



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Alexandre Costa Dias

ONTODL+

**An ontology description
language and its compiler**

October 2021



Universidade do Minho

Escola de Engenharia

Departamento de Informática

Alexandre Costa Dias

ONTODL+

**An ontology description
language and its compiler**

Master dissertation

Master Degree in Informatics Engineering

Dissertation supervised by

Pedro Rangel Henriques (Ph.D.)

Cristiana Esteves Araújo (M.Sc.)

October 2021

AUTHOR COPYRIGHTS AND TERMS OF USAGE BY THIRD PARTIES

This is an academic work which can be utilized by third parties given that the rules and good practices internationally accepted, regarding author copyrights and related copyrights.

Therefore, the present work can be utilized according to the terms provided in the license bellow.

License provided to the users of this work



Attribution-NonCommercial

CC BY-NC

<https://creativecommons.org/licenses/by-nc/4.0/>

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Alexandre Dias

ACKNOWLEDGMENTS

The development of this thesis was a long process which would have been impossible without the help of some people.

I would like to start by thanking my parents and my father especially for the invaluable support and motivation provided during this period.

I would also like to extend a big word of thank you to my mentors, Pedro Rangel Henriques and Cristiana Araújo, for always lighting the way and providing help while accompanying the development of this thesis.

I want to thank José Carlos Ramalho and Salete Teixeira as well, for providing the ontologies used for testing and demonstrating the work developed.

Lastly, I thank my friends for the moral support provided and for always being there when needed.

ABSTRACT

Ontologies are very powerful tools when it comes to handling knowledge. They offer a good solution to exchange, store, search and infer large volumes of information. Throughout the years various solutions for knowledge-based systems use ontologies at their core.

OntoDL has been developed as a Domain Specific Language using ANTLR4, to allow for the specification of ontologies. This language has already been used by experts of various fields has a way to use computer-based solutions to solve their problems.

In this thesis, included on the second year of the Master degree in Informatics Engineering, OntoDL+ was created as an expansion of the original OntoDL. Both the language and its compiler have been improved. The language was extended to improve usability and productivity for its users, while ensuring an easy to learn and understand language. The compiler was expanded to translate the language specifications to a vaster array of languages, increasing the potential uses of the DSL with the features provided by the languages.

The compiler and some examples of the DSL can be downloaded at the website https://epl.di.uminho.pt/~gepl/GEPL_DS/OntoDL/ created for the application and presented in the final chapters of the thesis.

Keywords: Ontology, Domain Specific Language, Automatic Code Generation

RESUMO

As ontologias são formalismos muito poderosos no que toca a manipulação de conhecimento. Estas oferecem uma boa solução para trocar, armazenar, procurar e inferir grandes volumes de informação. Ao longo dos anos, várias soluções para sistemas baseados em conhecimento usaram ontologias como uma parte central do sistema.

A OntoDL é uma Linguagem de Domínio Específico que foi desenvolvida através do uso de ANTLR4, para permitir a especificação de ontologias. Esta linguagem foi já utilizada por especialistas de diversas áreas como forma de utilizar soluções informáticas para resolver os seus problemas.

Nesta tese, incluída no segundo ano do Mestrado em Engenharia Informática, OntoDL+ foi criado como uma expansão tanto à linguagem e como ao seu compilador. A linguagem foi estendida para melhorar a usabilidade e produtividade dos seus utilizadores, mantendo-se fácil de aprender e perceber. O compilador foi expandido para ser capaz de traduzir as especificações de OntoDL+ para um leque de linguagens mais vasto, aumentando os potenciais usos da DSL através das funcionalidades providenciadas pelas linguagens alvo.

O compilador e alguns exemplos da DSL podem ser acedidos no sítio https://epl.di.uminho.pt/~gepl/GEPL_DS/OntoDL/ criado para a aplicação e mostrado nos capítulos finais da tese.

Palavras Chave: Ontologia, Linguagem de Domínio Específico, Geração Automática de Código

CONTENTS

1	INTRODUCTION	1
1.1	Motivation	2
1.2	Objetives	2
1.3	Research Method	3
1.4	Research Hypothesis	4
1.5	Document Structure	4
2	STATE OF THE ART	5
2.1	Ontologies	5
2.2	Logic Programming	6
2.2.1	Prolog	7
2.3	Domain Specific Languages	8
2.3.1	Advantages of DSLs	8
2.3.2	Disadvantages of DSLs	10
2.3.3	DSL Design and Implementation	10
2.4	Software development	11
2.4.1	Code Refactoring	11
2.4.2	Model Checking	17
2.5	Summary	18
3	ONTODL	20
3.1	Grammar	21
3.2	Semantic actions	25
3.3	Features	28
3.4	Summary	29
4	ONTODL+, SPECIFICATION	30
4.1	System Architecture	30
4.2	Language extensions	31
4.2.1	Triples grouping	31
4.2.2	Ontology Constraints	32
4.2.3	Idiomatic and syntactic normalization	33
4.2.4	Normalization of data properties	33
4.2.5	Relation specification	34
4.3	Compiler expansions	35
4.3.1	Alloy	35
4.3.2	Prolog	37

4.4	Summary	38
5	ONTODL+, DEVELOPMENT	39
5.1	Code Refactoring	39
5.1.1	Java Package	40
5.2	Language extensions	41
5.2.1	Triples grouping	41
5.2.2	Relation properties	42
5.2.3	Constraints	45
5.2.4	Attributes Normalization	46
5.2.5	Expands Feature	47
5.3	Compiler expansions	47
5.3.1	Alloy	47
5.3.2	Prolog	50
5.4	Summary	51
6	ONTODL+, WEBPAGE AND EXPLORATION	52
6.1	Home	52
6.2	Examples	53
6.3	Download	54
6.4	Contact	55
6.5	Summary	56
7	ONTODL+, RUNNING EXAMPLE	57
7.1	OntoMapa	57
7.1.1	Triple Grouping	59
7.1.2	Relation properties	59
7.1.3	Attributes Normalization	60
7.1.4	Expand	61
7.1.5	Processing	61
7.2	OntoJogo	66
7.2.1	Processing	67
7.3	Summary	71
8	CONCLUSION AND FUTURE WORK	72
8.1	Conclusions	72
8.2	Future Work	73

ACRONYMS

A

ANTLR ANOther Tool for Language Recongnition.

D

DSD Domain Specific Description.

DSL Domain Specific Language.

G

GPL General Purpose Language.

I

IDE Integrated Development Environment.

IRI Internationalized Resource Identifier.

O

ONTODL Ontology Description Language.

OWL Web Ontology Language.

P

PROLOG Programming Logic.

U

URI Uniform Resource Identifier.

URL Uniform Resource Locator.

INTRODUCTION

Ontologies can be used in knowledge domains as a form of declarative formalism. Through ontologies it is possible to represent the different objects, as well as relationships between them, in human-readable text by using the representation vocabulary defined for the ontology (Gruber, 1993). These can solve many problems regarding maintenance and growth of systems, due to the ability to write, search and store information in a way that is understandable for both humans and machines (Martini and Henriques, 2016; Gali et al., 2004).

The heterogeneity of the hardware system, as well as the different programming languages and network protocols, poses a challenge to the creation of shared, reusable knowledge-based software. The inter-operability of said system requires that they are able to operate and communicate using a formal knowledge representation. That said, in order to try and solve this problem the use of ontologies has since long been proposed and analysed (Gruber, 1995).

Currently, various knowledge-based systems are resorting to the usage of ontologies as a way to exchange information/knowledge between heterogeneous systems. The combination of knowledge from different systems, has also proven to be quite useful in information dependent systems (Araújo, 2016; Lehmann et al., 2015; Brickley and Miller, 2014).

As such the tool OntoDL was one of the tools created as a means to enable an easier way to specify ontologies and translate them into other languages. This Domain Specific Language (DSL) created using ANTLR4, has already been used to create and manage ontologies in other projects (Martins et al., 2019). It looks primarily to ensure an easy way to create and analyse ontologies without the need of previous knowledge basis relative to computing. However the language and its compiler can be still improved to widen the range of usages of the tool.

During this Master Thesis, OntoDL+ will be implemented as an improved version of OntoDL. The language will be extended for a better usage, and its compiler expanded in order to allow for a more wide variety of usages for the tool.

1.1 MOTIVATION

This Master Thesis was done in order to enable both less experienced and experts with the usage and specification of ontologies, to be more efficient in their work.

It also provides a more intuitive visualization of the ontologies created. Providing a more intuitive way to interpret the information and how it is represented.

Finally given the formal nature of ontologies and their connection with artificial intelligence, OntoDL+ also provided a closer and easier connection with logic programming and model checking.

1.2 OBJETIVES

Considering that the main purpose of this project is to improve OntoDL, the objectives defined for the Master Thesis here reported are:

1. Extend the language to allow the instantiation of attributes during individuals' declarations.
2. Extend the language to remove redundancies in the declaration of triples.
3. Adapt the context validator to ensure the semantic correction of every OntoDL+ specification.
4. Expand the capabilities of the language compiler by adding the possibility of conversion to other ontology formats.
5. Integrate the Domain Specific Language with external tools in order to allow for a better visualization of the ontologies created.
6. Expand the capabilities of the language compiler to allow for the conversion of ontologies to logic programming languages as well as the dynamic validation, inference and exploration of the ontology resorting to said languages.
7. Extend the capabilities of the language to allow a conversion to specification languages, enabling formal model checking of the ontologies' specifications written.

Objectives 1. through 3., aim at extending OntoDL+, as a means to improve its usability and the productivity of it's users while using the language.

Objectives 4. and 6. aim at expanding the DSL improving the contexts in which it might offer a proper solution to a system's problem.

Objective 5. has the purpose of providing users of the DSL with better tools for the development of ontologies using OntoDL+.

Finally objective 7. provides a way to enable formal static model checking of ontologies.

1.3 RESEARCH METHOD

In order to properly achieve the described objectives of this Master Thesis, the Design Science Research methodology was implemented.

Design Research is an inherently trial and error process, that is comprised of mainly two goals: creation/design and evaluation/research. The main purpose of the design process is to create an artifact, or work, that addresses a problem. The second step, consists in evaluating the created artifacts, analysing and testing them for their utility (Manson, 2006).

In order to ensure that this two main goals are achieved, several steps are to be taken during this research, these consist of (Manson, 2006; Kuechler et al., 2007):

1. Discovery of The Problem

A researcher as to come into contact with a meaningful problem, research system requirements and functionalities, and have a good insight on the related subjects/disciplines.

Most importantly, the researcher as to be fully aware of the problem that is trying to solve.

2. Solution Proposal

The researcher will then have to provide one or multiple suggestions of Tentative Designs. During this step, the functionalities of the different components and relationships among them should be very clear.

3. Development

During this step, the implementation of one or more artifacts will be done in order to achieve the proposed solution.

4. Evaluation

After developing the artifacts, these ought to be tested, and their behavior analysed. It is possible that there are deviations from the predicted behavior. If this is to happen, then the researcher should analyse the problem and the artifacts again, in order to correct the problem.

5. Conclusion

Even if there are possible deviations from the predicted behavior, there comes a time in which the research effort is considered 'good enough' marking the end of the research. At this stage, the achieved results should be documented and the knowledge gained should be classified as being firm (facts that can be applied repeatedly) or as loose-ends which may conduct to further research.

1.4 RESEARCH HYPOTHESIS

During this project it will be shown that it is possible to increase the usability of OntoDL, through the implementation of small changes to the language and its' compiler.

1.5 DOCUMENT STRUCTURE

After analysed the main objectives and motivation of this Master Thesis in Chapter 1, the state of the art will be presented in Chapter 2, with some key concepts and related work.

In Chapter 3, OntoDL was analysed to identify its features and grammar. The liabilities of OntoDL are also shown along this chapter.

In Chapter 4, OntoDL+ proposal is presented, indicating the language extensions and compiler expansions as well as the planned benefits from them. Meanwhile, Chapter 5 presents the development decisions that were taken during this project.

Chapter 6 covers the Website created to publicize OntoDL+, showing the pages of the site and explaining their purpose.

Chapter 7 contains practical examples to test and analyse the results of the development done in this Thesis.

Finally Chapter 8 the conclusions of this thesis are drawn and prospects of future work are presented.

