityData* abstract class that extended *EntityData* and was created to accommodate each domain's main entity (e.g., in the football context, *MatchData*). Then, the **entitiesmanager.php** includes an abstract class responsible for referring expressions generation



Figure 5.3: Directories tree of a specific context.

for entities. Finally, the **data_fetcher.php** file defines a *DataFetcher* abstract class to fetch data given an API endpoint or directory path.

The directory of a context includes all domain-specific files of that context. The **dictionary** and **templates** directories contain the JSON files for the vocabulary and language templates, respectively. In turn, the **fetcher** and **managers** directories possess the data fetcher, entities, properties, and templates managers, which extend the core manager classes described before, with adapted characteristics for the need of the specific domain. Moreover, a new *SummaryParts* abstract class was defined to be handled by the *TemplatesManager* class and deal with the summary division into sections. The following subsections describe football as the primary context and a dummy weather example to demonstrate the new capabilities of the restructured Prosebot system.

5.2.1.1 Football context

The football context already existed in the Prosebot system and has been developed throughout the years. One of the significant aspects considered during refactoring was keeping the football context intact despite the number of modifications made to the system, thus not interfering with match summaries generation. The football context comprises *MatchData* as the single main entity class, *CompetitionData*, *TeamData*, *PersonData*, and *Stat* as sub-entity classes and *Event* and *Curiosity* as auxiliary classes.

5.2.1.2 Weather context

In order to test the viability of the source code restructuring, especially the context generalization process, a new basic context for weather reports was added to the system. The OpenWeatherMap², through its API, was selected to be the data provider, given that its data and database are open and licensed by Open Data Commons Open Database License (ODbL)³. The API provided weather

²<https://openweathermap.org>

³<https://openweathermap.org/full-price#licenses>

conditions of the current day, given the city's id, following the same approach as the football context for match summaries generation.

The weather context comprises a main entity class *CityData* with its properties, which populates the sub-entity classes. The sub-entities include a *MainValuesData* class to express temperature, pressure and humidity values; *CloudValuesData* to convey the percentage of clouds in the sky; *WindValuesData* to indicate wind speed and direction; and a *WeatherTypesData* class to express one of the available types: “Clear”, “Clouds”, “Snow”, “Rain”, “Thunderstorm” or “Atmosphere”. Some basic template condition variables were created to check the weather type and verify the temperature in a quality manner: *#arg.is_hot*, *#arg.is_cold* and *#arg.is_neutral*, and so making it possible for different descriptions accordingly. An example summary generated with Prosebot is shown below for the English version:

- Output:

“A day with blue sky in Lisbon

In the city of Lisbon, in Portugal, it was a sunny day with blue sky.

The current temperature in the city is 25.01°, the maximum and minimum values for the day are 27.68° and 21.53°.

The humidity and pressure values are 44% and 1015hPa, respectively.

Sky with 0% cloudiness.

Wind blowing at a speed of 6.17m/s and with a direction of 10°.

So it's been a pretty hot day.”

As OpenWeatherMap could provide JSON and XML data formats, the *DataFetcher* abstract class and the *WeatherFetcher* sub-class were re-imagined to support also XML data, therefore expanding Prosebot's input data range. This was accomplished with the use of *simplexml_load_string()* PHP function and the *SimpleXMLElement* class.

5.3 Automatic Dictionaries Generation

Dictionaries are the *vars.json* files of every context that store the vocabulary and are used by the templates validation algorithm to check invalid variable names. Writing these files represents an arduous, extensive and monotonous task. With the no-code paradigm in mind and thinking on further contexts added to the system by the community, a re-implementation was envisioned to turn this into an automatic task Prosebot can do.

In the Prosebot system, each entity class implemented the *get_entity()* method, which the *TemplatesManager* object would then use to replace the templates' tokens with variable values. Starting in the main entity class, the method would return one of its property values or forward the request to one of the sub-entity classes. This procedure was carried out by a *switch statement* with options for every available token of an entity and corresponding *return statements*, as presented in the example of Listing 5.4 for the *CompetitionData* class.

```
1 public function get_entity($manager, $entity)
2 {
3     switch ($entity) {
4         case null:
5             return $this;
6         case "name": {
7             return $manager->get_competition_name($this);
8         }
9         default:
10            return null;
11    }
12 }
```

Listing 5.4: *CompetitionData*'s *get_entity* method, before restructuring.

After analysing each entity class's *switch statements*, three patterns arose regarding getting a token value. It was either obtained by simply returning the value of a class attribute, by forwarding the request from the main entity to a sub-entity class, or through referring expression generation with the help of an *EntitiesManager* object. Following these patterns, a new *EntityGetter* abstract class and subsequent *EntityGetterFlat*, *EntityGetterSub*, and *EntityGetterManager* subclasses were defined to accommodate each pattern, respectively. Listing 5.5 shows the prototype and description of each.

```

1  /*
2      EntityGetterFlat:
3      Gets the token's value directly from the return value of a method inside
        the corresponding Entity object.
4      $getter_function - Name of the get method implemented by the Entity object.
5      $has_only_index - Whether the getter function has an event key as parameter
        but not an event itself (optional).
6  */
7  __construct($getter_function , $has_event = false) {}
8
9  /*
10     EntityGetterSub:
11     Gets the Sub-Entity object. EntityGetterSub declarations are only used in
        token lists inside the Main-Entity class.
12     $getter_function - Name of the get method implemented by the Main-Entity
        that returns the Sub-Entity object.
13     $classname - Name of the class of the Sub-Entity.
14     $has_only_index - Whether the getter function has an event key as parameter
        but not an event itself (optional).
15  */
16  __construct($getter_function , $classname , $has_event = false) {}
17
18  /*
19     EntityGetterManager:
20     Uses an EntitiesManager object to handle referring expression generation
        and get the token's value.
21     $manager_function - Name of the method implemented by the EntitiesManager
        object that returns the variable's value. $arg_getter_function - Name of
        the get method implemented by the Entity object that returns the value of
        an argument used by the manager function (optional).
22  */
23  __construct($manager_function , $arg_getter_function = "")

```

Listing 5.5: *EntityGetter* classes.

Then, the *get_entity()* method was abstracted into the *EntityData* parent class, and the *switch statement* for each token was replaced by a *switch statement* for each pattern. These changes made it possible to create a separate static list of tokens for each entity that could be used for template tokens replacement and automatic dictionary generation. Now, each entity class implements a static list of available tokens, where elements' keys are the names of the tokens and their values are *EntityGetter* objects that tell the system how to obtain the variable values. Listing 5.6 shows a portion of the tokens list of the *MatchData* class. In the example:

1. *edition* is a token whose value is returned by the *get_edition()* method of the *MatchData* class;
2. *competition* is the left side of a token whose object is returned by the *get_competition()*

method, and the right side will be handled by the *CompetitionData* class (e.g., *competition.name*);

3. *weekday* is a token whose value will be handled by an *EntitiesManager* object. The *get_weekday()* method is implemented by the *EntitiesManager* object and uses the value returned by the *MatchData*'s *get_date* method as an argument to create the token's value.

```

1 static::$entities = [
2     "edition" => new EntityGetterFlat("get_edition"),
3     "competition" => new EntityGetterSub("get_competition", "CompetitionData"),
4     "weekday" => new EntityGetterManager("get_weekday", "get_date"),
5     (...)
6 ]

```

Listing 5.6: *MatchData*'s list of tokens, after restructuring.

With the newly implemented feature, the user just needs to go to the validation algorithm's page and press a button, which will call the *generate_dictionary()* method of the *TemplatesValidator* object. This function will automatically crawl over the lists of tokens, condition properties, and grammar connectors and produce the dictionary for a context. In the future, if needed, programmers can even expand the feature by building new patterns and adding new *EntityGetter* sub-classes.

5.4 API Decouple and Base Content Definition

One of the ambitions of ZOS with the conversion of Prosebot into open-source software was to decouple the *zerozero.pt* API, due to security and database exclusivity issues. A sample data directory was created in the football context, with JSON files comprising a similar structure, with fewer fields, of the content provided by the API without diminishing the summaries features. Besides, as the data fields of *zerozero.pt* API featured bilingual variable names with European Portuguese, English, and mixed forms, then the sample files and the parsing source code were translated to the standard English language.

The sample data available for the open-source version include three matches of different competitions: a world cup qualifier match between national teams, a match of the Portuguese league, and a match of the champions league. Some extra sample files are yet available for the matches' teams and head-to-head statistics. A similar approach was followed for the weather context, with sample data files retrieved from OpenWeatherMap's API in JSON and XML formats. After the data was gathered, each context-specific data fetcher, meaning the *FootballFetcher* and the *WeatherFetcher* classes, was slightly altered to locate the sample data in the respective directories instead from the APIs endpoints.

5.5 Code Analysis

Once the modifications to the system were concluded and the pretended organization was achieved, the Prosebot generator component was analysed using the SonarQube platform⁴ in terms of reliability, security, maintainability, and code duplication. Overall, the analysis caught a couple of bugs, a few security hotspots needing review, and a significant amount of code smells originated from the code restructuring.

A new iteration in the code restructuring was employed to improve the project's scores. Following the PHP standard rules of writing, almost 45% of the code smells found related to missing curly braces, `{`, around one-liner nested statements, like *if* statements and *for* loops. Moreover, some modifications were done to reduce loops' complexity by merging *if* statements and suppressing redundant jumps. Then, unused functions' parameters were removed, and variables with the same name as classes' properties were renamed, together representing almost 30% of the total number of code smells. Useless defined variables were also deleted to fix the bugs found. Some other general changes were made after. These include removing methods that could be inherited from the parent class, immediately returning expressions stored in useless temporary variables, renaming constants to fit the PHP naming standards, defining constants to avoid duplicating code, and simplifying regular expressions. Some functions were refactored and divided into smaller ones to meet SonarQube's cognitive complexity boundaries and make the code more perceptible. In addition, the *entitiesmanager.php* files for every language were refactored to encapsulate shared code in the parent class, thus avoiding code duplication.

In order to handle exceptions, a new *exceptions.php* file was created in the root of the project with the definition of five dedicated exception classes: *ValidationErrorException*, *UndefinedMethodException*, *UndefinedEntityException*, *UndefinedLanguageException*, *DataFetcherException*, for handling exceptions produced by templates validation, calling of undefined classes' methods, use of undefined entities, selection of an undefined language and fetching of data, respectively. While evaluating *try-catch* statements, in some cases, the exceptions were not being properly handled and were masking errors. Following this issue, in replacing template condition variables with values, using the *strpos()* PHP function for matching was throwing exceptions of undefined indexes. For example, when evaluating *player_goal* in a condition, it would match with *player_goal*, *player_goals*, *#arg.player_goal* and *#arg.player_goal*, and not just the exact variable *player_goal*. This was fixed with regular expression matching with word boundaries.

Finally, the security hotspots were reviewed and appropriately handled. The pseudo-random number generator functions were switched to cryptographically strong ones, and the debugging code of log injection was commented. Also, a new *.env* file was created to store the origins that are allowed to access the API's endpoints and substitute the previous configuration that permitted access to all origins.

⁴<https://www.sonarqube.org>

5.6 Architecture

5.6.1 Activity diagram

Diving further into the internal architecture of the generator component, Figure 5.4 shows an activity diagram representing the flow control and all the activities and main objects created and used from the initial state to the production of an article.

Initially, the user has to choose one of the available contexts. After that, the language and the main entity's id must be specified. The generator algorithm will create a *TemplatesManager* object for the given context and execute some activities. In turn, the *TemplatesManager* will create an *EntitiesManager* object to handle entities' text variations according to language and position in the article, a *PropertiesManager* object to construct templates' conditions, and the main entity object (e.g., *MatchData*, *CityData*). The main entity object is responsible for creating and populating itself and the sub-entities (e.g., *CompetitionData*, *PersonData*, etc.) resorting to a *DataFetcher* object to fetch data from either an API endpoint or a directory path to some samples. In addition, the *TemplatesManager* will also load the template files, each representing a part of the article (e.g., title, intro, final, etc.). For each file loaded, it will filter its templates with the help of the conditions created by the *PropertiesManager* and get the valid ones. From the filtered templates, one is chosen according to weight and a randomness algorithm. Finally, with the help of the *EntitiesManager* and the lists of tokens for each entity (main and sub), the generator can replace the template's tokens with values and write and assemble the paragraphs of the article.

5.6.2 Class diagram

The diagram in Figure 5.5 depicts the hierarchy of classes composing the generator component. A complete version of the diagram, with the principal properties and methods identified, can be seen in Figure B.1.