STATE OF THE ART

As previously stated, during this Master Thesis, OntoDL was improved, however there are some concepts and notions which ought to be clarified before advancing any further. Therefore, it will be organized in the following sections:

- An explanation about the concept of ontology and its usages
- Logic Programming languages and their applications in ontology specification
- An explanation about the concept do DSL and some of its advantages and drawbacks
- Software development which covers subtopics such as model checking and code refactoring.

2.1 ONTOLOGIES

An ontology is a formalism used to represent knowledge. In the knowledge engineering context, one of the most cited definitions of ontology is that “*An ontology is an explicit specification of a conceptualization.*” (Gruber, 1993). This was further expanded describing an ontology as a set of definitions associating a term with axioms, constraining its use and relating it to other terms (Falasconi et al., 1994). That being said, in computer science ontologies provide a formalism to represent knowledge, defining terms and describing their relationships, constraints and axioms within the knowledge domain described.

Formally, Serra and Girardi (2011) represent an ontology as the tuple $O = (C, H, I, R, P, A)$ where:

- **C** is the set of terms available within the knowledge domain of the ontology. This set can be described as $C = C_I \cup C_C$ where C_C is the set of concepts, and C_I the set of instances or individuals.
- **H** is the set of hierarchic relations described among concepts.
- **I** is the set of relationships between concepts and instances.

- **R** is the set of all other relationships in the ontology.
- **P** is the set of properties of classes
- **A** is the set of axioms

Based on [Serra and Girardi \(2011\)](#) definition, [Martini et al. \(2016\)](#) proposed as an alternative to define an ontology structure the 4-Tuple $O = (C, I, P, A)$, where:

- **C** is the set of concepts of the ontology. The set is equivalent to the set C_C from [Serra and Girardi \(2011\)](#).
- **I** is the set of individuals of the ontology. The set is equivalent to the set C_I from [Serra and Girardi \(2011\)](#).
- **P** is the set of properties of the ontology. The set can be described as $P = P_{dt} \cup P_O$ where,
 - P_{dt} is the set of data type properties between classes or individuals and atomic values. It is equivalent to the set P from [Serra and Girardi \(2011\)](#).
 - P_O is the set of object properties between classes or individuals and other classes or individuals. Adapted from [Serra and Girardi \(2011\)](#) such that $P_O = H \cup I \cup R$.
- **A** is the set of axioms of the ontology.

OntoDL as adopted the 4-Tuple definition from [Martini et al. \(2016\)](#) for its' DSL implementation. The syntax of the language was implemented creating sections to describe each set of components belonging to the tuple.

2.2 LOGIC PROGRAMMING

Logic programming is a formalism that was introduced by R Kowalski in 1974 ([Kowalski, 1974](#)). It has evolved to become a new programming paradigm with the introduction of Prolog ([Apt, 1990](#)). This is a logic programming language based on predicate logic, being a highly declarative programming language. Despite initially being created envisioning natural language processing ([Apt, 1990](#)), new applications for the paradigm were discovered. Currently, logic programming is also used in the fields of intelligent systems simulation ([Abelha et al., 2007](#)) and knowledge representation ([Lima et al., 2011](#)).

Ontology oriented languages such as OWL2's RDF-syntax can provide a large supply of tools to build and reason over ontologies. These languages, however, have some drawbacks associated, namely: OWL2 typically has complex development environments, these environments are directed to domain experts instead of computer scientists or programmers;

RDF-syntax can be too low level to use with axiomatized OWL2 ontologies (Vassiliadis et al., 2009). Comparatively, logic programming languages offer very simple, direct and intuitive representation of knowledge. Research has already been made into exploring logic programming for ontology specification (Vassiliadis et al., 2009).

The usage of a logic programming language enables the development of reasoning systems over the specified knowledge. Through these, it would be possible to create logic programs containing ontology specifications which would allow searching and querying the ontology. The user can further develop the program and specify verification rules to be applied to the ontology.

Given that logic programming is highly used in the knowledge representation fields, - and given that an ontology is the representation of knowledge of a given universe, - it is considered that this is an appropriate paradigm for ontology representation/specification.

2.2.1 Prolog

Vassiliadis et al. (2009) as expressed some advantages of using Prolog as an ontology processing language for ontology manipulation:

- Prolog programs are sets of *horn clauses* with the form Head :- Body.

Horn Clauses are used in Prolog to represent knowledge. An horn clause with an empty body is regarded as a *fact*. Facts can be used to represent explicit knowledge of the ontology. Horn clauses with non-empty bodies can provide relations through composition of other relations, as well as provide restraints to relations and terms.

- Prolog programs implement *not* using negation-as-failure to achieve the closed world assumption

In a Prolog program, it is assumed that the described knowledge represents all knowledge. If a predicate does not match the represented knowledge in the program, it is considered to be false.

- Prolog offers non-logical predicates such as *cut*

The *cut* predicate can be used to prune a search tree in prolog. Using *cut* it is possible to stop the backtracking process of a query, enabling a run-time restriction of solutions.

- Prolog offers meta-logical predicates to perform aggregate operations.

Aggregation operations such as finding all answers to a query are possible in Prolog.

By using Prolog it is also possible to use a querying system, enabling reasoning over the specified information. It is also possible to create verification queries in order to ensure that

the system follows desired rules. Therefore it is considered that Prolog is a language that enables ontology specification.

2.3 DOMAIN SPECIFIC LANGUAGES

As the name implies, a DSL differs from a general purpose language by being designed and implemented for a particular domain. Unlike General Purpose Language (GPL) that should be able to address every problem, regardless of the complexity and effort needed to implement a solution, by design, a DSL will not be able to do that, neither are they supposed to be able to do it (Kolovos et al., 2006; Van Deursen et al., 2000). So, in order to atone for this lack of generality, a DSL offers a lot of expressive power through notations and abstractions to increase the productivity of it's users. As previously stated, their expressive power and application are usually restricted to a particular domain problem (Van Deursen and Klint, 2002).

Languages such SQL, T_EX, LEX and YACC are some examples of well known DSLs (Van Deursen et al., 2000). These languages offer a good solution and great expressive power regarding, relational database management, document generation and parser generation (both LEX and YACC) respectively. One would be incapable of using these languages alone to solve every problem. Yet, as tools to solve problems regarding their particular domains, these DSLs are good options to be considered.

Therefore, generally speaking, DSLs will offer both advantages and disadvantages. It is important to keep in mind these benefits and drawbacks attached to the usage of DSLs. That is important in order to accurately determine whether or not it is worth using DSLs, as well as ensure an adequately designed DSL, correctly balanced regarding the advantages and disadvantages it provides.

2.3.1 Advantages of DSLs

As previously stated, it is important to determine the advantages of DSLs: (Van Deursen et al., 2000)

- **DSLs allow the design of solutions with a proper level of abstraction and idiom for the problem domain.**

DSLs can be used to adequately represent the concepts and constructs of a particular problem domain. Furthermore, these can be represented in the language adopted by the users of the DSL, enabling domain experts to use the DSL.

The creation of an higher level language also decreases the learning curve of the language for new users.

- **DSL programs allow the creation of more concise and self-documenting code, while improving re-usability.**

The usage of DSLs with an appropriate level of abstraction and available statements can significantly improve documentation. Higher level languages, closer to the level of natural language are easier to analyse and properly understand. If these abstractions and statements are appropriately defined the code produced will be shorter, and there will be no need to produce natural language documentation (Ladd and Ramming).

- **DSLs have the potential to increase productivity, reliability, maintainability, and portability.**

DSLs have been used to reduce the production time of products from an average of months, to weeks (Deursen and Klint, 1998), other cases of productivity being enhance several times is also talked in Kieburtz et al. (1996); Herndon and Berzins (1988).

Automatising code generation tasks through DSLs also enables a better understanding of modifications made to the language, and their impact, as well as the verification of the code generated, improving the overall reliability in the DSL (Deursen and Klint, 1998). The amount of times that a module has to be tested before it's considered finished may also be a factor to consider in the reliability of the generated code. If application modules have to be rewritten a lesser amount of times due to the usage of DSLs, it may be an indicator that the application writers have a better understanding of the code due to the appropriate abstraction level. Therefore, reducing the number of failed programs also increases reliability in the programs produced, which was the case verified by Kieburtz et al. (1996).

DSLs can further improve maintainability of costs of the applications (Deursen and Klint, 1998; Herndon and Berzins, 1988) and improve it's portability (Herndon and Berzins, 1988).

- **DSLs explicitly declare domain knowledge, enabling its conservation and reuse**

The knowledge generated through a DSL can be reused by different applications (Deursen and Klint, 1998).

- **DSLs can improve validation and optimization.**

Code generation DSLs also have the advantage of being able to introduce optimizations to the generated code, while simultaneously allowing the introduction of validation constraints in the language (Menon and Pingali, 1999; Basu et al., 1997; Bruce, 1997).

2.3.2 Disadvantages of DSLs

Now, having the advantages of DSLs in mind, we should also analyse the setbacks of implementing and using a DSL: (Van Deursen et al., 2000)

- **The overall costs in the design, implementation and maintaining the DSL.**

Despite the advantages, every DSL as an initial cost of designing and implementing it. Furthermore, maintaining and extending DSL can sometimes present more unexpected costs and difficulties (Deursen and Klint, 1998).

- **Appropriate scopes for the DSL may be hard to find.**

Finding a DSL for the correct domain, or with the proper abstractions presented for the task at hand may sometimes be difficult.

The usage of a DSL for a prolonged period of time may reveal some flaws in it, propelling the necessity of extending the language (Deursen and Klint, 1998).

- **The costs of educating DSL users.**

Despite being designed to be more appropriate for the development under the right domain, its users will always have to learn how to write Domain Specific Descriptions (DSDs), which comes with a cost.

2.3.3 DSL Design and Implementation

When designing and implementing a DSL, there are various techniques which can be used. They will commonly follow already established patterns, and some work has been done to try to establish such patterns (Spinellis, 2001; Gamma et al., 1995; Mernik et al., 2005). Among these, it is important to highlight:

- **Piggyback pattern** (Mernik et al., 2005; Spinellis, 2001), consisting of making use of an existing language within the DSL, to enable desired features from it. Therefore, the DSL uses an existing base language as an implementation tool. The compilation of such DSL specification will convert the DSL components to the base language, while leaving the base language code unchanged. ANTLR is an example of a DSL that follows such pattern. Being a DSL for grammar specification, ANTLR supports the use of Java to define semantic actions and attributes.
- **Language specialization pattern** (Mernik et al., 2005; Spinellis, 2001), in which an existing language is changed, removing features in order to restrict it. This pattern can be used with objectives such as: creating a safe DSL by restricting unsafe features

from an existing language, or improving the learning curve or usability by removing unneeded elements. OWL-Light is an example of a DSL following a specialization pattern.

- **Language extension pattern** (Mernik et al., 2005; Spinellis, 2001), occurs when a language is changed to add domain specific features. Typically the existing language features will remain available. This pattern differs from the piggyback pattern, since the piggyback uses a base language as an implementation tool for certain features, while the language extension pattern will change syntactic and semantic aspects of the base language to adapt it to the domain problem.
- **Source to source transformation pattern** (Mernik et al., 2005; Spinellis, 2001), in which the DSL code is entirely translated into an equivalent source code in a different language. This approach enables the use of tools provided by the target translation language.

2.4 SOFTWARE DEVELOPMENT

In software development, systems tend to evolve and, frequently, increase their complexity. With complexity increase, system comprehension and error correction also become harder. It's been estimated that over 80% of software costs derive from system evolution and maintenance (Ouni et al., 2016).

Some of the more frequent software development problems include: implementation problems - like inappropriate documentation and use of global variables (Stewart, 1999), long methods, etc., - and design problems - resulting from lack of modelling (Stewart, 1999) or model checking. To address some of these problems multiple techniques have been created to assist the process. In order to address implementation it's necessary to highlight code refactoring techniques as possibly some of the most used. However, formal techniques such as model checking, can aid in detection of system specification errors, increasing system robustness.

2.4.1 Code Refactoring

Code refactoring is a technique used to facilitate software maintenance tasks, and it improves software architecture or implementation, without changing the behavior of the program. These techniques are used to facilitate code manipulation improving comprehension reuse and future evolution.