The Prosebot generator is composed of context-independent and context-dependent classes. The first ones represent the core of the generator and are divided into five categories regarding their functionality, distinguishable in the diagram through their colors (the ColorADD⁵ system symbols were also included for color labeling):

- Grammar-related classes (green);
- Managers (blue);
- Entities parent classes (red);
- Entities getters (purple);
- Templates validation (yellow).

The remaining classes (white) are created according to the necessity of the context in use. The example diagram shows how the football context fits in the hierarchy.

⁵<https://www.coloradd.net/en/>

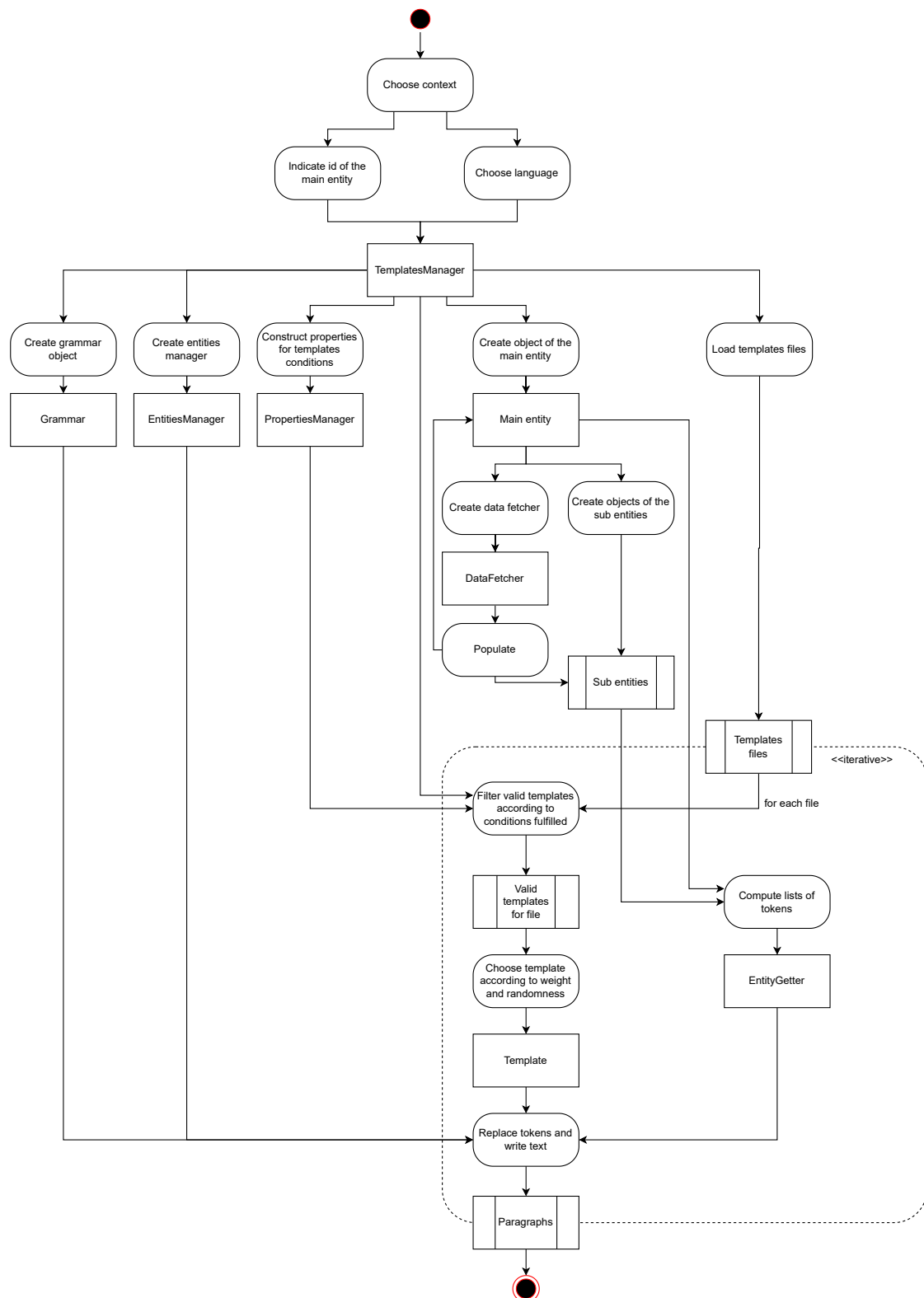


Figure 5.4: Prosebot generator's UML activity diagram.

one due to its characteristics of a copy-left license [36], which promotes knowledge sharing and cooperative improvement of the system:

- Permissions:
 - Allows for commercial and private use;
 - Allows the distribution and modification of the licensed materials;
 - “*Provides an express grant of patent rights from contributors*” [36].
- Conditions:
 - States that the licensed material should be accompanied by a copy of the license and copyright notice, and the source code must be made available upon its distribution;
 - Modifications to the licensed material should be documented and released under the same license.
- Limitations:
 - Limitation in liability;
 - It does not provide any warranty for the project.

5.7.2 Documentation and users support

Despite being an intuitive, organized system, Prosebot may be too extensive for programmers who pretend to start right ahead using it. To decrease the learning curve and clarify the relevant characteristics of the system, a set of wiki pages⁹ were written and included in Prosebot’s GitHub repository. The pages are distributed in three categories: internal system-related features, template writing, and usability and features expansion tutorials. The template writing category includes vocabulary lists with every available variable and description for the two supported contexts.

In addition to the documentation, it was essential to provide means for users to express concerns, ideas, and problems that may arise from the system’s utilization. This aspect was already addressed with the selection of GitHub as the publication location. GitHub provides support for creating issues and pull requests users can use to ask questions and report problems that the repository maintainer can track.

⁹<https://github.com/zerozeropt/prosebot/wiki>

Chapter 6

Templates Management Platform

As part of the Prosebot system, the Templates Management Platform is a pivotal component in opening the development environment to a broader audience by following the no-code paradigm. The platform was implemented using the React JS framework and the CoreUI library's React Admin Dashboard Template and components¹. It provides a user-friendly web interface for creating, editing, deleting, and validating templates.

6.1 API development

The Templates Management Platform has to communicate with other components to be able to execute validation and management of templates. A new API was integrated into the Prosebot generator component to serve the Templates Management Platform with the features of the validation algorithm (handled by the *ValidatorController* class) and the lists of available contexts and languages (handled by the *PropertiesController* class). The API's endpoints, and corresponding methods, are described in Table 6.1.

Table 6.1: Prosebot component API.

ID	Method	URL	Parameters	Data fields	Description
P01	GET	/api/properties/contexts	-	-	Get the list of contexts.
P02	GET	/api/properties/languages	-	-	Get the list of languages.
P03	POST	/api/validator/file	context, lang	data	Validate template file data ^a .
P04	POST	/api/validator/template	context, lang	condition, text	Validate template text-condition pair.

^aConsidering it as a sparse key file, as described in Subsection 4.2.2.

¹<https://coreui.io/react/>

In order to preserve Prosebot's template files while using the Templates Management Platform, a new Prosebot Editor component was created with a copy of the templates directory and an integrated API for their management. This architectural decision made it possible to overcome the need for authentication while keeping the Prosebot component secure and independent. The Prosebot Editor's API's endpoints, handled by the *TemplateController* class, are described in Table 6.2.

Table 6.2: Prosebot Editor component API.

ID	Method	URL	Parameters	Data fields	Description
PE01	DELETE	/api/templates/{id}	context, lang, id	-	Delete template file.
PE02	GET	/api/templates/names	context, lang	-	Get template files' names.
PE03	GET	/api/templates/{id}/data	context, lang, id	-	Get the data of a specific template file.
PE04	POST	/api/templates	context, lang	data, file-name	Create a template file.
PE05	PUT	/api/templates/{id}/data	context, lang, id	data	Edit the data of a template file.
PE06	PUT	/api/templates/{id}/name	context, lang, id	filename	Rename a template file.

6.2 Views

The platform comprises two leading views, easily accessed through the side navigation bar. The home page is the main view of the platform and provides an intuitive interface for template visualization and management. It is divided into two sections: on the right, there are the list of template files and the context and language switch inputs, and on the left, there is a visualization of a selected template file, divided by the keys of the JSON structure. Figure 6.1 shows an example visualization of the home page after opening the *title.json* template file.

In the file import page, Figure 6.2, the user may import a template JSON file to the system. First, the user must select the context and the language to which the template file belongs before uploading it, to ensure that it is placed correctly inside the Prosebot Editor's templates directory.

6.3 Features

This section describes in detail the functionalities provided by the platform and explains how to execute them using the interface. The features are sorted following a distribution that goes from the highest structural level, a template file, to the lowest level, a template text-condition pair.

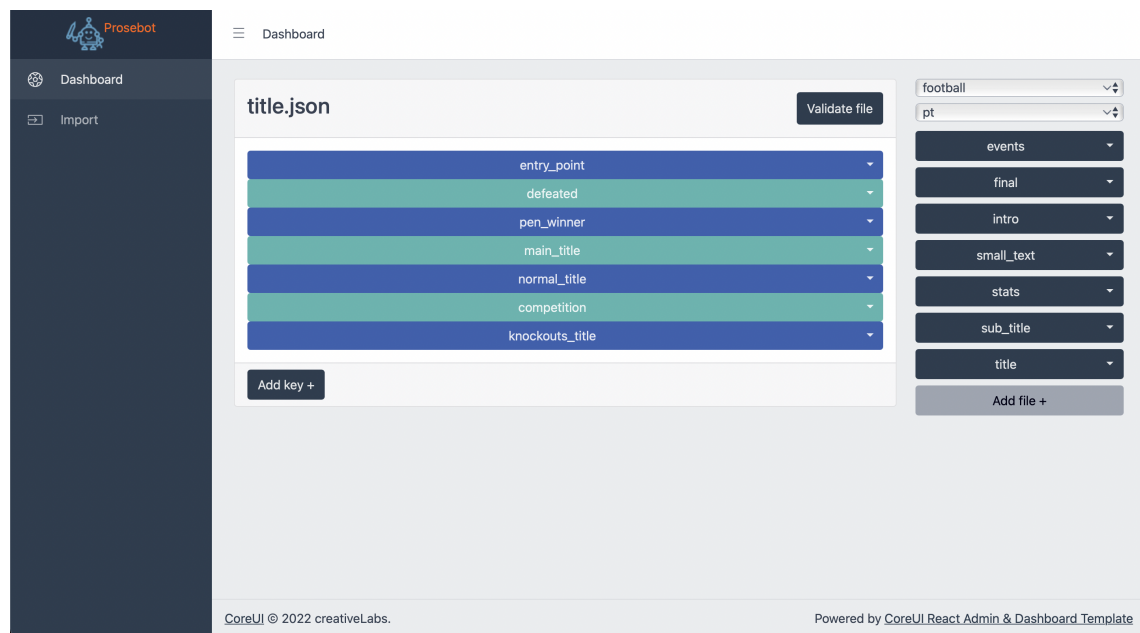


Figure 6.1: Home page.

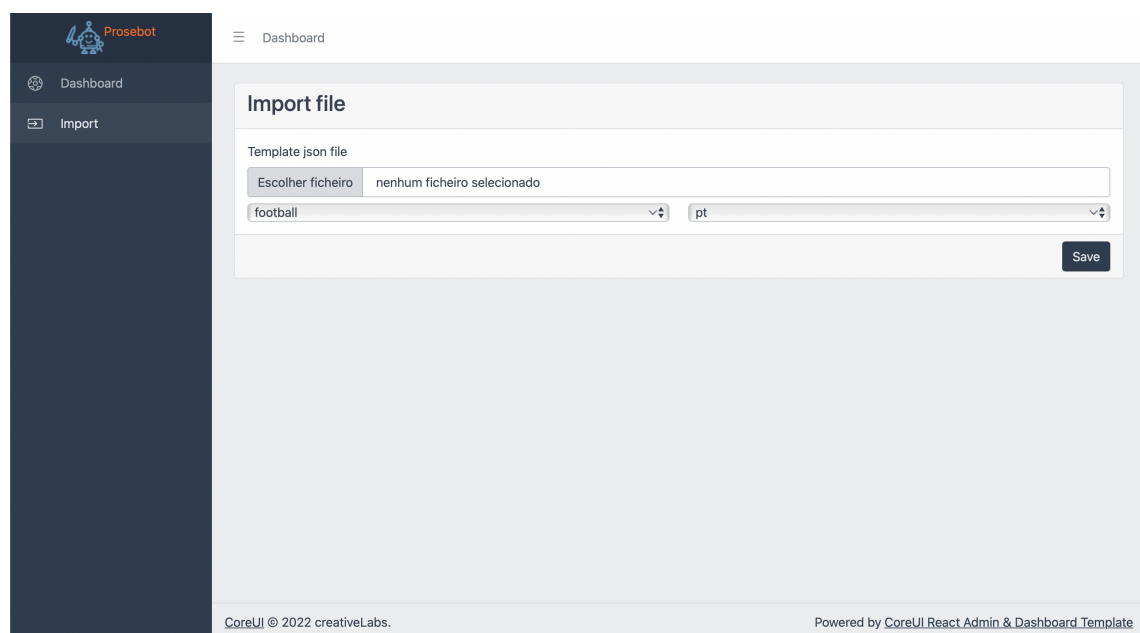


Figure 6.2: File import page.