The refactoring process starts with identifying the code portions to be refactored, typically referred to as AntiPatterns (Brown et al., 1998), Bad Smells (Mantyla et al., 2003; Murphy-

Hill and Black, 2010; Beck et al., 1999) or design defects (Ouni et al., 2016). They are code patterns that often occur in software implementation which increase difficulty in maintaining, comprehending and evolving the system. Detecting a smell does not necessarily mean that something is inherently wrong in the program, but such sections usually request some special attention, - either because they could likely be rewritten in order to be clearer and more efficient, or because they are more likely to be at the source of errors.

Following the smell identification, it should be determined which refactoring method to apply, and refactor the code, ensuring behavior preservation (Ouni et al., 2016). This can improve readability, performance, extensibility or other metrics. The same technique can improve various code analysis metrics simultaneously, but often it is necessary to compromise some metrics to benefit others.

After applying the technique, it should be evaluated its effect on the program, through comparison of metrics before and after the refactoring process is done. It is also important to ensure that the program retains consistency, updating documentation, tests, and other needed documents not included in the refactoring process (Ouni et al., 2016).

Bad Smells

A lot of work has been done in order to develop a taxonomy which allows identification and handling of various types of smells (Ouni et al., 2016; Mantyla et al., 2003; Murphy-Hill and Black, 2010; Brown et al., 1998; Sharma and Spinellis, 2018; Beck et al., 1999). Among these we can highlight some of the more frequent smells like:

– *Blob*

- Definition

The Blob (Brown et al., 1998; Ouni et al., 2016) also referred to as *God Class* (Sharma and Spinellis, 2018), is found in systems where a single class is responsible for the majority of the computations of the system. Typically the Blob possesses an abundance of methods, attributes or both and represents a controller class of the system (Brown et al., 1998; Sharma and Spinellis, 2018). Meanwhile, classes other than the Blob will primarily store and provide data with little to no processing responsibilities of their own (Ouni et al., 2016; Brown et al., 1998).

- Consequences

The Blob generally is accompanied by low cohesion of methods and data (Sharma and Spinellis, 2018). Most times, the Blob behaves like a procedural main program, compromising the advantages of an object-oriented design (Brown et al., 1998). Because of its excessive complexity, the Blob is hard to evolve and maintain. Its reuse and testing is often too complex to be viable as well.

- Solution

Brown et al. (1998) has proposed that to deal with the Blob, one should start by identifying groups of methods and attributes with a common behavior or feature. For instance, if a store system possesses most operations in a controller class, trying to identify which operations are related to customers and which are related to products. Following the identification of groups of functionalities, they should be migrated to their appropriate class, - i.e., methods relating individual customers should be migrated to the customer class, and so on. Finally any redundant associations should be removed.

- Comments

- Definition

Comments (Beck et al., 1999; Mantyla et al., 2003; Sharma and Spinellis, 2018; Murphy-Hill and Black, 2010), inherently are not a bad thing, however, thickly commented blocks of code often try to hide bad code (Beck et al., 1999). That usually indicates that one or more different bad smells may exist in that portion of the program, refactoring tends to render most of those comments irrelevant (Mantyla et al., 2003; Beck et al., 1999; Murphy-Hill and Black, 2010).

- Solution

Trying to identify other smells hidden by the comments smell, and refactor the code according to those new smells. Afterwards, some of the comments can be deleted.

- Data Class

- Definition

The Data Class (Ouni et al., 2016; Mantyla et al., 2003; Sharma and Spinellis, 2018; Beck et al., 1999) is a class with no processing responsibilities, containing only data. Often the only methods of these classes are comprised of getting and setting methods.

“Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility.” Beck et al. (1999)

- Consequences

If a class is important enough to be created, then at some point it will have to be manipulated far too much by other classes (Beck et al., 1999). Therefore, the origin of errors resulting from inappropriate manipulation or usage of the information of those classes might be much harder to track down, - making it harder to debug possible errors.

If a Data Class is manipulated by many different methods or classes, chances are, there will be repeated code spread among those classes/methods. Therefore, comprehension and evolution of the system might become harder.

- Solution

Find methods and classes that manipulate the information of the Data Class and either: try to move those methods to the Data Class; or if impossible, extracting the manipulation code into a different method and moving it to the Data Class.

– *Lava Flow*

- Definition

The Lava Flow (Brown et al., 1998; Sharma and Spinellis, 2018) or *Dead Code* (Brown et al., 1998) smell is comprised of a piece of code which nobody understands, no one exactly knows if it is used or not, so it just stays immovable in order to avoid bugs.

“Sometimes, an old, gray-haired hermit developer can remember certain details, but, typically, everyone has decided to “leave well enough alone”” Brown et al. (1998)

- Consequences

If Lava Flows are not removed at an early stage, they can propagate to other sections through code reuse (Brown et al., 1998).

The code becomes harder to analyse, verify and test (Brown et al., 1998).

Loading Lava Flow code into memory can be expensive, reducing performance and consuming computational resources.

- Solution

The best solution for Lava Flow is to prevent it entirely by properly planning the system architecture. If a Lava Flow is already in place the first step is to define a clear architecture. Such step should already help to identify unused lines of code allowing to remove them safely. Removing suspected dead code may introduce errors, but those should be properly analysed and understood, in order to avoid the creation of more dead code to solve it. (Brown et al., 1998)

– *Long Method*

- Definition

The Long Method (Murphy-Hill and Black, 2010; Mantyla et al., 2003; Sharma and Spinellis, 2018; Beck et al., 1999) smell is pretty self explanatory. In object oriented languages, using short methods pays off (Beck et al., 1999) methods to long usually indicate a lack of task delegation.

- Consequences

Long methods become harder to understand and evolve. Testing and debugging may also become more difficult with programs suffering from long methods.

- Solution

This smell can be solved by extracting portions of code to a different method and naming it appropriately (Beck et al., 1999).

– *Long Parameter List*

- Definition

The Long Parameter List (Mantyla et al., 2003; Sharma and Spinellis, 2018; Beck et al., 1999) smell identifies methods that receive long lists of parameters. In object oriented contexts, a method does not need to receive every piece of data it needs in the form of parameters. Instead, the method only needs to receive the appropriate data in order to allow the method to fetch additional information from other methods or classes.

- Consequences

Much like the Long Method smell, the Long Parameter List reduces ease of comprehension of the method, they rapidly become inconsistent and hard to use (Beck et al., 1999). Because it stems from inappropriate data transfer to the method in the form of parameters, evolving the method may be synonymous with adding more parameters, further reducing usability.

- Solution

Most times, the smell can be fixed by passing parameters with objects that can access the intended data, instead of passing the data directly (Beck et al., 1999) - since it is highly likely that there will be objects which can request multiple of the necessary data fields.

– *Shotgun surgery*

- Definition

The Shotgun Surgery (Ouni et al., 2016; Mantyla et al., 2003; Sharma and Spinellis, 2018; Beck et al., 1999) smell relates to methods or classes which are used by a large amount of external methods. Therefore, when making changes to the method of class, a lot of external methods or modules will have to be revised and tested again, - you take a change that you want to make on your system, load it on your shotgun, aim it at the intended method or class, and after firing the change, a lot of classes will be hit by shrapnel.

- Consequences

When changing the method presenting the smell, changes will occur in a lot of different classes, increasing the risk of missing some changes causing errors. Evolution and debugging of the system becomes harder.

- Solution

The solution is to identify groups methods and fields affected by the change, and try move them to a single class -, creating one if needed. (Beck et al., 1999)

– *Spaghetti Code*

- Definition

Spaghetti Code (Ouni et al., 2016; Sharma and Spinellis, 2018; Brown et al., 1998) defines a code block without any clear structure. It is typically resultant from procedural thinking while programming in object-oriented paradigm (Ouni et al., 2016).

- Consequences

Spaghetti Code highly undermines object-oriented mechanisms such as polymorphism and inheritance (Ouni et al., 2016; Brown et al., 1998; Sharma and Spinellis, 2018). Maintenance costs also increase and code reuse is very difficult. If the problem is not addressed eventually there will be a time when maintaining the code is more expensive then create an entirely new solution (Brown et al., 1998).

- Solution

In order to refactor a block of spaghetti code, one should start by removing direct accesses to variables of classes -, replacing them by getting and setting methods. While analysing the code, identify small processes that can be extracted to smaller methods, easier to use and understand. Remove unused portions of code, since it improves readability of the program and prevents the appearance of Lava Flows. Ensure that the naming system of classes, functions or data types, follows a system standard. (Brown et al., 1998)

– *Switch Statement*

- Definition

The Switch Statement (Mantyla et al., 2003; Sharma and Spinellis, 2018; Beck et al., 1999; Murphy-Hill and Black, 2010) smell refers to switch statement on type codes which are duplicated throughout many code sections. It doesn't make use of polymorphism, and when you have to make any change to the switch, you have to find every instance where it is used to update it.

- Consequences

Evolution of the system might become slower where changes to the switch are involved, due to the need to find and replicate the change across the code.

- Solution

A solution can be achieved by extracting the switch statement onto a method and moving the method to an appropriate class. Depending on the circumstances, it may also be appropriate to replace the type codes with subclasses making use of hierarchy and polymorphism. (Beck et al., 1999; Mantyla et al., 2003)

2.4.2 Model Checking

In software development, system modeling should be done at an early stage. By specifying the system and its' properties it is possible to identify and fix design flaws with more efficiency and reliability - after system implementation, changing its' design might mean multiple changes to several modules of the system, making the process slower and more complex. By using model checking tools and techniques it is possible to better comprehend and reason about the system to be developed.

Through system model checking, it's possible to make both static and dynamic verification of the system. Static verification allows a structural verification of the system, ensuring that it is correctly built. Dynamic verification allows behavioral analysis of the system, verifying the system's interactions and its' evolution over time -, also verifying that the properties remain valid at every stage of execution(Macedo et al., 2016).

The more successful model checker provide a simple yet expressive and flexible language. Typically static modeling relies formal languages based on first order logic, while dynamic modeling relies on languages base on temporal logic. Furthermore, most model checkers focus mainly on only one of those verification strategies, - static or dynamic. Nevertheless, there are some model checkers which allow for a simultaneous static and dynamic verification, given however, that extensions are used. Using the Electrum extension for Alloy, it's possible to simultaneously perform structural and dynamic verification of the system (Brunel et al., 2018).

Performing static model checking on an ontology, would be equivalent to verify a specification of the concepts, relations, inference rules and restrictions defining the ontology -, ensuring the preservation of those rules. By statically verifying an ontology, it is possible to check if the specification is adequate to represent the intended domain knowledge. Applying dynamic verification to an ontology would involve specification of the system that the ontology integrates, along with its actions and effects. It is intended to analyse processes allowing to ensure that the system holds certain properties over time.

Alloy

Alloy is a specification language based on first order logic and relational calculus (Dennis and Seater, a). Using the language it is possible to specify models along with its rules and restrictions. The language is highly declarative and it enables both static and dynamic modeling. However, without the use of extensions, an Alloy specification should focus only on one type of modeling. That means that in order to properly verify a system statically and dynamically, two specifications are advised. For example, when model checking an ontology, each static model will represent a possible state of the ontology, which is allowed by the specification written, while its' properties will represent invariants (Dennis and Seater, c) - it's appropriate to verify structural correctness. However, each dynamic model will represent transitions between states of the ontology, while its' properties represent interactions with the system (Dennis and Seater, c) - it's appropriate to verify processes which interact with the ontology.

Currently, Alloy is used mostly to model systems (Cunha et al., 2015; Macedo et al., 2016; Ferreira and Oliveira, 2009), however, it's considered that the language can also be used to model ontologies. When statically specifying an ontology, given that Alloy *signatures* are sets of atoms with common features, - which allow the definition of hierarchy among them, - they are adequate to represent the classes of the ontology. Similarly, the fields would be used to represent the ontology's relations, along with domain, counterdomain and cardinality. Properties can also be defined to specify invariants. During dynamic specification of an ontology, the ontology is specified according to the system it integrates. In an ontology dynamic model each atom would represent a state of the ontology -, or system to which the ontology belongs, - while properties are used to represent actions. (Dennis and Seater, b)

Using Alloy Analyzer, it's possible to verify if there is any inconsistency that prevents the creation of models. When a specification poses no errors, the analyzer will create and populate models according to the established signatures and relations. It's also possible to create predicates and assertions to define additional desired system properties and verify if they are respected, or force the creation of new models following the properties. The created models can then be graphically viewed and styled using the analyzer, allowing for a better understanding of the specified ontology.

2.5 SUMMARY

Along this chapter, the concepts of Ontology, Logic Programming, DSL and Software Development have been explored. The formal ontology structure used by OntoDL was explained. Programming languages have been analysed in the context of ontology manipulation languages. The advantages, disadvantages and some of the best practices regarding DSL development have been described. Finally, the main concepts concerning software develop-

ment have been presented, identifying good practices such as model checking, as well as common bad practices present in software and how to solve them.

ONTODL

OntoDL is a DSL created to specify ontologies. The language was created using ANTLR4 and aims at being simple and intuitive. Therefore, the main purpose of the language is to offer an easy to learn syntax, requiring little to no previous knowledge to ensure its usability and correction. The DSL also tries to improve productivity of developers using the language. Comparatively to the syntax of other ontology specification languages, such as the RDF-Schema or OWL, OntoDL improves productivity by defining specific sections for each group of declarations. This way, the language becomes structurally more rigid, however, there are benefits to that strictness such as:

- **Ease of access to declarations:** OntoDL groups declarations with similar character, i.e., there are sections destined to the specification of information of certain entities. The classes/concepts, used in a given ontology will all be specified in the same section. The same principle applies to the relationships, triples and individuals to be used. Said separation by sections eases the burden of analysing, growing, debugging and maintaining the ontologies developed. This way, tasks that involve declarations of certain specific components of the ontology (such as classes or relationships of the ontology for instance), become much easier and non-fallible. That is due to the close proximity of said declarations, being that they will be enclosed in a well defined region of the specification. The same can't be said of languages like OWL which, despite giving more freedom to users in regards to the order of declarations by enabling an arbitrary order of declaration, also makes it harder to search, analyse and update said declarations over time.
- **Self-explanatory syntax:** The separation of the different sections of declarations is well distinct. The terminal symbols and reserved words used to delimit the borders of the sections were defined in a clear manner, which will in turn point the different sections of the DSL where declarations should be included.

- **Reduction of the necessary source code:** Through the usage of OntoDL it is possible to considerably reduce the necessary code to specify ontologies. Much of the redundant syntax used in other languages can simply be concealed due to the language design.

OntoDL also has a compiler to process the language files, recognizing the declared information and detecting errors. If a specification possesses no errors, OntoDL's compiler will create two output files, one of which possessing the same specification written in OWL, and the other possessing a DOT specification to enable the creation of an ontology graph.

3.1 GRAMMAR

The language was created using Portuguese as the idiom to attribute names to the non-terminal symbols. Despite the good choice of names, easing the burden of analysing and comprehending the grammar to Portuguese people, it is also quite limiting. That is due to the fact that developers that do not know how to read and write in Portuguese will encounter unnecessary development challenges when working with the grammar.

The ontology's starting production is *ontologia*, showed in listing 3.1. In it, it's possible to distinguish four primary sections to be used in ontology specification, those being: concepts, individuals, relationships and triples. It's important to note that in OntoDL, there is no section uniquely oriented to the declaration of data properties.

```
ontologia : 'Ontologia' TEXT
           conceitos
           individuos?
           relacoes
           triplos
           ','
           ;
```

Listing 3.1: Production *ontologia*

It's also possible to observe that, such as the non-terminal symbols, some of the restricted words used also have Portuguese as its' idiom. This limitation might reduce the potential users of the DSL to only Portuguese speaking users. It's also possible to note that the idiom used in name attribution is inconsistent, given that terminal symbols are written using English as its' idiom while non-terminals and reserved words use Portuguese.

It's possible to declare data properties/attributes to be associated with concepts during concept declaration in an ontology, because of the productions in listing 3.2. Said attributes should be associated with individuals instantiated as part of the defined class. However, this type of declaration is not optimal, since should two or more concepts possess the same

attribute, the name and type of the attribute will have to be declared multiple times. In said situations, given that the attribute is the same, its' type should always be the same as well, but the current syntactic structure creates redundant code, diminishing productivity and increasing the likelihood of errors by enabling the declaration of the same attribute with different types.

```

conceitos  : 'conceitos'
           '{' c1=conceito
           (',' c2=conceito)*
           '}'
           ;

conceito   : TEXT
           atributos?
           ;

atributos  : '[' a1=atributo
           (',' a2=atributo )*
           ']'
           ;

atributo   : atr=TEXT ':' tipo=TEXT
           ;

```

Listing 3.2: Concept related Productions

The individuals specification section, listing 3.3, is well design when concerning code reduction and productivity increase, however some improvements could be made. In the grammar design, it is uncertain the purpose of the symbol TEXT?, given that the semantic actions of its' production never make use of it. As there is already a terminal symbol dedicated to comment capture, it is possible to exclude comments as a possible use of said terminal. As the semantic actions also ignore the symbol entirely as well, it seems to be no more than a mistake in the grammar design. Said mistakes eases the appearance of mistakes such incomplete ontologies that do not use the proper separators in individual declaration, causing in turn the creation of translated ontologies that do not contemplate all of the declared individuals.

```

individuos : 'individuos'
           '{' TEXT TEXT?

```



```

    ( ' , ' TEXT TEXT? ) *
    ' } '
;

```

Listing 3.3: Production individuos

The relationships specification section, listing 3.4, is very well designed regarding its' simplicity, however it's not as descriptive as OWL defined relationships. That's due to the fact that it's possible to declare more information regarding relationships in OWL than in OntoDL. For instance, it is possible to declare relational cardinality in OWL, which can not be done in OntoDL. Despite being an optional feature, it might be important in situations where it is necessary to ensure the structural correction of the ontology. Therefore, it is considered that the relationships specification area is still incomplete.

```

relacoos : 'relacoos '
          '{ ' TEXT
          ( ' , ' TEXT ) *
          ' } '
;

```

Listing 3.4: Production relacoos

Finally, the triple declaration section, listing 3.5, also possesses some interesting features, such as enabling the usage of some relationships without requiring its' previous declaration. These however, are hard-coded in the grammar, making it impossible to use any other relationship without its' previous declaration. That means that the usage of external vocabularies, such as the usage of relationships, concepts or data properties from a vocabulary, - for example the FOAF vocabulary (Brickley and Miller, 2014), - is also impossible. Said behavior reduces the potential usages of OntoDL, therefore, some companies and organizations might choose to not use the language. Even so, by design, this is the desired behavior for the language, given that it's mainly focused on an institutional and educational usage. It is desired that the users make its' own specifications and understand them, instead of using external ontologies specified by someone else, be those in OWL or other language different from OntoDL.

```

triplos
: 'triplos '
  '{ ' triplo
  ( ' ; ' triplo
  ) * ' } '
;

```

```

triplo :
    '(? c1=TEXT '= ( 'iof'|'instancia' ) '=>' c2=TEXT atribInsts? )'?
  | '(? c1=TEXT '= ( 'pof'|'parte' ) '=>' c2=TEXT )'?
  | '(? c1=TEXT '= ( 'is -a'|'subclasse' ) '=>' c2=TEXT )'?
  | '(? c1=TEXT '= ( 'synonym'|'sinonimo' ) '=>' c2=TEXT )'?
  | '(? c1=TEXT '= rel=TEXT '=>' c2=TEXT )'?
    ;

atribInsts
    : '[' atribInst
      ( ',' atribInst )*
      ']'
    ;
atribInst : atrib=TEXT '= valor=TEXT
    ;

```

Listing 3.5: Triple Related Productions

The usage of hard-coded declared relationships in the grammar design provides other advantages, such as semantic verification of the relationships offered by the language. Additionally, the declaration of data properties of individuals, can only be done during the instantiation of an individuals' class. That in turn eases comprehension of the ontology, given that by grouping relevant information regarding an individual, the specification can be translated to natural language in a very intuitive manner.

```

brg = iof => City [description='Archbishops' city',
                  district='Braga',
                  name='Braga',
                  population='140000'];

```

Listing 3.6: Triple Example

As we can see in listing 3.6, the declared triple will be read in natural language as “brg is an instance of City whose description is Archbishops' city, the district is Braga, the name is Braga and population is 140000”. Given the close proximity of the DSL declaration with natural language, its' expected to fair well in educational contexts. It also ensures the necessary and sufficient vocabulary to ensure the declaration and comprehension of ontologies, as well as the knowledge contained in them, making it in a clear and intuitive manner.

3.2 SEMANTIC ACTIONS

Despite providing the intended processing of information, the semantic actions of the language are very poorly structured and disorganized, making it harder to analyse and comprehend. In them it is possible to observe two primary problems: excess of inherited and return attributes in various productions, with unclear names regarding its' usage - Long Parameter List 2.4.1; to extensive to comprehend and analyse semantic actions - Long Method 2.4.1. This problem can be verified in a generalized manner throughout the grammar source file. These are design and not implementation errors and many of them repeat themselves in only slightly different variations in different productions. Therefore, it will not be done an individual documentation of every problem in the productions, instead a general overview of the underlying problems will be done. To such an end, one of the productions with more of these problems will be analysed: the production of the symbol `triplo`.

```
triplo [HashSet<String> concts , HashSet<String> indivs , HashSet<String> rels
      , HashMap<String , List<String>> tabConcsin , String outputDOTin , String
      outputOWLin]
      returns [String trip , boolean erroTriplo , HashMap<String , List<String>>
              tabConcsout , String outputDOTout , String outputOWLout]
```

Listing 3.7: Triple Production Attributes

Beginning by the initial declaration of `triplo`, listing 3.7, it's possible to detect the Long Parameter List smell. The production inherits six different attributes, four of those attributes were initially declared in the grammars' starting production (`ontologia`). The other two attributes represent input texts to be altered to produce output texts. The return of the production suffers the same problem with excessive attributes. Being there a control error Boolean, two output strings created from the input strings, a String containing the triples' information and finally an output concept table. At a first glance, the output concept table seems to be created from the input concept table. Further analyses of the attributes revealed that:

1. The output HashSet is unrequired

As stated, at a first glance the output HashSet `tabConcsout` seems to be created from `tabConcsin` as a different object. However, the semantic actions reveal that at any moment those two objects represent different things. Never at any moment the object `tabConcsin` is cloned, therefore the information present in the two objects at the end of the productions' semantic actions is exactly the same. Such fact is easily highlighted by observing the last semantic action present in each of the `triplo`'s productions, that being exactly:

```
$triplo.tabConcsout = $triplo.tabConcsin;
```

So, at the end of performing the semantic actions, `tabConcsout` is always the same as `tabConcsin`. Therefore the usage of two different variables with different names pointing to the same object is a poor practice that hardens comprehension. A single object `tabConcs` would be enough.

2. The input and output Strings represent unnecessary computational work at the moments used.

The input Strings have the ontologies' knowledge recognized to the point, converted to both DOT and OWL. These however, are in constant growth and represent duplicated information, while its' usage will only come to be necessary at the end of the ontology recognition. Furthermore, they will only be used in case that there are no semantic errors in the ontology, worst case would be a single semantic error at the last triple of the ontology. All the excess storage used to store the information in the proper syntax of the languages to be translated would only amount to the memory of the computer and computational resources used to create them being wasted. At the end, all the generated strings would be discarded though to a single semantic error. If the information was properly stored within objects destined to receive it, a lot less resources would be used in information storage. And only after ensuring that there are no mistakes in the ontology description would the computational resources be used to create the output Strings.

3. All the inherited HashSet could be represented using a single object.

The four HashSet represent all of the ontology information retrieved until the triple recognition. That said, the inheritance of four objects might be confusing when a single object `Ontology` would be able to represent all the required information.

```
triplo
@init { List<String> lstAtribs = new ArrayList<String>();
        int numAtribsPrev=0; int numAtribsReal=0;
        Boolean erroAtribs = false;}
```

Listing 3.8: Triple Production Initial Variables


```

rel.text + "\"" /i\n\t\t<Class
IRI=\"#" + $c2.text + "\"" /> \n
\t</ObjectPropertyRange> \n";
}
}
$triplo.tabConcsout = $triplo.tabConcsin;
}

```

Listing 3.9: Triple Production, Semantic Actions

The production has 3 distinct moments in its' semantic actions: error control code (painted red), triple information storage (painted black) and output generation code (painted blue). These three moments could be easily extract onto external methods easing code analyses and comprehension, as well as debugging and error correction. Creating objects to store information as previously suggested would also present be advantageous in the correction of this design error, allowing a cleaner separation of methods. All the blue code could also be extracted onto a method that would only be executed at the end of the ontology recognition, ensuring that no resources are unnecessarily wasted. Naturally the same could be done regarding the error control code, given that many of the semantic checks are common to various different productions of the symbol.