6.3.1 Manage template files

As mentioned, a user may import a predefined template JSON file (PE04) through the import page's form, Figure 6.2. A user may also visualize the template files' data (PE03). On the right side of the home page, the user should see a list of all the available template files and, on top of it, inputs to switch between contexts and languages. By clicking on one of the file items, a view

of the internal data of the file will open, showing the template keys. In the same way, the user can click on each key to open it and see the corresponding templates or use the forwarding token links inside the template texts, as presented in Figure 6.3 by the “template.defeated” token. In particular, the forwarding token links represent a valuable means to follow the flow of a template text.

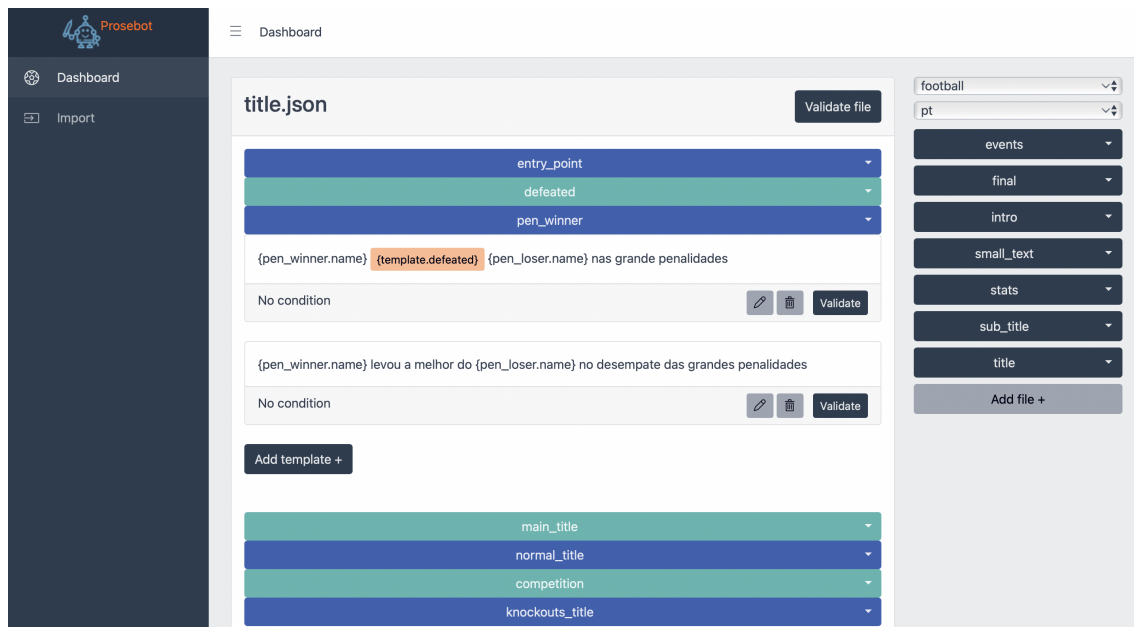


Figure 6.3: Template file view.

Adding to the previous features, the platform allows for direct template file editing through simple interface interactions, each requesting the respective API with server-side support:

- Add a file (PE04): press the “Add file +” button, give the new file a name (default: filename), and save.
- Rename file (PE06): press the arrow alongside the name of the template file, choose “Edit”, rename the file and save.
- Delete file (PE01): press the same arrow, choose “Delete”, and confirm when a pop-up appears, as presented in Figure 6.4.
- Validate file (P04): open the file, press the “Validate file” button and check for error messages on the screen.

6.3.2 Manage template keys

A template JSON file is composed of keys that structure the speech. After opening a file, the user can visualize, add, edit or delete its template keys:

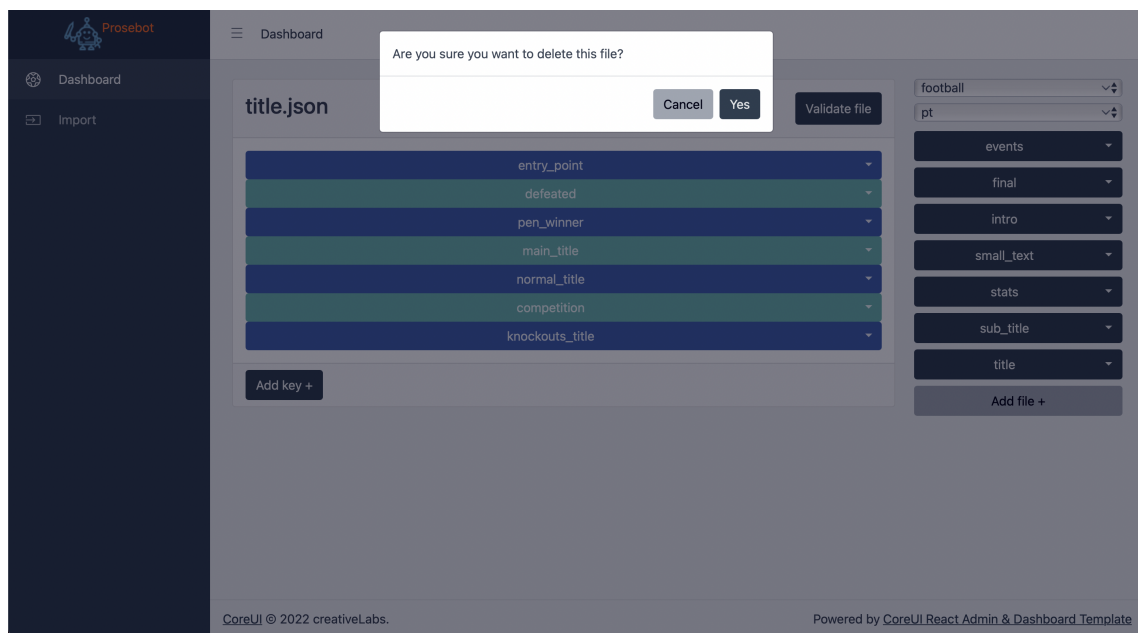


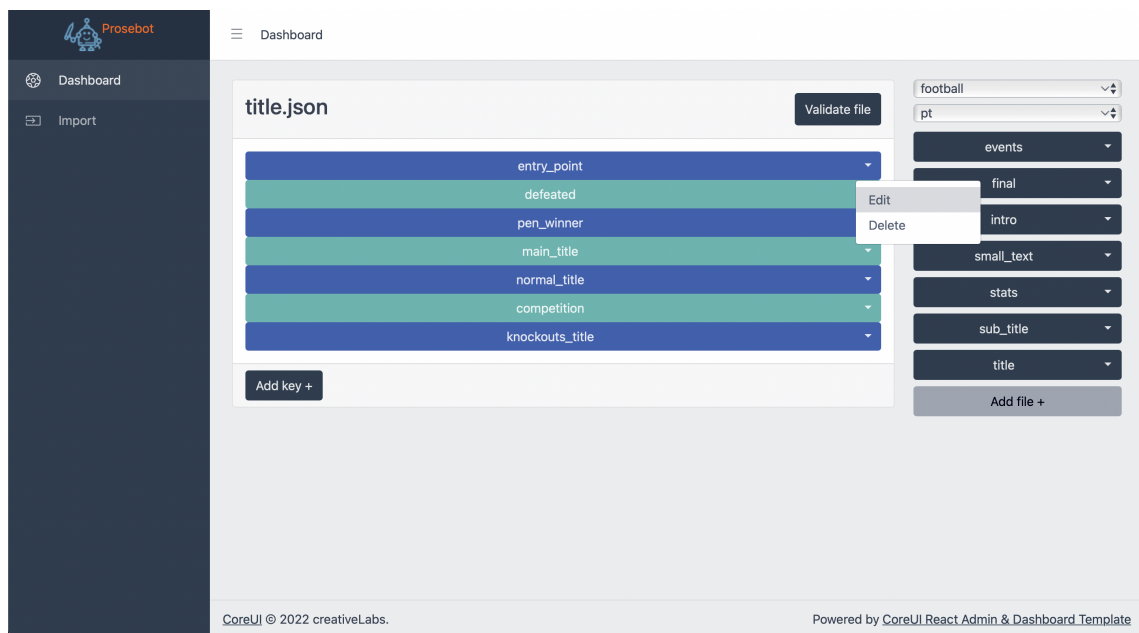
Figure 6.4: Delete template file pop-up.

- Add key (PE05): press the “Add key +” button, give it a name (default: template) and save.
- Rename key (PE05): press the arrow alongside the name of the template key, choose “Edit”, rename the key, and save, as illustrated in Figure 6.5.
- Delete key (PE05): press the same arrow, choose “Delete”, and confirm when a pop-up appears.

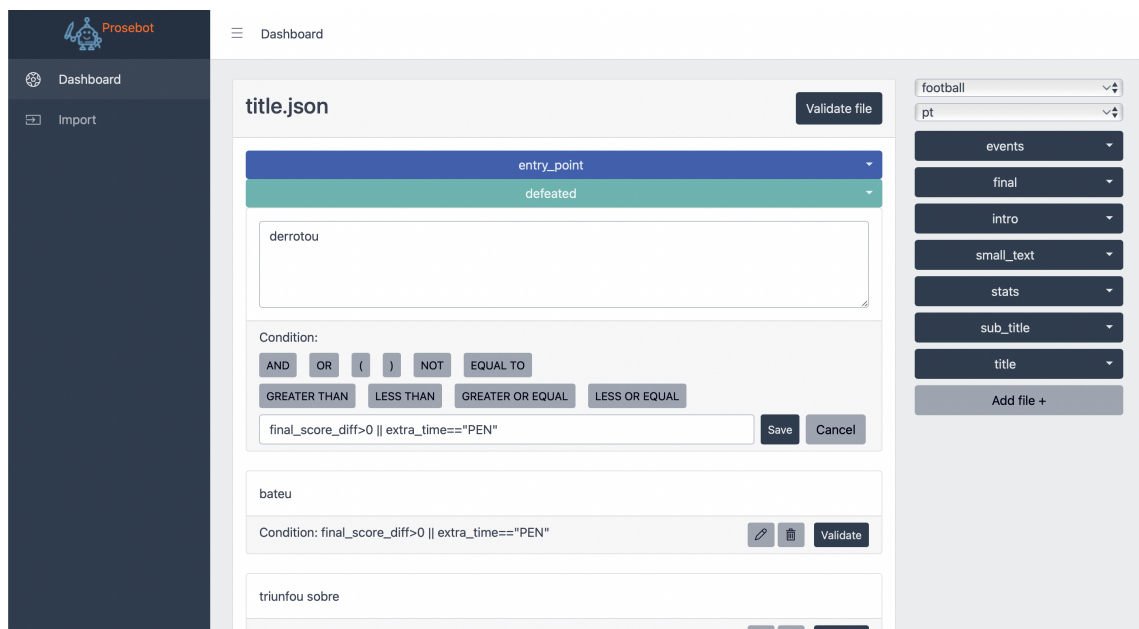
6.3.3 Manage templates

Achieving the lowest structural level, a template is a single text-condition pair element inside a template key. The user can edit text and condition individually, and then validate them.

- Add template to key: with a template key opened, press the “Add template +” button, write the text and the condition (default: empty), and save. The system will automatically validate the pair and ask for confirmation if there are errors.
- Edit template: click the pencil icon, edit the template and save, as illustrated in Figure 6.6. Later, resulting from the feedback given during the UX interviews described in Chapter 7, some buttons with boolean signals were added to help create composite boolean expressions. As discussed, this new inclusion would be valuable for lay users in constructing new conditions. For example, by clicking on the “AND” button, a new “&&” signal will be written in the input field for the condition.
- Delete template: click the trash icon and confirm when a pop-up appears.