The comprehension of the semantic actions and tests performed, as well as bug correction would be much easier by structuring the grammar in order to take advantage of objects created to store and process information present in the ontology specification.

3.3 FEATURES

File recognition is not the only task of OntoDLs' compiler, the language gains usability through the features implemented by its' compiler. These are a central part of the DSL, serving two main purposes: analyses and semantic error detection in the ontology specification, as well as conversion of the analysed information.

The semantic error detection is done throughout the ontology recognition. If any semantic error is detected in the specification, data conversion will not follow throw. Even so, to ensure a smooth development cycle, the compiler ensures that it's possible to detect all the semantic errors of an ontology through a single compilation. That way it's possible to avoid a repetitive cycle of compilation and error correction to ensure semantic correction. The checks performed to the ontology ensure that:

- **There are no multiple declarations of ontology terms or triples.**

During the ontology reading process, all the concepts, individuals, relationships and triples are analysed to check if the element was previously registered. An error will be generated in case of multiple declarations of the same element.

- **It's not possible to used unrecognized terms in triple specification.**

During triple specification, all the different terms belonging to the triple are verified, to ensure they were previously declared in the ontology. It is important to note that the relationships *is-a*, *synonym*, *pof* and *iof* are the only exception to this rule. Given that they are relationships provided by the language without requiring its' previous declaration.

- **Hard-coded relationships are coherent.**

For the four relationships provided by the ontology, additional semantic checks are performed. These ensure that the terms used were not only declared, but also of the correct type. In the usage of the *synonym* relationship both the left and right side terms should be concepts or both should be relationships. It is not possible to declare a concept as *synonym* of a relationship or vice versa. The usage of individuals along with the *synonym* relationship is also forbidden. The *iof* (*instance of*) relationship ensures that the left term is an individual and the right term is a concept, further ensuring that all the data properties were duly specified. The *pof* (*part of*) relationship ensures that both terms are concepts, or both are individuals. And the *is-a* relationship ensures that both terms are concepts.

After the specification file recognition, if there are no syntactic or semantic errors, OntoDLs' compiler will generate two output files. These will contain the same ontology, but specified in OWL and DOT. That enables the usage of the ontology in external tools that support OWL and its' graph visualization through the DOT file compilation.

3.4 SUMMARY

Along this chapter, the OntoDL was analysed. The grammar specification was introduced and explained. The semantic actions were also explored, exposing some of the key bad practices used during the implementation. Lastly, the features provided by the language were also described. The possible limitations of the language were also identified to allow its further improvement.

ONTODL+, SPECIFICATION

In this chapter, the several objectives of ONTODL+ are analysed, presenting a more detailed overview of the problem at hand, and planned solutions.

4.1 SYSTEM ARCHITECTURE

In order to accomplish the established goals, the following architecture has been designed for OntoDL+:

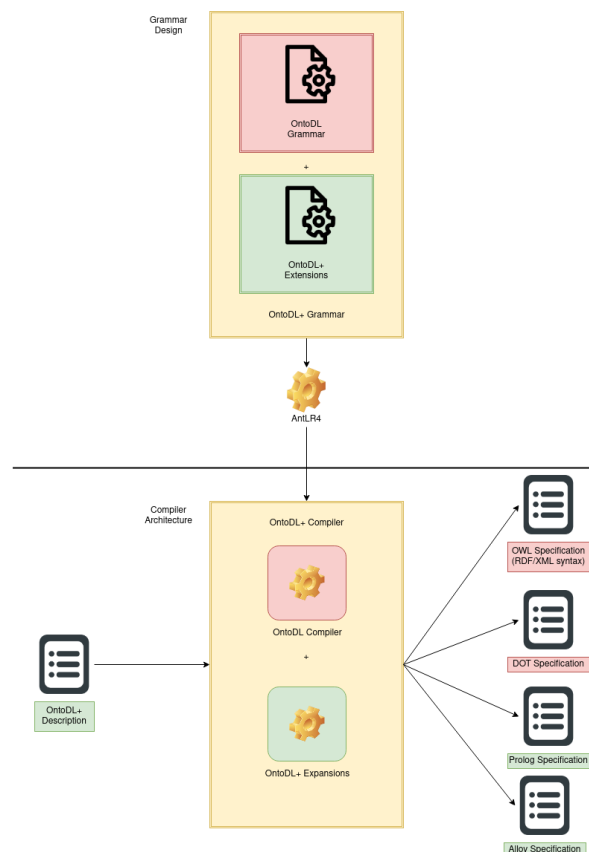


Figure 1: System Architecture

The architecture of OntoDL+ can be split into grammar design and compiler architecture. The grammar design encompasses the syntax and reserved vocabulary as well as the semantics provided by the domain specific language. The OntoDL+ grammar design is the result of extending OntoDL. The grammar design is written in ANTLR4 and after processed, it will generate an OntoDL+ compiler. The compiler architecture addresses the processing of the domain specific descriptions written in OntoDL+ and its' translation into other specification languages. The OntoDL+ compiler is an expanded version of the OntoDL compiler.

The color scheme serves to represent the differences and similarities among OntoDL and OntoDL+. The red boxes represent features already achieved through OntoDL. The green boxes represent the target goals implemented by OntoDL+, being that, OntoDL+ also provides all the features already present in OntoDL.

4.2 LANGUAGE EXTENSIONS

Several language extensions were planned to implement in the language. These extensions address some of the issues with the previous DSL, OntoDL.

4.2.1 Triples grouping

Previously, each triple existent in an OntoDL ontology has to be declared as a separate triple. This generates a lot of redundant syntax when dealing with individuals and concepts with a lot of relations/properties.

An ontology being currently developed using OntoDL, showed in listing 4.1, provides a great example of this flaw.

```

ano2 =desenvolve=> Pensamento_Computacional;
ano2 =desenvolve=> Raciocinio_Logico;
ano2 =desenvolve=> Analise;
ano2 =desenvolve=> Abstracao;
ano2 =desenvolve=> Exploracao;
ano2 =desenvolve=> Criacao;
ano2 =introduz=> Decomposicao;
ano2 =introduz=> Variavel;
ano2 =introduz=> Depuracao;

```

Listing 4.1: Triple Declaration Example

OntoDL+ provides a more simplified syntax, by removing redundant syntax. As a first approach OntoDL+ enables the user to omit repeated entities. This measure will provide less textual syntax such as shown in listing 4.2.

```

ano2 =[desenvolve=> Pensamento_Computacional;
       desenvolve=> Raciocinio_Logico;
       desenvolve=> Analise;
       desenvolve=> Abstracao;
       desenvolve=> Exploracao;
       desenvolve=> Criacao;
       introduz=> Decomposicao;
       introduz=> Variavel;
       introduz=> Depuracao;
      ]

```

Listing 4.2: Triple Declaration in OntoDL+

That being done, OntoDL+ also removes even more redundancy by allowing the omission of repeated relationships. With this change OntoDL+ will provide the syntax presented in listing 4.3.

```

ano2 =[desenvolve=> Pensamento_Computacional ,
       Raciocinio_Logico ,
       Analise ,
       Abstracao ,
       Exploracao ,
       Criacao;
       introduz=> Decomposicao ,
       Variavel ,
       Depuracao;
      ]

```

Listing 4.3: Triple Declaration in OntoDL+

With these changes OntoDL+ will reduce the volume of code necessary to specify an ontology, and improve the readability of the specified ontologies.

4.2.2 *Ontology Constraints*

OntoDL+ also extends OntoDL by allowing the creation of constraints. Currently, OntoDL does not support the creation of constraints for the relations. By introducing this feature, OntoDL+ will allow the creation of better structured ontologies.

These restrictions can be described in a new optional section of an OntoDL+ specification file, and they are converted to Prolog only. The constraint specification syntax follows a syntax similar to Prolog although not entirely equal. Through these constraints, it will be possible to use Prolog to verify if the ontology satisfies the described constraints, that way, it will be possible to impose more rigor into OntoDL+ specifications.

```

restrictions {
  G =belongs_to_platform=> P :-
    G =is_available=> A, (A==Digital_Offline; A==Digital_Online) .
}

```

Listing 4.4: Restriction example

As can be seen in listing 4.4, within the restrictions section it is possible to select a relationship to be restricted. The restriction rules follow the syntax from Prolog, being represented by in the format *Head* : – *Body*. The triples will be represented using OntoDL+ syntax, and the structure of the head and body is similar to Prolog.

4.2.3 Idiomatic and syntactic normalization

OntoDL was designed mainly to be used by Portuguese language speaking users. Although some key relations, such as instance of, are supported in both Portuguese as *instancia* and in English as *iof*, some syntactic elements such as *triplos*, and *relacoes* are only supported in Portuguese, not having the English equivalent *triples*, and *relations*. OntoDL+ will provide an English equivalent for such syntactic elements, enabling non-Portuguese speaking users to better use the DSL.

Furthermore, the terminology used by OntoDL is not properly aligned with the commonly used ontology terminology, neither with the OWL terminology. So, while still supporting the current OntoDL terminology, alternative possibilities for terminology will be introduced. These try to normalize terms such as *concepts* and *relations* currently in use by OntoDL, and provide the alternative (and more standard) terminology *classes* and *properties* respectively.

4.2.4 Normalization of data properties

OntoDL+ provides an alternative to the current definition of data properties, normalizing its definition and preventing redundant syntax such as shown in listing 4.5

```

conceitos { Pessoa[nome:string, CC: string, contacto: string],
  Comarca[nome: string],
  Objecto [desc : string],
  Sentenca [desc : string]
}

```

Listing 4.5: Concepts Declaration

In which the type of the data properties *nome* and *desc* have to be specified more than one time.

Therefore, the syntax of listing 4.6 was proposed as a way to improve readability of the DSL.

```

atributos {
  nome: string ,
  CC: string ,
  contacto: string ,
  desc : string
}

conceitos { Pessoa[nome, CC, contacto],
  Comarca[nome],
  Objecto [desc],
  Sentenca [desc]
}

```

Listing 4.6: Attributes Normalization

Having a section destined to describe the attributes to be used in the ontology, will increase productivity by reducing the amount of code necessary to represent the an ontology. It will also enable easier correction and management of the attributes used. This change also looks to introduce sets of literals as a range of data properties.

4.2.5 Relation specification

In order to enable a more complete specification, OntoDL+ extended OntoDL relations specification. By allowing the description of properties of each relation, it is possible to create ontologies with more rigor in its' structure. It's also possible to specify the relation domain and counter-domain. This is an optional feature that will be translated to the Alloy specification. The example in listing 4.7 shows a possible usage of the feature:

```

relacoes {
  temDestino[domain: Ligacão, codomain: Cidade],
  temOrigem[domain: Ligacão, codomain: Cidade],
  éDestinoDe[props: inverseOf=temDestino],
  éOrigemDe[props: inverseOf=temOrigem]
}

```

Listing 4.7: Relations Example

Within square brackets it's possible to specify domain, codomain, and properties in any intended order. It's also possible to describe multiple properties by enumerating them within curved brackets. However, the properties allowed are restricted, the properties provided by OntoDL+ are:

- simple
- entire
- injective
- surjective
- representation
- function
- abstraction
- surjection
- injection
- bijection
- inverseOf= $\langle R \rangle$

By declaring a relation A as an inverse of B, the declaration of domain and counter domain can be omitted in either relation A or B, since they can be inferred. Triple declaration can also be simplified, since the inverse triples will be automatically generated by OntoDL+ compiler.

By making use of said properties it's possible to create more complete specifications, and by enabling it's translation to Alloy, better models can be created.

4.3 COMPILER EXPANSIONS

Along with the language extensions, OntoDL+ introduces some expansions to the compiler. These look, primarily, to enable the translation of the DSL to a larger variety of languages. Along with OWL and DOT, OntoDL+ will also translate its' ontologies to Alloy and Prolog.

4.3.1 Alloy

With the usage of Alloy, OntoDL+ looks to provide a better way to create correct ontologies, by enabling the static model checking of the ontologies' classes, constraints and relations.

The ontology model checking step is recommended to be followed before specifying triples associated with individuals. The best recommendation would be for the user to specify the ontologies' relations, concepts, and triples which do not require the usage of individuals - such as relationships among concepts.

By generating the Alloy specification, it will be possible to use the Alloy Analyser in order to create models and simulations of ontologies, as well as verify properties and constraints. That way it will be possible to ensure much more correction when creating ontology specifications - given that if the specification is lacking, then either the verifications will fail, or some undesirable ontologies will be created as examples by the Alloy tool. In such fashion, it will be possible to detect flaws and possible errors much quicker without having to commit such mistakes, nor have them propagate through the ontology specification - saving time and effort.

Using the Alloy language, the translation from OntoDL+ can become quite intuitive. Given its appropriate vocabulary it enables a representation of most of the ontology elements present in OntoDL+ descriptions. The translation of said elements can also be a direct translation given that Alloy specification language possesses elements which can be considered equivalent to its' ontology counterpart.

- Individuals

Alloy does not support the static declaration of individuals. Since individuals are generated randomly by the Alloy tool to create various models, individuals will not be translated to Alloy.

- Concepts

Concepts are used to describe sets of instances of the same type. The Alloy equivalent is the sig (signature), each signature represents a set of atoms of the same type.

- Attributes

Attributes represent data atoms which will be related to a given individual in ontology specification. Given that it's impossible in Alloy to specify individuals, and since each atom is considered to be the simplest of entities and indivisible, attributes are also not translated to Alloy. That is because attributes would represent information regarding the instance, and following OntoDL+ specification rules, attributes are always present in a 1:1 relation with an individual. Therefore, attributes would add complexity to the model, without adding any modelling benefit.

- Relations

Given the relation domain, - which can be explicitly defined in OntoDL+ or inferred through triple analysis, - the relation can be added as a field in the corresponding relation in the Alloy specification.

- Relation Properties

The properties provided by OntoDL+ will be translated into Alloy facts to enforce in the model, ensuring that the generated models follow the properties specification.

- Triples

There is only one type of triple which will influence the Alloy model, which is triples describing the *is-a* relation among concepts. Other triples will not have influence in ontology specification.

After the translation to Alloy, the user can generate models as is, however, OntoDL+ specification will only translate the limited set relation properties provided. An OntoDL+ specification does not possess any dedicated section to define system properties nor any invariant which will be translated to Alloy. Therefore, it's recommended that the created model is further refined in order to define and verify any desired, more complex, properties.

4.3.2 Prolog

Prolog is also a target language of conversion for OntoDL+. By translating ontologies from the DSL to Prolog, OntoDL+ looks to exploit some of the advantages previously presented in section 2.2.

The translation to Prolog will enable the usage of a different language to:

- Manipulate ontology knowledge

Although a translation from Prolog to OntoDL+ will not be developed during this Thesis, at any given moment, an OntoDL+ user will be able to convert its ontology to Prolog. This conversion allows further addition of facts to the translated ontology, enabling its' enrichment.

- Enable a querying system for OntoDL+

The primary advantage sought after with this expansion, will be the introduction of a querying system for ontologies.

The usage of Prolog as a querying system, may bring further advantages as a validation tool for ontologies. The automatic generation of queries that verify the constraints of a given ontology can provide a validation system for ontologies created using OntoDL+.

Prolog as a language also enables a simple translation from OntoDL+. Most of the translated knowledge is declared as facts of the ontology, while any Prolog restraints that might have been defined in the OntoDL+ specification, will be translated as clauses with head and body.

During translation, the triples, concepts, individuals, relations and attributes will be represented as facts of the specification. The possible restrictions will be translated generating a clause with both head and body. The properties of the relations however, will not be translated to Prolog.

The created Prolog specification, serves mostly as a factual knowledge base, all the knowledge specified in OntoDL+ will be translated to Prolog. However, in order to use Prolog effectively, the user should further specify new clauses to enable reasoning.

4.4 SUMMARY

Along this chapter the language extensions and compiler expansions were proposed. The language extensions were analysed and justified based on the problems of the OntoDL. In order to increase the range of usages of the DSL, some compiler expansions were also envisioned based its uses and utility. The compiler expansions were also described regarding what components of the ontology would be translated for representation, its importance and meaning. It was also explained why some components of the ontology would not be translated to every target language.

ONTODL+, DEVELOPMENT

In order to enable an incremental continuous development of OntoDL+, the previous instance of OntoDL has suffered some refactoring. The OntoDL development was made in a non scaleable approach without proper division of different components of the semantic actions. Given that the development is being conducted in ANTLR4, it was decided to abstract some of source code needed in the semantic actions and create a java package in order to support the actions. The package created enables the encapsulation of code reducing the difficulty in both debugging and comprehension. Such package will be further analysed in section 5.1. The language extensions where developed after the refactoring of the previous code. Those will be detailed in section 5.2. Finally, we will analyse the compiler expansions developed in section 5.3.

5.1 CODE REFACTORING

As analysed in section 3.2 the semantic actions previously present in OntoDL, suffered from various smells such as Long Method, Spaghetti Code and Long Parameter List. These smells were softened in order to enable an easier analysis and development of the software.

Various refactoring techniques where used to improve the previous iteration of OntoDL. Most of the smells, - namely Spaghetti Code and Long Method smells, - where refactored using method extraction techniques. The cases of long parameter list present in the grammar were addressed with the creation of classes which would represent objects with the adequate information. These resulted in the creation of various Java classes and consequently the creation of a Java package. The combination of these abstraction techniques has allowed a serious reduction in the semantic code present in the grammar file. That resulted in a clearer, easier to read and maintain grammar.

5.1.1 Java Package

The package *ontodlp* was created in order to divide the source code present in OntoDL into reusable functions and classes that enable a better management of the software. To establish the intended features of the language, the classes shown in figure ?? were created.

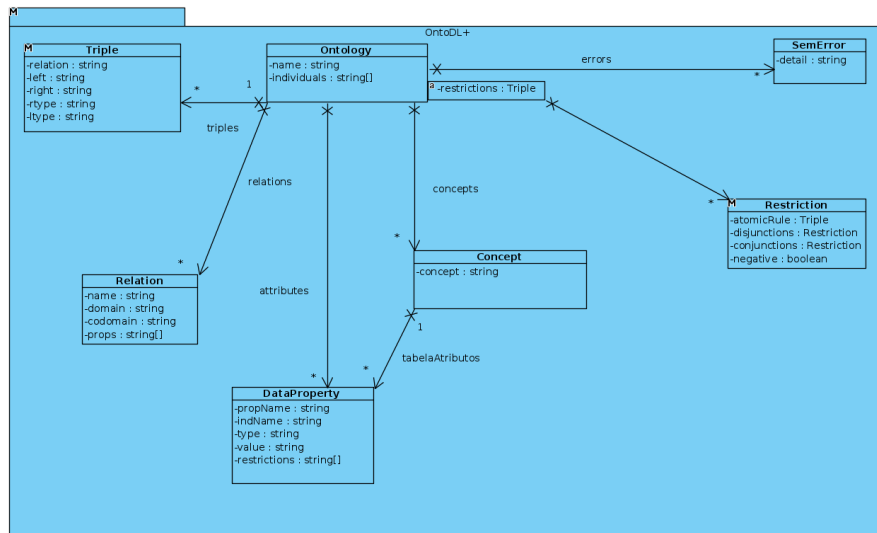


Figure 2: Package Architecture

The classes were created to enable the usage of objects to store and manipulate the information required for the ontology recognition, reducing the problem of Long Parameter Lists in the grammar. Each class has attributes representing the data necessary to describe the component of the ontology. The Ontology class is the main class, aggregating most of the ontology knowledge into one single class. The class also possesses various control methods used to coordinate the translation of the various objects contained within the class, as well as the entirety of the semantic error verification code. The Triple, Concept, Restriction and DataProperty classes encompass the attributes necessary to describe the objects, along with methods to manipulate and translate the information stored into the target languages. The SemError and Relation classes are Data Classes grouping important information to be used (mostly by the Ontology class).

It was considered unnecessary to create a class to just represent the individuals, since during OntoDL+'s individual declaration, the only known information regarding an individual is its' name, therefore, each individual is stored as a string in the Ontology class. Further data involving the individuals can be declared in the triples section, either as data properties describing an individual, - which will be stored as DataProperty objects, - or as relationships that the individual has with other concepts or individuals - which will be stored as a Triple. That is due to the fact that each individual is considered an atomic component described

only by name, which in turn can be required to describe some triples or data properties, however, the individual can stand alone without being used by any triple or property.

The creation of the package has enabled the creation and distribution of methods through the classes, reducing the cases of Long Method and Spaghetti Code smells. However, since most of the ontology semantic verification code requires information regarding various components of the ontology simultaneously, the `Ontology` class takes a role of controller class, centralizing semantic error control methods. Such development has created a slight Blob smell in the `Ontology` class.

5.2 LANGUAGE EXTENSIONS

OntoDL language was improved in order to enable easier to write and maintain OntoDL specifications. However, since this language was already in usage by some people, compatibility with the original OntoDL specification was taken into account. Therefore, OntoDL+ does offer numerous extensions, however, it is entirely compatible with the previous OntoDL language. The extensions were done in an incremental manner without removing or forcing the replacement of previously introduced rules.

OntoDL+ differs from OntoDL mostly by presenting new optional functionalities. Since they allow specification of more optional rules for the ontology, these can be used to specify more complete ontologies. OntoDL+ language also allows the creation of more concise, shorter and easier to read and write specifications when compared to OntoDL language. This is derived from the focus given to the simplification of the language, allowing a user to specify the same information while writing less code.

5.2.1 Triples grouping

OntoDL rules for triple analyses was quite restricted. It required the user to write quite a lot of additional, repeated, information. That was because, by design, the language required the user to write every triple in its entirety, as shown by the productions in listing 5.1.

```

triplo [Ontology onto] :
    '(? c1=TEXT '=' IOF '=>' c2=TEXT attribInsts? )'?
  | '(? c1=TEXT '=' POF '=>' c2=TEXT )'?
  | '(? c1=TEXT '=' ISA '=>' c2=TEXT )'?
  | '(? c1=TEXT '=' SYN '=>' c2=TEXT )'?
  | '(? c1=TEXT '=' rel=TEXT '=>' c2=TEXT )'?
  ;

```

Listing 5.1: OntoDL Triple Specification

Therefore, to provide the proposed alternative syntax, new rules were added to the grammar, shown in listing 5.2. These rules allow the specification of triples which shared equal information on the left-side of the triple. Similarly, triples with matching left-side and relationship can also be grouped due to the new rules added.

```

triplo [Ontology onto] :
    '(? c1=TEXT '=' IOF '=>' c2=TEXT atribInsts? )'?
  | '(? c1=TEXT '=' POF '=>' c2=TEXT )'?
  | '(? c1=TEXT '=' ISA '=>' c2=TEXT )'?
  | '(? c1=TEXT '=' SYN '=>' c2=TEXT )'?
  | '(? c1=TEXT '=' rel=TEXT '=>' c2=TEXT )'?
  | c1=TEXT '=[ ' relacionamentos ( ';' relacionamentos ) * ' ] '
;

relacionamentos [Ontology onto, String leftSide]
    : rel=TEXT '=>' c2=TEXT ( ',' relMultiplo ) *
  | SYN '=>' c2=TEXT ( ',' relMultiplo ) *
  | ISA '=>' c2=TEXT ( ',' relMultiplo ) *
  | POF '=>' c2=TEXT ( ',' relMultiplo ) *
  | IOF '=>' c2=TEXT atribInsts? ( ',' relMultiplo ) *
;

relMultiplo [Ontology onto, String leftSide, String relation]
    : c2=TEXT;

```

Listing 5.2: OntoDL+ Triple Specification

As can be noticed, the already established productions were not altered or removed in order to preserve the previously OntoDL syntax.

5.2.2 Relation properties

Relational properties were introduced to OntoDL+ mainly to improve the accuracy of the alloy models generated. However, these are also compatible with OWL, which also supports the properties introduced. Since conversion to other languages is required, the properties were introduced in a static manner, inhibiting any non-supported properties. At its core, 5 properties were introduced:

1. InverseOf=<Relation>

Determining that a relation is the inverse relation of <Relation>.

2. Simple

Determining that a relation is simple, i. e. every existing value in the domain has at most one corresponding value in the co-domain.