- Validate template: click the “*Validate*” button and check for error messages on the screen, as presented in Figure 6.7.



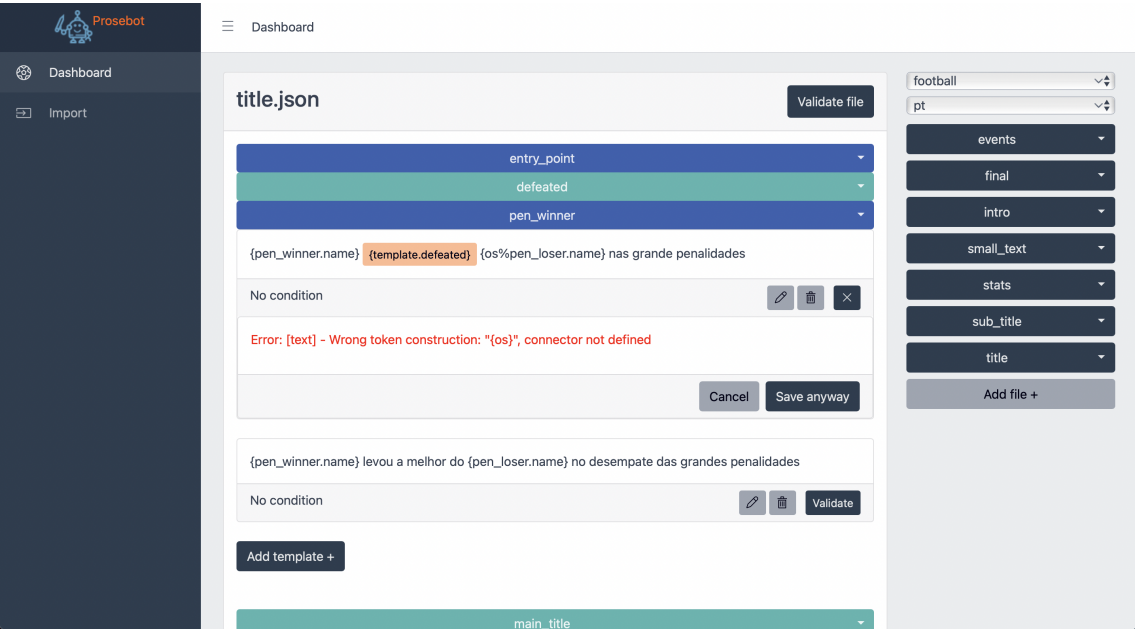


Figure 6.7: Template validation.

Chapter 7

Evaluation

This chapter presents the methodology followed to evaluate each of the elements developed and analyses the results obtained.

7.1 Methodology

7.1.1 Templates validation logs

In order to evaluate the implementation and usefulness of the templates validation algorithm, the “validate and get status” action, presented in Subsection 4.2.6, was integrated into the *zerozero.pt*’s site lab to produce automatic logs each time a journalist edited a template text. The logs are composed of seven fields described in Table 7.1.

Table 7.1: Templates validation logs fields.

Name	Description
id	Identification number of the log
status	0 or 1, where 1 depicts a successful validation and 0 one with errors
error_message	“Success” or the error message returned by the validation
prosebot_sentence	The edited template text
language	One of the supported languages
createdon	The editing date
createdby	The editor username

7.1.2 Code quality metrics

The PHP source code of the Prosebot generator component was evaluated after open-source refactoring using the SonarQube platform. With SonarQube, it was possible to inspect and analyse the

code quality and obtain thorough reports on reliability, security, maintainability, and code duplication. The platform labels those measures on a scale from A to E, following the meanings presented in Table 7.2.

Table 7.2: SonarQube labeling scale. Source: SonarSource S.A [73].

Scale	Maintainability	Reliability (at least)	Security (at least)
A	$\leq 5\%$ of the time spent in the system	0 Bugs	0 Vulnerabilities
B	6 to 10%	1 Minor Bug	1 Minor Vulnerability
C	11 to 20%	1 Major Bug	1 Major Vulnerability
D	21 to 50%	1 Critical Bug	1 Critical Vulnerability
E	$\geq 50\%$	1 Blocker Bug	1 Blocker Vulnerability

To better understand the distinction between the initial and final versions of the Prosebot generator, the produced reports were compared, and the scores were evaluated.

7.1.3 User experience interview

To obtain feedback regarding the user experience with the Templates Management Platform, four journalists of the *zerozero.pt*'s newsroom, who worked in the templates writing, were asked to participate in a short individual interview, accompanied by a brief presentation of the platform in action and a short period of up to ten minutes to experiment with it. Beforehand, an interview guide was written following an online tutorial entitled "Writing an Effective Guide for a UX Interview"¹. The complete interview guide is presented in Appendix C. It comprises several guiding and follow-up questions, with the final objective of answering the following research questions:

1. Did users like the platform's presentation, organization, and coloring?
2. What implemented features were more relevant to the users?
3. Would users change how some functions are executed on the platform?
4. What other features would users expect the platform to have?
5. Do users think the platform is intuitive?
6. Do users think the platform is useful?

7.2 Results

7.2.1 Templates validation algorithm

Upon integration of the algorithm into *zerozero.pt*'s site lab, two initial tests were carried out. Listing 7.1 shows the two logs produced and corresponding correctly generated error messages.

¹<https://www.nngroup.com/articles/interview-guide/>

```

1 // Template text and error message (1):
2 // <p>\r\n {sadasdas </p>\r\n
3 // <b>Error:</b> [text] - Missing "closing bracket }" on chunk "<p>{sadasdas </p>
  >"
4
5 // Template text and error message (2):
6 // <p>\r\n {beste_player.name}</p>\r\n
7 // <b>Error:</b> [text] - Wrong token construction: "{beste_player}", entity
  not defined

```

Listing 7.1: Testing logs of the validation algorithm integration.

Once the algorithm was integrated and operational, the edition of templates by journalists started to produce automatic logs. In total, 795 logs were produced. As the algorithm was being improved between editions, some logs produced wrong results due to a version mismatch between local development and site lab integrated code. After invalid logs removal, 768 remained and were considered for the evaluation. Following a language distribution, 56 were produced in Brazilian Portuguese, 359 in English, 311 in Spanish, and 42 in European Portuguese. Coming into this project, the English and Spanish templates were less developed, which may explain the considerable amount of editions for those languages. On the other hand, European Portuguese templates were closer to a finished version, resulting in fewer editions.

In general, the number of editions decreased over time as the languages got closer to finished versions, with more than half of the logs occurring in April, 417, 146 in May, 194 in June, and only 11 in July. Despite the number of logs not being enough to take statistical meaning from the experiment that could generalise the results, after analysing the data for this specific case, it was possible to note a decrease in the number of writing errors over time made by the *zerozero.pt*'s journalists, both in absolute value and percentage relative to the number of logs produced for each month. Figure 7.1 shows a line chart of the number of detected writing errors across the four months of the experience, and Figure 7.2 presents the percentage of errors over the same period.

Diving further into the validation, 727 template text editions were validated as successful, and 41 threw error messages. All 768 editions were correctly validated according to the writing rules of the templates. Of the errors thrown, 33 corresponded to wrongly defined connectors, 7 to wrongly constructed hyperlinks, and 1 to an invalid property. The Brazilian Portuguese, English, and Spanish templates had their starting point in the European Portuguese version and were then translated and expanded. This factor may explain the large number of wrongly defined connectors and the single error of an invalid property, as the connectors vary between languages while entities' and properties' names are kept.

7.2.2 Open-source refactoring

Initially, the code of Prosebot of the version before this project's work was analysed with the SonarQube platform. The results depicted label A in terms of security and maintainability, with a

Month	Nr errors
April	33
May	8
June	0
July	0

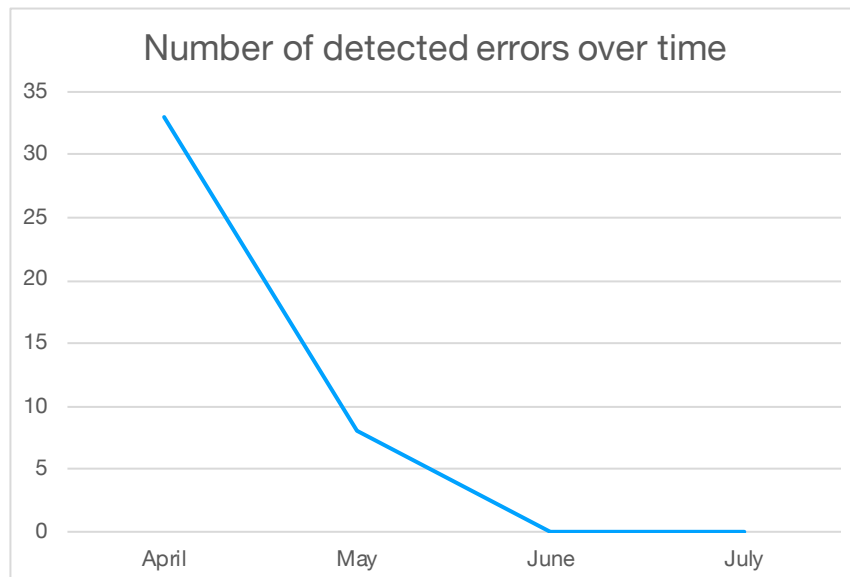


Figure 7.1: Number of detected writing errors over time.

total of 0 vulnerabilities, 6 security hotspots needing review, and 261 code smells with an estimated working debt of 4 days and 3 hours. Regarding reliability, 1 major bug was found, resulting in label C. As for code duplication, the analysis reported a percentage of 15.1% in a total of 23 duplicated blocks. An overview of the scores can be seen in Figure 7.3.

The code analysis made to the final version of the Prosebot generator achieved label A results for the three categories of maintainability, reliability, and security. It reported a total of 0 bugs, 0 vulnerabilities, 1 security hotspot needing review, 41 code smells, and 10.9% of code duplication, as depicted in Figure 7.4.

After revision of the results, the single security hotspot was considered safe, and 18 code smells were acknowledged as false positives since they originated either from deprecated rules or from wrongly indicated unused functions' parameters that need to exist. The remaining 23 issues were left unresolved as the changes would not benefit the final result much, while the effort to overcome them would be high. These included classes with more methods than the 20 allowed, functions with more than 7 parameters, more than 150 lines of code or too high cognitive complexity, methods with more than 3 returns, and use of generic exceptions instead of dedicated ones. The main results can be seen in Figure 7.5.

Analysing further each measure, starting with reliability, there was an improvement from the initial to the final version, with the correction of the major bug and the conversion of label C to

Month	Nr errors	Nr logs	Percentage of errors
April	33	417	7,91%
May	8	146	5,48%
June	0	194	0,00%
July	0	11	0,00%

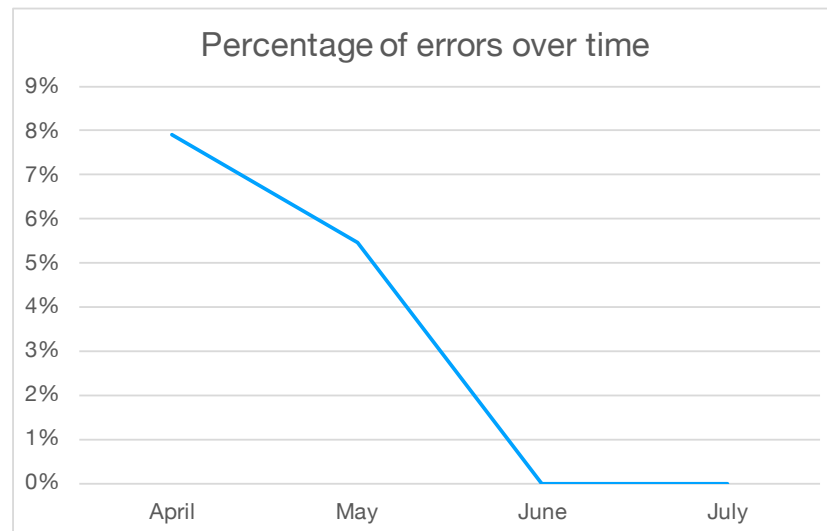


Figure 7.2: Percentage of writing errors over time relative to the number of logs produced.