3. Entire

Determining that a relation is entire, i. e. every existing value in the domain has at least one corresponding value in the co-domain.

4. Injective

Determining that a relation is injective, i. e. every existing value in the co-domain has at most one value in the domain corresponding to it.

5. Surjective

Determining that a relation is surjective, i. e. every existing value in the co-domain has at least one value in the domain corresponding to it.

From the introduction of these 5 basic properties, 6 other “free” properties can be derived in order to enable an easier declaration of properties.

1. Representation

A relation that is both entire and injective is called a representation.

2. Function

A relation that is both simple and entire is called a function.

3. Abstraction

A relation that is both simple and surjective is called an abstraction.

4. Injection

A relation that is an injective function, therefore, a relation that is simultaneously injective, simple and entire.

5. Surjection

A relation that is a surjective function, therefore, a relation that is simultaneously surjective, simple and entire.

6. Bijection

A bijection or isomorphism is a relation that is simple, entire, injective and surjective.

The domain and co-domain of a relation can now also be specified in OntoDL+. In order to enable these improvements, new productions were introduced to OntoDL grammar. Namely, `relInf`, `properties` and `property`, are all new relations created for OntoDL+ grammar.

These productions, shown in listing 5.3, ensure a stable version of OntoDL+ capable of handling various different new information regarding the specification of relations. They also enable an easier and more optimal usage of alloy as a model checking tool for ontology specifications.

```

relacoes
    : 'relacoes'
      '- ' r1=TEXT ('[' relInf (',' relInf)* ']' )?
      (',' r2=TEXT ('[' relInf (',' relInf)* ']' )?) *
      ','
    ;

relInf
    : DOMAIN ':' d1=TEXT
      | CODOMAIN ':' c1=TEXT
      | 'props' ':' properties
    ;

properties
    : '(' p1=property (',' p2=property)* ')'
      | p1=property
    ;

property
    : BIJECTION
      | INJECTION
      | SURJECTION
      | ABSTRACTION
      | FUNCTION
      | REPRESENTATION
      | SURJECTIVE
      | SIMPLE
      | ENTIRE
      | INJECTIVE
      | INVERSEOF '=' rel=TEXT
    ;

```

Listing 5.3: OntoDL+ Relation Specification

Although the introduction of static properties through grammar constraints is not the ideal solution in terms of scalability of the language, its' design ensures a more strict and assertive usage. This strictness along with the introduction generation of the alloy specification, is expected make it easier to ensure the structural correction of ontologies.

5.2.3 Constraints

The constraints have been added to OntoDL+ as a new feature of the language. Given the different capabilities and usages of the different targeted languages of conversion, the constraints will not be translated to every language. Being that they were a feature directed to increase Prolog usability in ontology specification, they are only being translated to Prolog as well.

Following a similar logic to Prolog, although some changes are noticeable, just like Prolog rules, the constraints will have a head and a body. The body can be comprised by conjunction or disjunction of predicates, but there can not be any conjunctions or disjunctions in the head. Triples used in the constraints will retain an OntoDL+ syntax, however, some other operators were changed.

Unlike other sections of the specification where `' '` and `' ; '` are used as separators, in the constraints section, `' . '` is actually the separator for different constraints. The `' ; '` and `' ; '` symbols are used instead as symbols meaning 'and' and 'or', respectively. Also following Prologs' grammar, the conjunction (`' & '`) has a higher precedence than the disjunction (`' ∨ '`). Therefore, constraints of the form:

```
Triple :-
  PredicateA , PredicateB ;
  Predicate C, PredicateD .
```

Will be analysed as:

```
Triple :-
  (PredicateA ∧ PredicateB) ∨ (PredicateC ∧ PredicateD) .
```

Instead of:

```
Triple :-
  PredicateA ∧ (PredicateB ∨ PredicateC) ∧ PredicateD .
```

A predicate can also include disjunctions with higher precedence than the conjunctions, but they must be duly signaled by being within brackets.

The translated file also includes the rule `not(X)`, which allows for the negation of predicates with a closed world approach.

```
not(X) :- !, fail.
not(X).
```

Listing 5.4: Implementation of predicate 'not/1' in Prolog

Negations in the constraints section of the OntoDL+ specification, will be translated to Prolog as `not(Constraint)`, as illustrated in listing 5.4.

5.2.4 *Attributes Normalization*

The attribute normalization was implemented and proposed as an alternative to the previous OntoDL attribute specification. It looks to reduce redundancy as well as errors in ontology specification. However, due to the need to support OntoDL syntax fully, the previous declaration format for attributes in placed can be freely used and intertwined with the normalized format with few restrictions. The user is free to choose where it is most convenient to declare the attributes, however it is recommended that said declarations take place in the attributes section to reduce ontology errors and increase productivity.

```
attributes {
    attribA:string,
    attribB:string
}

concepts {
    ConcA[attribA, attribC:string],
    ConcB[attribA, attribB]
}
```

Listing 5.5: Attributes Normalization Example

An attribute declared in the attributes section needs to have its' type declared and its' type can't be re-declared in the concepts section. That is, there isn't only one place capable of accepting attribute declaration, but there can not be multiple declarations of the same attribute. Therefore, the attributes section is entirely optional, it is not required to exist in an OntoDL+ specification as long as all the attributes used are properly declared following OntoDL syntax, like `attribC` of the example listing 5.5.

5.2.5 Expands Feature

The expands feature was also added in order to improve organization of extensive ontologies. It's a command that allows the expansion of source code from a different file in the ontology file following the syntax of listing 5.6.

```
expands <file`name>;
```

Listing 5.6: Expands Example

During a preprocessing phase which was added to OntoDL+ processor, the occurrences of a file expansion will be replaced by the target file content. The preprocessing will create a temporary file containing all the ontology knowledge within it. Afterwards, OntoDL+'s compiler will process the file and delete it upon completion. This allows for the introduction of modularity to OntoDL+ specifications. This extension can be especially useful when dealing with large ontologies. By allowing the import of knowledge from different files, it is possible to create a knowledge structure or organization, enabling easier maintenance and readability.

5.3 COMPILER EXPANSIONS

The OntoDL+ compiler is an expansion of the OntoDL compiler. It supports the features from its' predecessor while implementing and translating the new features added to the language. Furthermore the compiler was also expanded to include two target languages for translation. With the expansion, OntoDL+'s compiler can not only translate the ontology specification to OWL (RDF/XML syntax) and DOT, but also to Alloy and Prolog.

5.3.1 Alloy

OntoDL+'s compiler as introduced the translation of ontology specification to an Alloy static model specification. As analysed in section 4.3.1 the Alloy specification will depend mostly on the specification of concepts, relations and relation properties, although relations should have specified its' domain and counter-domain to ensure better models.

```
concepts {
  Available ,
  Ending ,
  Game[name:string , description:string , age`rating:enum(3, 7, 12, 16, 18) ,
    storyline:enum('real world', 'fiction', 'null')],
```

```

    Game_Mode,
    Genre,
    Player_Number
}

relations {
    belongs_to_genre [domain:Game, codomain:Genre],
    has_mode [domain:Game, codomain:Game_Mode],
    has_player_number [domain:Game, codomain:Player_Number],
    is_available [domain:Game, codomain:Available ],
    with_ending [domain:Game, codomain:Ending]
}

```

Listing 5.7: OntoDL+ Example

Given the provided (partial) specification of listing 5.7, the Alloy equivalent model, shown in listing 5.8 will have 6 signatures, - representing the concepts, - with a total of five fields among them - the ontology relations.

```

sig Game{with_ending : Ending,
    belongs_to_genre : Genre,
    has_player_number : Player_Number,
    is_available : Available,
    has_mode : Game_Mode}

sig Available{}

sig Player_Number

sig Ending

sig Genre

sig Game_Mode

```

Listing 5.8: Alloy Specification

Since there are no properties restricting the cardinality of relations, it has not been represented in the model. However, it is possible to note that not only was a signature created for each concept, the relations were also created as fields of its domain signature, while also specifying its' counter-domain - hence showing the importance of domain and counter-domain. If the domain of a relation is not specified, it will not be represented as the field of any signature. If the counter-domain is not specified, it will be specified univ, allowing the field to relate to every declared signature.

The properties are mostly represented as facts or combinations of facts, with the exceptions being the simple, entire and function properties. These three properties do not generate additional facts, instead they restrict the field cardinality:

- simple

The simple property refers to a field which: for each instance of signature A generated, there will be at most one signature B connected to it by the field.

```
sig A {field: lone B}
```

- entire

The entire property refers to a field which: for each instance of signature A generated, there will be at least one signature B connected to it by the field.

```
sig A {field: some B}
```

- function

The function (simple and entire) property refers to a field which: for each instance of signature A generated, there will be exactly one signature B connected to it by the field.

```
sig A {field: one B}
```

The properties `inverseOf`, `surjective` and `injective` will be translated as Alloy facts:

- `inverseOf`

Since Alloy allows the representation of the inverse of a relation, the inverse property is declared by equivalence. So if a relation A is the inverse of a relation B, then the inverse relation of A is equal to B. Therefore, those facts are stated as:

```
fact {
    ~A = B
}
```

- `surjective`

The surjective property is ensured by imposing that *the image of the relation is reflexive* (definition 5.36 from Oliveira).

```
fact {id [<Codomain>] in img [<relation>]}
```

- injective

The injective property is ensured by imposing that *the kernel of the relation is coreflexive* (definition 5.36 from Oliveira).

```
fact {ker [<relation>] in id [<Domain>]}
```

The properties: representation, abstraction, injection, surjection and bijection, are defined as a combination of facts of the previous simpler properties 5.2.2 (surjective, injective, function, entire and simple).

5.3.2 Prolog

The OntoDL+ compiler also supports translation to Prolog. As referenced in 4.3.2 the translation will target most of OntoDL+ specification, the exception being the relation properties. During translation, the names of individuals, concepts and relations used in facts might be slightly changed from its' OntoDL+ original representation. Since in Prolog words starting with capital letter are considered as variables, any term starting with capital letter will be changed to lowercase. The restrictions support the use of variables which will remain unchanged in Prolog, as long as they do not match the name of any concept, individual or relation.

- Individuals, concepts and relations

Each individual, concept, or relation is declared separately. For each existing entity of those types, one fact describing its existence is generated. The syntax used for fact declaration is of the form $\langle \text{Type} \rangle (\langle \text{name} \rangle)$. - i.e. an individual named John will be declared as `individuo(john)`.

- Triples

Triples are comprised of the relation, left side and right side. Therefore, the translation to Prolog goes as follows $\langle R \rangle (\langle LS \rangle, \langle RS \rangle)$. - i.e. factually stating that there is a relation R among the left side atom LS, and the right side atom RS.

- Attributes

Since the same individual can have multiple literals described as data properties/attributes, the translation makes use of Prolog lists to reduce the number of facts. Therefore, the attributes of a same individual are grouped as following the structure: `properties(<Individual>, [<list of literals>])`.

- Restrictions

The restrictions were added in OntoDL+, and just like in Prolog, they follow the form `Head :- Body`. However, the syntax for triple, individual, concept and relation declaration is the OntoDL+ syntax. Therefore, the translation follows by translating the expressions in both head and body from OntoDL+ syntax, to Prolog syntax.

After translation the user will be able to create enhance the specification by adding more predicates, or simply use the specification as is to achieve a querying system.

5.4 SUMMARY

Along this chapter, the implementation decisions taken were presented. The refactored solution of the previous implementation was presented, identifying some most commonly used techniques during the process. The ensuing package was also explained describing its structure and organization. The language extensions were described, highlighting some of the key changes that had to be done to language grammar. Finally, the compiler expansions were described, expressing the information to be translated and the choices made during the translation process.

ONTODL+, WEBPAGE AND EXPLORATION

In order to publicize the OntoDL+ language and its compiler, a website was created using HTML, and is currently being hosted at https://epl.di.uminho.pt/~gepl/GEPL_DS/OntoDL/. It was designed to present the features provided by the language, as well as offer a download platform for users of the language. The website includes four web pages in total:

1. **Home** serves to present the language, its advantages and a small introduction
2. **Examples** provides some code snippets explaining the language extensions in more detail
3. **Download** includes an executable version of the compiler application, a template and examples
4. **Contact** featuring the contact information of the development team

6.1 HOME

The home page of the website shown on Figure 3 is a very simple presentation page. It contains a small introduction to the DSL, as well as the expansions and extensions that OntoDL suffered. It also includes a short video which was created to give a general overview of the DSL and its capabilities. The video was published on YouTube, and is currently being displayed on the website as well.