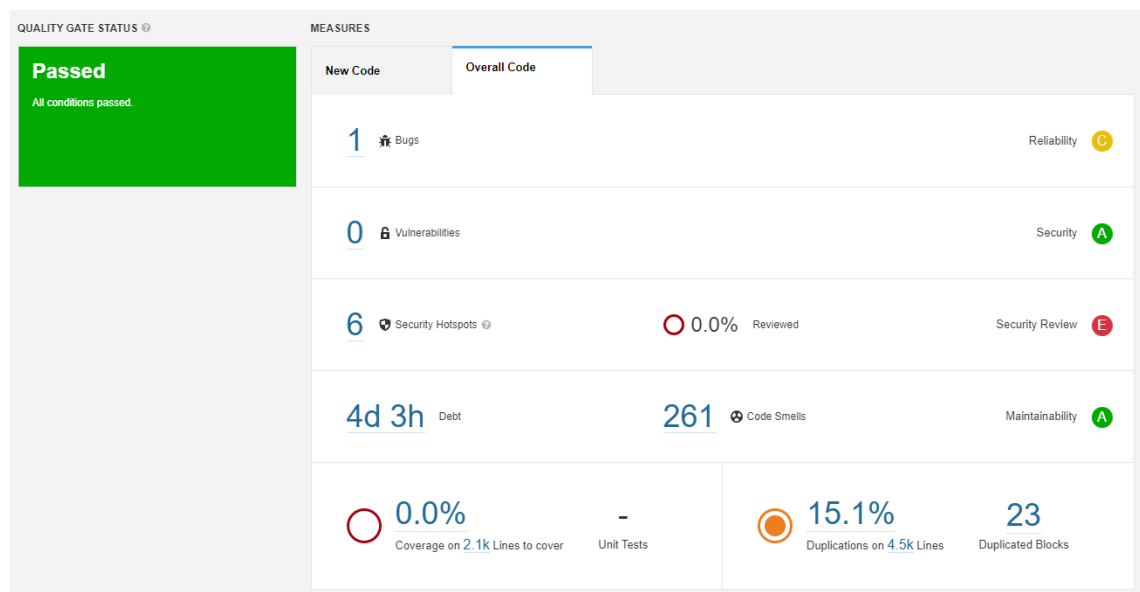


Figure 7.3: SonarQube measures report of the version before the project's work.

label A. Following into the security category, the number of vulnerabilities stayed at zero, keeping

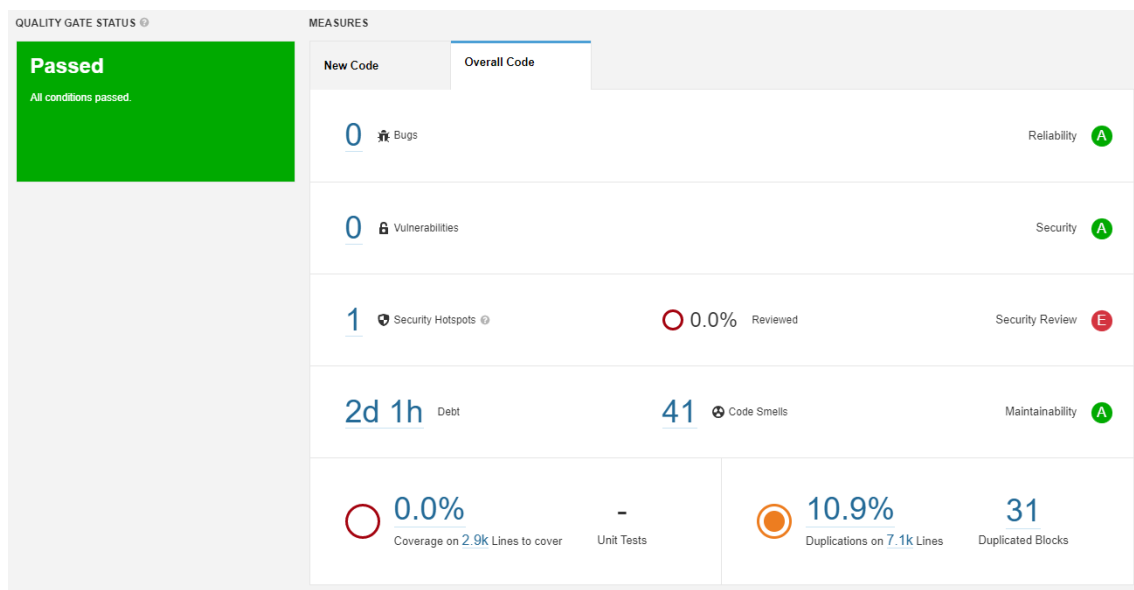


Figure 7.4: SonarQube measures report of the final version before revision.

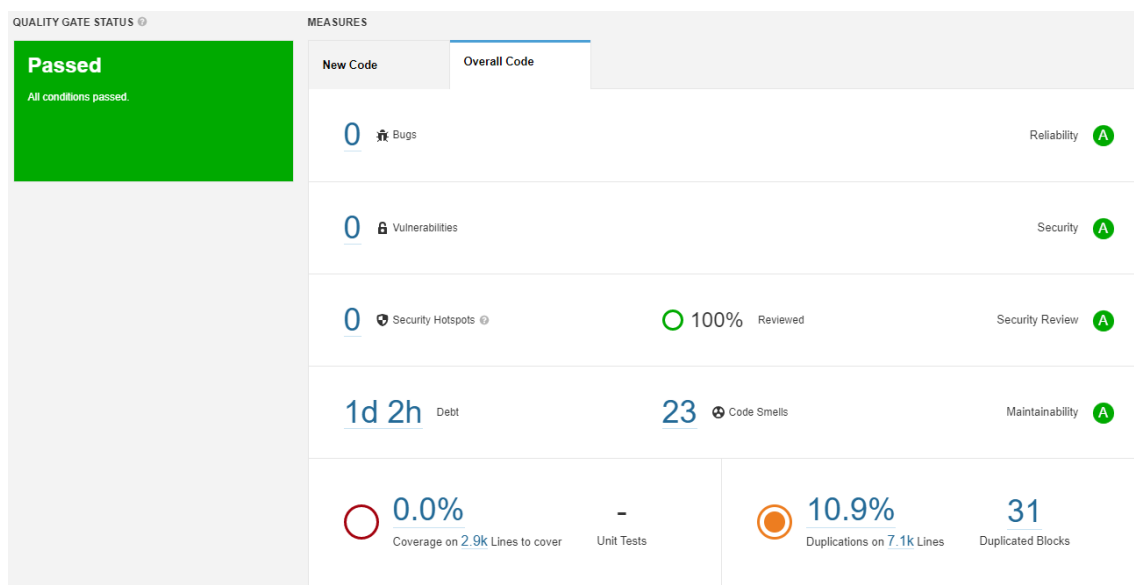


Figure 7.5: SonarQube measures report of the final version after revision.

the label A. The code that originated the security hotspots was adequately reviewed and changed when necessary. The maintainability suffered a considerable improvement, going from an estimated working debt of 4 days and 3 hours, corresponding to a debt ratio of 1.6%, to a working debt of 1 day and 2 hours and a debt ratio of 0.3%. Although the project size increased in the number of lines of code from 4,518 to 7,144 and files from 24 to 54, the duplicate code density decreased from 15.1% to 10.9%. Most of the repeated code is centered on the grammars and entities managers' files between each language. By design, each file must be considered independent,

despite similarities in the code. For example, in the grammar files, some languages may have the same connectors and linguistic functions while others may not. Relevant to notice also that the density of comments had a significant increase from 3.5% to 21.6%, resulting from the inclusion of standard PHP comments in classes, methods, and properties. Finally, although the overall scores improved, the analyses reported an increase in the cognitive complexity of the project from 884 to 1,104, following a calculation method described in the SonarQube documentation for metrics definition [73]. This increase is mainly explained by the context generalization, the development of the templates validation algorithm, and the restrictive number of *loops* and *returns* imposed by SonarQube analysis inside functions.

7.2.3 Templates Management Platform

The results from the user experience interviews for the Templates Management Platform were auspicious, revealing unanimity on most topics and providing new ideas for future improvement. In terms of profile, no significant differences were found between the participants:

- All four declared to use frequently web platforms, mainly social media and news channel websites;
- Two of them were aware of some technical terms and concepts, however, none had programming skills or experience;
- In order to rule out possible differences of opinion due to different visual perceptions, each user was asked whether they were color-blinded, resulting in negative answers.

Regarding the platform's overall presentation, participants agreed that it was well structured and organized, with intuitive interactions, making it easy and logical to use and operate. As highlighted by one of the respondents about the platform - *"I think it is intuitive, it has the parameters well organized (...) I do not have to navigate that much to find the things I need"*. Another interviewee mentioned - *"It allows someone who does not have programming experience to understand at least a little bit better how it works (...)"*. In the same way, unanimously, the participants declared to understand the templates presentation order and division into files, keys, and text-condition pairs. When asked whether they would prefer to spread the functionalities into more web pages/screens of visualization or keep the current condensed version, all agreed the current version was more intuitive and practical. However, one user highlighted the benefits of adding an option to open a template key in a new window to give the possibility of working side by side in template writing.

Talking about specific features of the platform that called the participants' attention, one gave notable importance to the existence of forwarding token links between templates - *"(...) the short cuts you implemented, that is, those buttons to go to other places, I think they are very positive because the platform we use does not have that (...) and yours I believe it is more intuitive because of that. (...) For people who will eventually use this platform and that are not familiarized with what we do, it is most important for them to understand what each does, and by jumping to there,*

I believe it becomes more perceptible". Another feature mentioned was the presentation of the files list menu on the right and the template visualization on the left as a natural and intuitive way of organizing the visualization - *"(...) by having the options in the right and then opening all templates, and knowing that you can jump from an intro to the events or the cards, everything there well structured, I believe it is an advantage"*. Another participant mentioned that the templates validation was a crucial feature to reduce working time and give assurance to what the user is doing - *"I think the validation function because one previous difficulty of this process was the appearance of sentences that did not make sense, things that did not close, so at this moment once you write something you know already if that is validated or not, it is important to save time"*. The same participant also commented that allowing for editing any part of the template files was a helpful feature that contrasted with the more restricted system they use at the moment - *"Mainly due to the fact of everything being editable by us (...) we could make that area of the platform more intelligible for us, the people that feed the platform every day with content"*.

Moreover, all respondents claimed they understood how to execute each major operation of visualizing, editing, deleting, and validating template files, keys, and text-condition pairs. Besides, they would not change how any of them works and felt that the interface made the template's visualization more comprehensible. On the other hand, when asked what other features they would include in the templates management, some great ideas came into the discussion:

- The inclusion of a live preview of text generation, with some sentence examples and possible final results;
- An option to choose the platform's language for users who do not speak English;
- Descriptions explaining the function of each template file in the final result of the generated summary;
- The addition of a new web page with the content of the vocabulary definitions present on the wiki of the Prosebot's repository to help in template writing;
- The implementation of assisting elements for helping users in creating new conditions. Ultimately, this feature was implemented and is described in Subsection 6.3.3.

One significant decision during the architecture definition of the platform was creating the Prosebot Editor's component as a copy of the templates directory. This choice was explained to the participants during the presentation, following appropriate non-technical speech to meet users' understanding and lack of programming skills. Despite all interviewees confirming they understood the need for having such a copy, when asked about including login authentication to surpass this necessity, the participants showed a clear misinterpretation and lack of understanding of the differences and objective of such change in the system. For that reason, and not being an issue directly related to the experience of using the web platform, the answers about this topic were dropped. The exception was the mention of a new feature highlighted by a participant that comprises the existence of authentication and different templates for each registered account.

Regarding the platform's impact on the future of their work, all users preferred using it to manage Prosebot's templates instead of editing the raw template files. One participant emphasized this difference - *"The impact would be absurd, it would be tremendous because it would obviously be much easier using that platform than understanding the files. With it, it would be possible to execute. With the other, only the files, would be more complex. It would require programming knowledge we do not necessarily have"*. Similarly, they agreed that the platform would make it easier and faster to manage templates. Finally, when questioned whether they thought the platform was useful, the opinion was unanimously affirmative. One user stated - *"It is a great idea! I hope it continues to evolve more and more and that it ends up becoming a platform that everyone uses widely"*.

The number of participants in the UX interviews was openly short, and the results allow just for hypothesizing. Thus should not be generalized to other domains or a different number of interviewees. Returning to the research questions defined in Subsection 7.1.3, and considering only the interview answers given:

1. Did users like the platform's presentation, organization, and coloring?
 - Yes.
2. What implemented features were more relevant to the users?
 - The organization and presentation of the platform, the intuitive interactions, the template visualization and validation, the existence of shortcuts, and the fact that it supports the edition of every part of a template file.
3. Would users change how some functions are executed on the platform?
 - No.
4. What other features would users expect the platform to have?
 - Live preview of generated sentences, choice of the interface language, explaining descriptions, and vocabulary definitions to help construct tokens.
5. Do users think the platform is intuitive?
 - Yes.
6. Do users think the platform is useful?
 - Yes.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

At the end of the project, the main goals were accomplished, and an operational version of Prosebot was published as open-source software for free public use on the GitHub platform.

In-depth research on commercial and open-source Natural Language Generation systems and Prosebot background led to a better understanding of the system and its enclosed opportunities. During the development phase, the first steps comprised the enhancement of the generation module of Prosebot and the correction of the summaries produced, with a particular focus on multilingual support and language-related expressions. Then, an algorithm was developed and integrated into Prosebot to validate the writing of templates, thus reporting errors and warnings according to defined syntactic rules and the definition of vocabularies of valid variable names.

Secondly, the Prosebot system suffered a remodeling to make it more accessible to a larger audience. The source code was restructured, the context was generalized so that Prosebot could embody new domains, and the *zerozero.pt*'s API was decoupled and replaced by properly selected sample data. Also, a new dummy weather context was added to provide an example for future users on how to integrate new domains into the system. Thinking on user support, the internal architecture of the generation module was defined and illustrated, and a set of wiki pages composed of tutorials, descriptions, and vocabulary definitions were written.

Moreover, a components architecture was employed, adding two new components beyond the Prosebot generator: the Prosebot Editor, a copy of the templates directory, and the Templates Management Platform. The latter depicts a web interface platform to meet the no-code paradigm and help manage templates. Two APIs were implemented and integrated into Prosebot Editor and Prosebot generator components to manage and validate templates, respectively. The Templates Management Platform communicates through these APIs and acts as a user-friendly interface lay users can use to manage each part of a template file structure. The Prosebot system and documentation were published in a public GitHub repository under an appropriate open-source license.

Finally, further analysis of the Prosebot generator's code with the SonarQube platform showed great results regarding reliability, security, maintainability, and code duplication, and an improvement compared to previous versions.

8.2 Future Work

The work developed paved the way for many projects and new opportunities, from the generation module to the Templates Management Platform. In terms of multilingual support, future development could comprise the inclusion of new languages. During the development phase, besides the improvements made to the Brazilian Portuguese, English, European Portuguese, and Spanish languages and the inclusion of Italian, support for French and German was initiated. However, due to the lack of time, those were left unfinished and out of the final product. On the other hand, with the current possibility of integrating new domains into Prosebot, future expansions may focus on including new entities, properties, and templates for summary generation of players' biographies, teams descriptions, match previews, or other types of domain data. The players' biographies' context support was briefly approached during the development phase. Moreover, a great addition to the templates validation algorithm would be for it to suggest corrections and automatically fix some detected writing errors.

In terms of the Templates Management Platform, future improvements could include drag and drop of interface elements, the possibility of creating a preview of the generated texts, and automatic completion and suggestion of variables during token construction. A more in-depth approach may dive further into the no-code paradigm and use code generation for domain entities, properties, and grammar creation, thus involving lay users even more in the development environment. The components architecture of the Prosebot system can also be changed by including authentication and removing the Prosebot Editor component.

References

- [1] João Aires. “Automatic Generation of Sports News”. Master’s thesis, FEUP, UP, Porto, Portugal, 2016. [Online]. Available: <https://hdl.handle.net/10216/85152>.
- [2] aivancity. “Ludan STOECKLÉ” aivancity.ai. <https://www.aivancity.ai/en/corps-professoral/ludan-stoeckle>. (accessed May. 11, 2022).
- [3] Ion Androutsopoulos, Gerasimos Lampouras, and Dimitrios Galanis. “Generating Natural Language Descriptions from OWL Ontologies: the NaturalOWL System”. *J. Artif. Intell. Res.*, vol. 48:pp. 671–715, 2013. doi: 10.1613/jair.4017.
- [4] Recherche appliquée en linguistique informatique. “jsRealB: a bilingual text realiser for web programming.” ali.iro.umontreal.ca. <http://rali.iro.umontreal.ca/rali/?q=en/jsrealb-bilingual-text-realiser>. (accessed June. 14, 2022).
- [5] John Bateman and Michael Zock. “suregen-2.” fb10.uni-bremen.de. <http://www.fb10.uni-bremen.de/anglistik/langpro/NLG-table/details/SUREGEN-2.htm>. (accessed May. 26, 2022).
- [6] John A. Bateman. “What is KPML?” Universität Bremen. <http://www.fb10.uni-bremen.de/anglistik/langpro/kpml/README.html>. (accessed Feb. 20, 2022).
- [7] John A. Bateman. “Enabling technology for multilingual natural language generation: the KPML development environment”. *Natural Language Engineering*, vol. 3(no. 1):pp. 15–55, Mar. 1997. doi: 10.1017/S1351324997001514.
- [8] Cem Bozsahin, Geert-Jan M. Kruijff, and Michael White. “Specifying grammars for OpenCCG: A rough guide”. *Included in the OpenCCG distribution*, 2005. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.466.4317&rep=rep1&type=pdf>.
- [9] Stephan Busemann. “Best-First Surface Realization”. In *Eighth International Natural Language Generation Workshop, INLG*, Herstmonceux Castle, Sussex, UK, June 12-15 1996. [Online]. Available: <https://aclanthology.org/W96-0411/>.
- [10] Stephan Busemann. “Ten Years After: An Update on TG/2 (and Friends)”. In Graham Wilcock, Kristiina Jokinen, Chris Mellish, and Ehud Reiter, editors, *Proceedings of the Tenth European Workshop on Natural Language Generation, ENLG*, Aberdeen, UK, August 8-10 2005. ACL. [Online]. Available: <https://aclanthology.org/W05-1603/>.
- [11] Lionel Clément. “Elvex.” GitHub repository. <https://github.com/lionelclement/Elvex>. (accessed May. 24, 2022).

- [12] Ann A. Copestake and Dan Flickinger. “An Open Source Grammar Development Environment and Broad-coverage English Grammar Using HPSG”. In *Proceedings of the Second International Conference on Language Resources and Evaluation, LREC*, Athens, Greece, 31 May - June 2 2000. European Language Resources Association. [Online] Available: <http://www.lrec-conf.org/proceedings/lrec2000/html/summary/371.htm>.
- [13] Anne Copestake, John Carroll, Dan Flickinger, Robert Malouf, and Stephan Oepen. “Using an Open-Source Unification-Based System for CL/NLP Teaching”. In *Proceedings of the ACL 2001 Workshop on Sharing Tools and Resources*, 2001. [Online] Available: <https://aclanthology.org/W01-1512>.
- [14] Luís Correia. “Evaluation Metrics for Text and Creation of Writing Tool for Sports Journalism”. Master’s thesis, FEUP, UP, Porto, Portugal, 2020. [Online]. Available: <https://hdl.handle.net/10216/128563>.
- [15] Phillip M. Cunio, Alessandra Babuscia, Zachary J. Bailey, Hemant Chaurasia, Rahul Goel, Alessandro A. Golkar, Daniel Selva, Eric Timmons, Babak E. Cohan, Jeffrey A. Hoffman10, et al. “Initial development of an earth-based prototype for a lunar hopper autonomous exploration system”. In *AIAA SPACE 2009 Conference & Exposition*, Sept. 14-17 2009. doi: 10.2514/6.2009-6713.
- [16] Robert Dale. “Natural language generation: The commercial state of the art in 2020”. *Natural Language Engineering*, vol. 26(no. 4):pp. 481–487, June 2020. doi: 10.1017/S135132492000025X.
- [17] Hercules Dalianis. “ASTROGEN - Aggregated deep and Surface naTuRal language GENerator”. <https://people.dsv.su.se/~hercules/ASTROGEN/ASTROGEN.html>. (accessed May. 24, 2022).
- [18] Hercules Dalianis, P Johannesson, and A Hedman. “Validation of STEP/EXPRESS Specifications by Automatic Natural Language Generation”. *Proceedings of RANLP’97: Recent Advances in Natural Language Processing*, pages pp. 11–13, 1997. [Online]. Available: <https://people.dsv.su.se/~hercules/papers/STEP-NLGvalidation-foto.pdf>.
- [19] Nicolas Daoust and Guy Lapalme. “JSREAL: A text realizer for web programming”. In *Language Production, Cognition, and the Lexicon*, pages 361–376. Springer, 2015.
- [20] Pablo Duboue. “Building Recursive-Descent Natural Language Generators”. <http://duboue.net/blog5.html>. (accessed May. 12, 2022).
- [21] Pablo Duboue. “Generating dynamic prose in PHP”. <http://duboue.net/papers/makewebnotwar20111128.html>. (accessed May. 20, 2022).
- [22] Ondřej Dušek. “TGen.” GitHub repository. <https://github.com/UFAL-DSG/tgen>. (accessed May. 26, 2022).
- [23] Ondřej Dušek and Filip Jurčíček. “Sequence-to-Sequence Generation for Spoken Dialogue via Deep Syntax Trees and Strings”. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL*, volume 2: Short Papers, Berlin, Germany, August 7-12 2016. The Association for Computer Linguistics. doi: 10.18653/v1/p16-2008.
- [24] Michael Elhadad. “CLINT - A Hybrid Template/Word-based Text Generator”. <https://www.cs.bgu.ac.il/~elhadad/clint.html>. (accessed May. 24, 2022).

- [25] Michael Elhadad. “FUF/SURGE.” cs.bgu.ac.il. <https://www.cs.bgu.ac.il/~elhadad/surge>. (accessed June. 1, 2022).
- [26] Michael Elhadad. “*Using argumentation to control lexical choice: a functional unification implementation*”. Columbia University, 1993.
- [27] Michael Elhadad and Jacques Robin. “An Overview of SURGE: a Reusable Comprehensive Syntactic Realization Component”. In *Eighth International Natural Language Generation Workshop, INLG 1996, Herstmonceux Castle, Sussex, UK, June 12-15, 1996 - Posters and Demonstrations*, 1996. [Online] Available: <https://aclanthology.org/W96-0501/>.
- [28] Victor R. Essers and Robert Dale. “Choosing a Surface Realiser: Exploring the Differences in Using KPML/Nigel and FUF/SURGE”. 1998. [Online] Available: <https://www.semanticscholar.org/paper/Choosing-A-Surface-Realiser-Exploring-the-Di-in-and-Essers-Dale/5a4c9a60e886801a172a889e7a3ab38b02fc30f2#citing-papers>.
- [29] Pedro Fernandes. “Community-based Sports Articles Generation Platform using NLG and Post-Editing”. Master’s thesis, FEUP, UP, Porto, Portugal, 2021. [Online]. Available: <https://hdl.handle.net/10216/135617>.
- [30] Association for Computational Linguistics. “Downloadable NLG systems.” aclweb.org. https://aclweb.org/aclwiki/Downloadable_NLG_systems. (accessed May. 5, 2022).
- [31] Stuart Frankel. “Narrative Science signs agreement to be acquired by Salesforce.” Narrative Science’s Resource Blog. [Blog]. <https://narrativescience.com/resource/blog/narrative-science-signs-agreement-to-be-acquired-by-salesforce>. (accessed Feb. 25, 2022).
- [32] Dimitrios Galanis and Ion Androutsopoulos. “Generating Multilingual Descriptions from Linguistically Annotated OWL Ontologies: the NaturalOWL System”. In Stephan Busemann, editor, *Proceedings of the Eleventh European Workshop on Natural Language Generation, ENLG*, Schloss Dagstuhl, Germany, June 17-20 2007. [Online]. Available: <https://aclanthology.org/W07-2322/>.
- [33] Dimitrios Galanis and John Koutsikakis. “Software and data.” Natural Language Processing Group - Department of Informatics - Athens University of Economics and Business. <http://nlp.cs.aueb.gr/software.html>. (accessed Feb. 20, 2022).
- [34] Albert Gatt and Ehud Reiter. “SimpleNLG: A realisation engine for practical applications”. In Emiel Krahmer and Mariët Theune, editors, *Proceedings of the 12th European Workshop on Natural Language Generation (ENLG 2009)*, pages 90–93, Athens, Greece, March 30-31 2009. The Association for Computer Linguistics. [Online]. Available: <https://aclanthology.org/W09-0613/>.
- [35] GitHub. “SimpleNLG.” GitHub repository. <https://github.com/simplenlg/simplenlg>. (accessed May. 31, 2022).
- [36] GitHub, Inc. “GNU General Public License v3.0.” choosealicense.com. <https://choosealicense.com/licenses/gpl-3.0/>. (accessed June. 30, 2022).
- [37] Ben Goertzel, Cassio Pennachin, Samir Araujo, Fabricio Silva, Murilo Queiroz, Ruiting Lian, Welter Silva, Mike Ross, Linas Vepstas, and Andre Senna. “A general intelligence oriented architecture for embodied natural language processing”. In *Proceedings of the 3d Conference on Artificial General Intelligence*, pages 1–8. Atlantis Press, June 2010. doi: 10.2991/agi.2010.16.

- [38] Patrizia Grifoni. “Multimodal fission”. In *Multimodal human computer interaction and pervasive services*, pages 103–120. IGI Global, 2009. doi: 10.4018/978-1-60566-386-9.ch006.
- [39] Mika Härmäläinen. “Poem machine-a co-creative nlg web application for poem writing”. In *Proceedings of the 11th International Conference on Natural Language Generation*, pages 195–196, Tilburg, The Netherlands, November 5-8 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-6525.
- [40] Mika Härmäläinen and Jack Rueter. “Development of an Open Source Natural Language Generation Tool for Finnish”. In *Proceedings of the Fourth International Workshop on Computational Linguistics of Uralic Languages*, pages 51–58, Helsinki, Finland, January 8–9 2018. Association for Computational Linguistics. doi: 10.18653/v1/W18-0205.
- [41] David Hart and Ben Goertzel. “Opencog: A software framework for integrative artificial general intelligence”. In *Artificial General Intelligence 2008, Proceedings of the First AGI Conference*, pages 468–472, University of Memphis, Memphis, TN, USA, March 1-3 2008. IOS Press.
- [42] Sepp Hochreiter and Jürgen Schmidhuber. “Long Short-Term Memory”. *Neural Comput.*, vol. 9(no. 8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735.
- [43] Martin Kay. “Functional Unification Grammar: A Formalism For Machine Translation”. In Yorick Wilks, editor, *10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics, Proceedings of COLING ’84*, pages 75–78, Stanford University, California, USA, July 2-6 1984. ACL. doi: 10.3115/980491.980509.
- [44] Eric Kow. “About GenI.” kowey.github.io. <http://kowey.github.io/GenI/about.html>. (accessed May. 27, 2022).
- [45] Eric Kow. “Graphical User Interface.” kowey.github.io. <http://kowey.github.io/GenI/manual/gui.html>. (accessed May. 27, 2022).
- [46] Eric Kow. “*Surface realisation: ambiguity and determinism. (Réalisation de surface : ambiguïté et déterminisme)*”. PhD thesis, Henri Poincaré University, Nancy, France, 2007. [Online]. Available: <https://tel.archives-ouvertes.fr/tel-00192773>.
- [47] Dirk Kraus. “Suregen-2: a shell system for the generation of clinical documents”. In *EACL 2003, 10th Conference of the European Chapter of the Association for Computational Linguistics*, pages 215–218, Agro Hotel, Budapest, Hungary, April 12-17 2003. The Association for Computer Linguistics. [Online]. Available: <https://aclanthology.org/E03-2008/>.
- [48] ZOS LDA. ZOS.pt. <https://www.zos.pt>. (accessed Feb. 15, 2022).
- [49] Blake A. Lemoine and Lafayette UL. “*NLGen2: a linguistically plausible, general purpose natural language generation system*”. PhD thesis, University of Louisiana at Lafayette, 2010.
- [50] Steven Levy. “Can an Algorithm Write a Better News Story Than a Human Reporter?”. *WIRED*, Apr. 2012. [Online]. Available: <https://www.wired.com/2012/04/can-an-algorithm-write-a-better-news-story-than-a-human-reporter/>.

- [51] François Mairesse, Milica Gasic, Filip Jurčicek, Simon Keizer, Blaise Thomson, Kai Yu, and Steve J Young. “Phrase-Based Statistical Language Generation Using Graphical Models and Active Learning”. In Jan Hajic, Sandra Carberry, and Stephen Clark, editors, *ACL 2010, Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 1552–1561, Uppsala, Sweden, July 11-16 2010. The Association for Computer Linguistics. [Online]. Available: <https://aclanthology.org/P10-1157/>.
- [52] William C Mann and Christian MIM Matthiessen. “Nigel: A Systemic Grammar for Text Generation”. Technical report, University of Southern California Marina Del Rey Information Sciences Inst, 1983.
- [53] Ruli Manurung, Graeme Ritchie, Helen Pain, Annalu Waller, Dave O’Mara, and Rolf Black. “The construction of a pun generator for language skills development”. *Applied Artificial Intelligence*, vol. 22(no. 9):pp. 841–869, Sept. 2008. doi: 10.1080/08839510802295962.
- [54] Tomáš Mikolov, Martin Karafiát, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. “Recurrent neural network based language model”. In Takao Kobayashi, Kei-kichi Hirose, and Satoshi Nakamura, editors, *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association*, pages 1045–1048, Makuhari, Chiba, Japan, September 26-30 2010. ISCA. [Online] Available: http://www.isca-speech.org/archive/interspeech_2010/i10_1045.html.
- [55] Paul Molins and Guy Lapalme. “JSrealB: A Bilingual Text Realizer for Web Programming”. In Anja Belz, Albert Gatt, François Portet, and Matthew Purver, editors, *ENLG 2015 - Proceedings of the 15th European Workshop on Natural Language Generation*, pages 109–111, University of Brighton, Brighton, UK, September 10-11 2015. The Association for Computer Linguistics. doi: 10.18653/v1/w15-4719.
- [56] Johanna D. Moore, Mary Ellen Foster, Oliver Lemon, and Michael White. “Generating Tailored, Comparative Descriptions in Spoken Dialogue”. In Valerie Barr and Zdravko Markov, editors, *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference*, pages 917–922, Miami Beach, Florida, USA, 2004. AAAI Press. [Online]. Available: <http://www.aaai.org/Library/FLAIRS/2004/flairs04-155.php>.
- [57] Amit Moryossef, Yoav Goldberg, and Ido Dagan. “Step-by-Step: Separating Planning from Realization in Neural Data-to-Text Generation”. In Jill Burstein, Christy Doran, and Tamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, volume 1 (Long and Short Papers), pages 2267–2277, Minneapolis, MN, USA, June 2-7 2019. Association for Computational Linguistics. doi: 10.18653/v1/n19-1236.
- [58] Erik T. Mueller. *Daydreaming in humans and machines: a computer model of the stream of thought*. Intellect Books, 1990.
- [59] Erik T. Mueller and Michael G. Dyer. “Towards a computational theory of human daydreaming”. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society*, pages 120–129, 1985. [Online]. Available: <https://arxiv.org/abs/cs/9812010>, <http://web.cs.ucla.edu/~dyer/Papers/CogSci85Daydream.html>.
- [60] Erik T. Mueller and U. Zernik. “GATE reference manual”. Technical report, UCLA-AI-84-5. Artificial Intelligence Laboratory, Computer Science Department, University of California, Los Angeles, 1984.

- [61] Jekaterina Novikova, Ondrej Dusek, Amanda Cercas Curry, and Verena Rieser. “Why We Need New Evaluation Metrics for NLG”. *CoRR*, abs/1707.06875, 2017. [Online] Available: <http://arxiv.org/abs/1707.06875>.
- [62] University of Aberdeen. “STANDUP: System To Augment Non-speakers’ Dialogue Using Puns”. <https://www.abdn.ac.uk/ncs/departments/computing-science/standup-315.php>. (accessed May. 24, 2022).
- [63] University of Illinois Chicago. “Measuring Your Impact: Impact Factor, Citation Analysis, and other Metrics: Citation Analysis”. [researchguides.uic.edu](https://researchguides.uic.edu/c.php?g=252299&p=1683205). <https://researchguides.uic.edu/c.php?g=252299&p=1683205>. (accessed Aug. 8, 2022).
- [64] Raquel Pires. “Prosebot: o comentador de bancada baseado em inteligência artificial.” *Notícias Universidade do Porto*. <https://noticias.up.pt/prosebot-o-comentador-de-bancada-baseado-em-inteligencia-artificial/>. (accessed Feb. 01, 2022).
- [65] Jonathan A. Rees, Norman I. Adams, and James Richard Meehan. “*The T manual*”. Computer Science Department, Yale University, 1984.
- [66] Ehud Reiter. “What is NLG.” Ehud Reiter’s Blog: Ehud’s thoughts and observations about Natural Language Generation. [Blog]. <https://ehudreiter.com/what-is-nlg/>. (accessed Feb. 16, 2022).
- [67] Ehud Reiter. “Why isn’t there More Open-Source NLG Software?” Ehud Reiter’s Blog: Ehud’s thoughts and observations about Natural Language Generation. [Blog]. <https://ehudreiter.com/2017/03/17/open-source-nlg-software/>. (accessed Dec. 27, 2021).
- [68] Ehud Reiter and Robert Dale. “Building Applied Natural Language Generation Systems”. *Natural Language Engineering*, vol. 3(no. 1):pp. 57–87, May 1997. doi: 10.1017/S1351324997001502.
- [69] David Reitter. “Multimodal Functional Unification Grammar”. <http://david-reitter.com/compling/mug/index.html>. (accessed May. 26, 2022).
- [70] David Reitter. “A development environment for multimodal functional unification generation grammars”. *ITRI-04-01 INLG04 Posters: Extended*, page p. 32, Aug. 2004. [Online]. Available: <http://david-reitter.com/pub/reitter2004inlg.pdf>.
- [71] Vasco Ribeiro. “Jornalista-Robot: produção automática de conteúdos de texto como apoio ao jornalismo desportivo”. Master’s thesis, FEUP, UP, Porto, Portugal, 2019. [Online]. Available: <https://hdl.handle.net/10216/121340>.
- [72] Clay Richardson and John Rymer. “Vendor landscape: The fractured, fertile terrain of low-code application platforms”. *FORRESTER*, Jan. 2016. [Online]. Available: <https://www.forrester.com/report/vendor-landscape-the-fractured-fertile-terrain-of-low-code-application-platforms/RES122549?objectid=RES122549>.
- [73] SonarSource S.A. “Metric Definitions”. [docs.sonarqube.org](https://docs.sonarqube.org/latest/user-guide/metric-definitions/). <https://docs.sonarqube.org/latest/user-guide/metric-definitions/>. (accessed July. 15, 2022).
- [74] Sandhya Singh, Hemant Darbari, Krishnanjan Bhattacharjee, and Seema Verma. “Open source NLG systems: A survey with a vision to design a true NLG system”. *International Journal of Control Theory and Applications*, vol. 9(no. 10):pp. 4409–4421, 2016. [Online]. Available: https://serialsjournals.com/abstract/42229_cha-24.pdf.

- [75] João Soares. “Statistical Language Models applied to News Generation”. Master’s thesis, FEUP, UP, Porto, Portugal, 2017. [Online]. Available: <https://hdl.handle.net/10216/106475>.
- [76] Mark Steedman. “*The syntactic process*”. Language, speech, and communication. MIT Press, 2004.
- [77] Ludan Stoecklé. “RosaeNLG Documentation.” RosaeNLG.org. <https://rosaelng.org>. (accessed May. 11, 2022).
- [78] Ludan Stoecklé. “RosaeNLG Tutorial for English.” RosaeNLG.org. https://rosaelng.org/rosaelng/3.2.2/tutorials/tutorial_en_US.html. (accessed June. 15, 2022).
- [79] Mariet Theune, Esther Klabbers, Jan-Roelof De Pijper, Emiel Krahmer, and Jan Odijk. “From data to speech: a general approach”. *Natural Language Engineering*, vol. 7(no. 1):pp. 47–86, Mar. 2001. doi: 10.1017/S1351324901002625.
- [80] TokenMill UAB. “Accelerated Text.” AcceleratedText.com. <https://www.acceleratedtext.com/>. (accessed Feb. 20, 2022).
- [81] TokenMill UAB. “Introduction” Accelerated Text Documentation. <https://accelerated-text.readthedocs.io/en/latest/>. (accessed Feb. 28, 2022).
- [82] Chris van der Lee, Emiel Krahmer, and Sander Wubben. “PASS: A Dutch data-to-text system for soccer, targeted towards specific audiences”. In *Proceedings of the 10th International Conference on Natural Language Generation*, pages 95–104, Santiago de Compostela, Spain, September 4-7 2017. Association for Computational Linguistics. doi: 10.18653/v1/W17-3513.
- [83] Chris van der Lee, Bart Verduijn, Emiel Krahmer, and Sander Wubben. “Evaluating the text quality, human likeness and tailoring component of PASS: A Dutch data-to-text system for soccer”. In Emily M. Bender, Leon Derczynski, and Pierre Isabelle, editors, *Proceedings of the 27th International Conference on Computational Linguistics, COLING*, pages 962–972, Santa Fe, New Mexico, USA, August 20-26 2018. Association for Computational Linguistics. [Online] Available: <https://aclanthology.org/C18-1082/>.
- [84] Pierre-Luc Vaudry and Guy Lapalme. “Adapting SimpleNLG for Bilingual English-French Realisation”. In Albert Gatt and Horacio Saggion, editors, *ENLG 2013 - Proceedings of the 14th European Workshop on Natural Language Generation*, pages 183–187, Sofia, Bulgaria, August 8-9 2013. The Association for Computer Linguistics. [Online]. Available: <https://aclanthology.org/W13-2125/>.
- [85] Tsung-Hsien Wen and Steve J. Young. “Recurrent neural network language generation for spoken dialogue systems”. *Comput. Speech Lang.*, vol. 63:101017, 2020. doi: 10.1016/j.csl.2019.06.008.
- [86] Michael White. “Reining in CCG Chart Realization”. In Anja Belz, Roger Evans, and Paul Piwek, editors, *Natural Language Generation, Third International Conference, INLG 2004, Brockenhurst, UK, July 14-16, 2004, Proceedings*, volume 3123 of *Lecture Notes in Computer Science*, pages 182–191. Springer, 2004. doi: 10.1007/978-3-540-27823-8_19.

- [87] Michael White. “Designing an extensible API for integrating language modeling and realization”. In *Proceedings of Workshop on Software*, pages 47–64, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics. [Online] Available: <https://aclanthology.org/W05-1104.pdf>.
- [88] Michael White. “Efficient realization of coordinate structures in Combinatory Categorical Grammar”. *Research on Language and Computation*, vol. 4(no. 1):pp. 39–75, 2006.
- [89] Michael White and Kim K. Baldridge. “Adapting Chart Realization to CCG”. In *Proceedings of the 9th European Workshop on Natural Language Generation, ENLG@EACL*, Budapest, Hungary, April 13-14 2003. The Association for Computer Linguistics. [Online]. Available: <https://aclanthology.org/W03-2316/>.
- [90] Michael White, Mark Steedman, Jason Baldridge, and Geert-Jan Kruijff. “OpenCCG.” GitHub repository. <https://github.com/OpenCCG/openccg>. (accessed Feb. 20, 2022).
- [91] Zhaohang Yan. “The Impacts of Low/No-Code Development on Digital Transformation and Software Development”. *CoRR*, abs/2112.14073, Dec. 2021. doi: 10.48550/arXiv.2112.14073.
- [92] Šarūnas Navickas. “AcceleratedText: A short guide.” medium.com. <https://medium.com/accelerated-text/acceleratedtext-a-short-guide-d6363a50de6a>. (accessed Feb. 28, 2022).
- [93] Žygimantas Medelis. “No Code — a perfect fit for Natural Language Generation.” medium.com. <https://medium.com/accelerated-text/no-code-a-perfect-fit-for-natural-language-generation-bbe3027d22ed>. (accessed Dec. 27, 2021).

Appendix A

Open-Source NLG Solutions

Table A.1: Open-source NLG solutions.

Stage	Name	Year	Programming Language	Languages	License	Institution
Linguistic Realiser	FUF/SURGE [25, 26, 27, 28]	1992-96	Common Lisp	English	GNU GPL v.2	Ben Gurion University of the Negev, Israel
	KPML [6, 68, 7]	1993	ANSI Common Lisp	Multilingual	-	University of Bremen, Germany
	TG/2 [9, 74, 10]	1998	Lisp	Multilingual	-	German Research Center for Artificial Intelligence, University of Saar, Germany
	GenI [46, 44]	2007	Haskell	English, French	GNU GPL v.2	-
	SimpleNLG [34, 35]	2009	Java	Multilingual	MPL v.2	University of Aberdeen, UK
	JSrealB [55, 4]	2015	JavaScript	English, French	Apache 2.0	University of Montreal, Canada
	Syntax Maker [40]	2018	Python	Finnish	Apache 2.0	University of Helsinki, Finland
Sentence Planner, Linguistic Realiser	ASTROGEN [18, 17, 74]	1996-99	Prolog	English	-	Royal Institute of Technology and Stockholm University, Sweden
	MUG [74, 70]	2002	Prolog	English	GNU GPL v.2	MIT Media Lab Europe, Dublin, Ireland
	OpenCCG [90, 8, 89, 88, 87, 86]	2003	Java	English	GNU LGPL v2.1	-
	TGen [23, 22, 51]	2014	Python	English	Apache 2.0	Charles University in Prague, Czech Republic
	RNNLG [85, 54, 61, 42]	2016	Python	English	Apache 2.0	University of Cambridge, UK
	RosaeNLG [77, 2]	2018	JavaScript, Pug	Multilingual	Apache 2.0	LF AI & Data Foundation, Linux Foundation
Text Planner, Sentence Planner, Linguistic Realiser	Suregen-2 [47, 74, 5]	2002	Common Lisp	English, German	-	University for Health Sciences, Medical Informatics and Technology, Austria
	NaturalOWL [3, 33, 32]	2007	Java	English, Greek	GNU GPL v.2	Athens University of Economics and Business, Greece
	PASS [82, 83]	2017	Python	Dutch	GNU GPL v.3	Tilburg University, The Netherlands
	Chimera [57]	2019	Python	English	MIT	Bar Ilan University, Israel
	Accelerated Text [80, 93, 81]	2020	JavaScript, Clojure	Multilingual	Apache 2.0	TokenMill UAB, Lithuania
-	DAYDREAMER [59, 58]	1983-88	GATE, T	English	GNU GPL v.2	University of California, USA
	LKB [13]	1991	Common Lisp	Multilingual	MIT	University of Cambridge, UK; University of Sussex Falmer, UK; Stanford University and YY Software, USA; University of Groningen, The Netherlands
	CLINT [74, 24]	1999-2000	C++	English	-	Ben Gurion University of the Negev, Israel
	STANDUP [74, 62, 53]	2003	Java	English	Custom License	University of Dundee, University of Edinburgh and University of Aberdeen, UK
	NLGen & NLGen 2 [74, 37, 49]	2009	Java	English	Apache 2.0	University of Louisiana at Lafayette, USA
	PHP-NLGen [21, 20]	2011	PHP	English, French	MIT	Textualization Software Ltd., Canada
	Elvex [11]	2019	C++	English, French	GNU GPL v.3	Bordeaux University, France

Appendix B

Complete Class Diagram

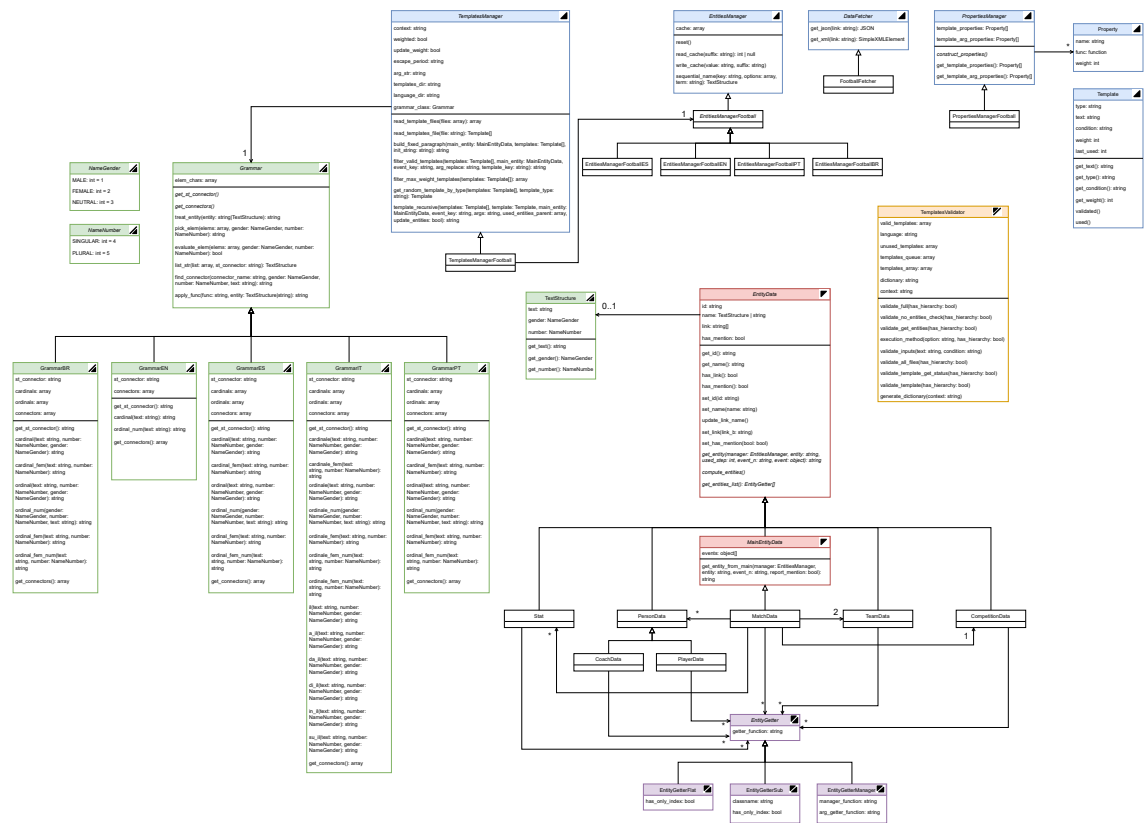


Figure B.1: Prosebot generator's complete UML class diagram.

Appendix C

Templates Management Platform UX Interview Guide

Research Questions:

1. Did users like the platform's presentation, organization, and coloring?
2. What implemented features were more relevant to the users?
3. Would users change how some functions are executed on the platform?
4. What other features would users expect the platform to have?
5. Do users think the platform is intuitive?
6. Do users think the platform is useful?

Platform Presentation:

1. Give an initial description of the web platform, its purpose, and overall organization.
2. Explain the creation of a copy of the templates directory.
3. Present the template file structure and explain how it influenced the visualization.
4. Present the two main pages of the platform.
5. Showcase each significant feature: import a file, visualize, create, edit, delete and validate each part of the template files.
6. Finally, let the user experiment freely with the interface.

Interview Guide:

1. Before we begin, would you mind telling me a bit about your personal experience with web applications and interfaces?

- Are you colorblind?
 - Do you have programming experience?
 - How often do you work with web applications?
2. What do you think in terms of the platform's presentation and organization?
 - Would you spread the functionalities into more web pages/screens of visualization, or do you prefer this condensed version?
 - Do you understand the templates division into files, keys, and text-condition elements?
 3. After seeing the platform working, what were the main features that caught your attention?
 - Do you understand how to execute each of the major functionalities?
 - Do you like the way each of them is executed?
 - Do you understand how to visualize the templates?
 - Do you understand how to import templates?
 - Do you understand how to manually add new templates?
 - Do you understand how to edit and delete templates?
 - Do you understand how to validate templates?
 4. What would you change in the platform?
 - Did you think the interface interactions explained were intuitive?
 - Would you change the way any implemented feature works?
 - What other features would you add to the platform?
 5. What do you think of the platform needing a copy of the templates directory?
 - Do you understand why the copy is needed?
 - What do you think about including authentication to prevent the necessity of having a copy of the templates directory?
 - Do you think it is worth it?
 - How much do you think it would impact the user experience?
 6. What impact would this platform have on your work while managing Prosebot's templates?
 - Would you use this platform to manage Prosebot's templates?
 - Would you instead use this platform than directly editing the JSON files?
 - Do you think it would make the templates' visualization more comprehensible?
 - Do you think it would turn it easier to manage the templates?
 - Do you think it would turn it faster to manage the templates?
 - Do you think the platform is useful?
 7. Would you like to share anything else about this topic?