Some information is contained within collapsible cards to reduce the text shown to the users of the website, ease their reading, comprehension and navigation on the site.

OntoDL+

Home Examples Download Contact

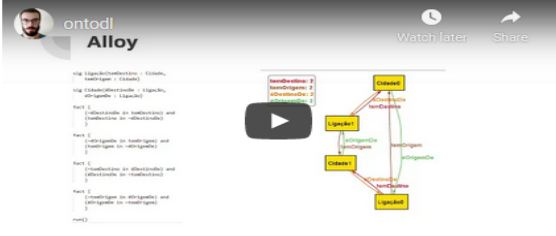
OntoDL+

OntoDL+ is a Domain Specific Language created as an improved version of OntoDL. The language allows ontology representation and, through its compiler, conversion to other ontology representation languages. The OntoDL language has been extended to OntoDL+ in order to improve usability. And OntoDL+ compiler has also been expanded translating ontologies to more languages than the previous OntoDL compiler.

OntoDL+ Was created by Alexandre Dias from OntoDL. It looks to increase the capabilities of OntoDL by improving both the language, and its' compiler. This Master Thesis project was done under the guidance of Pedro Rangel Henriques and Cristiana Araújo.

[Compiler Expansions](#) [Language Extensions](#)

OntoDL+ on a glance:



The screenshot shows a video player interface with a play button and a video thumbnail. The thumbnail displays an OntoDL+ ontology specification on the left and a corresponding graph representation on the right. The graph shows nodes for 'OntoDL+', 'Ligando1', 'Ligando2', and 'Ligando3' connected by edges. The video player also includes a 'Watch later' button and a 'Share' button.

Figure 3: Home page

6.2 EXAMPLES

The examples web page contains a few code snippets to better highlight some of the language extensions and its advantages. It includes an OntoDL ontology specification as base example along with equivalent OntoDL+ alternatives. Each alternative features small incremental changes that can be done to the specification using OntoDL+, improving its readability and reducing the code required.

OntoDL+ Home Examples Download Contact

Examples

The following examples describe an ontology expressing road connections provided by an imaginary company. All of them describe the same ontology utilizing different functionalities that OntoDL+ has incorporated.

The full version of the ontologies used in the examples, can be downloaded in the download section.

Road connections

Road Map Simple

Road Map With Triple Grouping.

The following ontology has incorporated the possibility to group triples starting with the same individual/concept.

```
triplos {
  brg = [ iof => Cidade [descrição="Cidade dos Arcebispos", distrito="Braga", nome="Braga", população="140000"];
  destinoDe => lig-ptl-brg;
  origemDe => lig-brg-bcl,
           => lig-brg-gmr,
           => lig-brg-fml];
  .....
}
```

One of the capabilities introduced to the OntoDL language, was the Grouping of triples. This improvement tries to adress the following issues:

- Reduce code repetition and possibility of errors.
- Boost productivity of the users.
- Increase readability.

Road Map With Relation properties

Figure 4: Examples

To improve comprehension of the highlighted features, only the snippets of code being changed are presented, as shown in Figure 4. In such fashion, it becomes simpler to focus on the improvements provided by each extension.

6.3 DOWNLOAD

The download page provides various resources that the users can download and use to their discretion, as highlighted in Figure 5. It includes the source code of the application, enabling users to further extend the language or compiler if proven necessary. It also provides an executable version of the OntoDL+ compiler, describing the system requirements and short instructions on how to use the application.

OntoDL+ can be used in any computer possessing a version of java superior to version 16.0.1 of openjdk installed. In order to use it, you will simply have to download de OntoDL.jar file, and run:

```
java -jar <path_to_ontoDL.jar> <targetfile>
```

OntoDL

Jar package file	OntoDL.jar
Source code	src.zip

Templates

OntoDL+ Template File	template.odlp
-----------------------	-------------------------------

Examples

OntoDL example	mapa.txt
OntoDL+ with triples Grouping.	mapaWithGroup.txt
OntoDL+ with concept properties.	mapaWithProps.txt
OntoDL+ with attributes.	mapaWithAtribs.txt
OntoDL+ with expands.	mapaWithExpands.zip

Figure 5: Download

It further contains a template file, which can be downloaded and immediately compiled to explore and test the language. The full specification of the ontologies shown in the examples page is also available at the downloads page.

6.4 CONTACT

The contacts web page is a simple page containing information regarding how a user can contact the developers of the application. As shown in Figure 6, it includes the emails and personal web pages of the team responsible for OntoDL+'s maintenance.

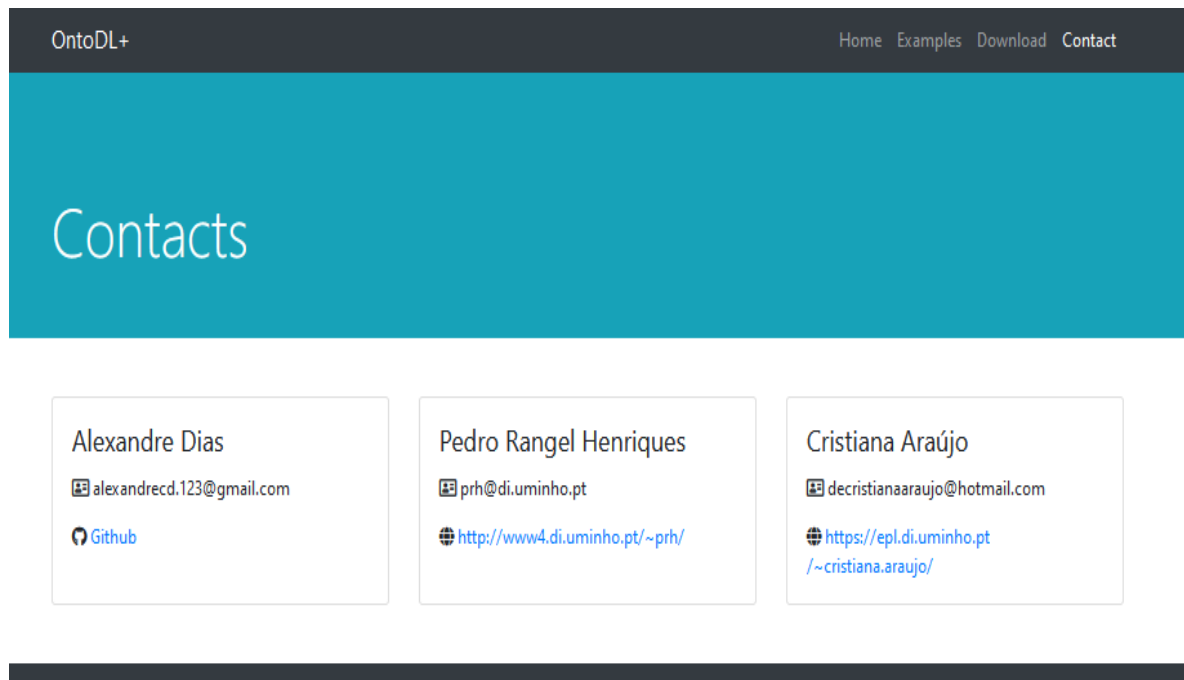


Figure 6: Contact

6.5 SUMMARY

Along this chapter, the WebPage for OntoDL was created and explained. The different pages were explained given the purpose of its use, and some images of the website were shown.

ONTODL+, RUNNING EXAMPLE

The language being used and tested in parallel with its development. Saete Teixeira has used OntoDL+ in a master thesis to formally describe the ontology OntoJogo (Teixeira, 2021). An excerpt of OntoJogo will be used in this chapter to illustrate some of the language extensions. José Carlos Ramalho has also provided an example of an ontology called OntoMapa written in turtle syntax (Ramalho, 2020). It was used to create a smaller version of the ontology written in OntoDL+ language. The OntoDL+ excerpts that will be presented along this chapter are publicly available on the website described in Chapter 6, in the download page ¹.

7.1 ONTOMAPA

OntoMapa is the ontology provided by José Carlos Ramalho, it was created to describe the road connections between cities. It was shortened and translated to OntoDL, having been incrementally changed to incorporate the new language features. The various iterations followed the sequence of implementing the triple grouping feature, followed by the introduction of relation properties and use of the attribute section. In a final iteration the ontology was adjusted making use of the all the new features.

```
conceitos {  
  Ligação[ distância:string ],  
  Cidade[ descrição:string , distrito:string , nome:string , população:string ]  
}
```

Listing 7.1: OntoMapa Concepts

As can be observed in listing 7.1, the ontology possesses only 2 concepts: City (Cidade), and Connection (Ligação). Each instance of city (cidade) will have its own description (descrição), district (distrito), name (nome) and population (população). Meanwhile the instances of connection (Ligação) will specify its length (distância).

¹ <https://epl.di.uminho.pt/gepl/GEPLDS/OntoDL/download.html>

```

individuos {
    brg, bcl, ptl, gmr, fml, prt, vct, %Cities

    lig-ptl-bcl, lig-ptl-brg, lig-brg-bcl, lig-brg-gmr, lig-brg-fml,
    lig-gmr-fml, lig-fml-bcl, lig-fml-prt, lig-bcl-vct, lig-vct-prt,
    lig-gmr-prt %Connections
}

relacoes {
    temDestino,
    temOrigem,
    éDestinoDe,
    éOrigemDe
}

```

Listing 7.2: OntoMapa Individuals and Relationships

The ontology will instantiate 7 cities and 11 connections, shown in listing 7.2. It will further specify 4 relationships, which, during the first iterations, did not have any properties specified. Finally the ontology possesses the triples specification as exemplified in listing 7.3. Since this is only an OntoDL translation, the triple specification does not have any triple grouping yet.

```

triplos {
    brg = iof => Cidade [descricao="Cidade dos Arcebispos", distrito="Braga",
        nome="Braga", populacão="14000"];
    bcl = iof => Cidade [descricao="Cidade do Galo", distrito="Braga", nome="
        Barcelos", populacão="60000"];
    ptl = iof => Cidade [descricao="Vila mais antiga de Portugal", distrito="
        Viana do Castelo", nome="Ponte de Lima", populacão="70000"];
    ....

    lig-ptl-bcl = iof => Ligação [distância= "19"];
    lig-ptl-bcl = temDestino => bcl;
    lig-ptl-bcl = temOrigem => ptl;
    bcl = éDestinoDe => lig-ptl-bcl;
    ptl = éOrigemDe => lig-ptl-bcl;
    ...
}

```

Listing 7.3: OntoMapa Triples

7.1.1 Triple Grouping

During the first iteration of the ontology modification, it was changed in order to use the triple grouping feature, as exemplified in listing 7.4. This caused no change in the concepts, individuals or relationships sections, evolving only the triples section.

```

triplos {
  brg =[ iof => Cidade [ descrição="Cidade dos Arcebispos", distrito="Braga",
    nome="Braga", população="140000" ],
    éDestinoDe => lig - ptl - brg ,
    éOrigemDe => lig - brg - bcl ;
    => lig - brg - gmr ;
    => lig - brg - fml ] ;

  bcl =[ iof => Cidade [ descrição="Cidade do Galo", distrito="Braga", nome="
    Barcelos", população="60000" ],
    éOrigemDe => lig - bcl - vct ,
    éDestinoDe => lig - ptl - bcl ;
    => lig - fml - bcl ;
    => lig - brg - bcl ;
    => lig - brg - gmr ] ;

  ptl =[ iof => Cidade [ descrição="Vila mais antiga de Portugal", distrito="
    Viana do Castelo", nome="Ponte de Lima", população="70000" ],
    éOrigemDe => lig - ptl - bcl ;
    => lig - ptl - brg ] ;

  ...
}.

```

Listing 7.4: OntoMapa Triple Grouping

The improvement has created an ontology file with a bigger number of lines due to different indentation decisions. However, the major benefit comes from the amount of non-spacing characters required to specify the ontology. While the OntoDL version of the ontology needed a total of 2714 non-spacing characters for a complete specification, the introduction of the triple grouping feature reduced said number to 2293, a decrease of more than 15%. Therefore, it was possible to increase the productivity while using the language by allowing to specify the same amount of knowledge, while reducing the work necessary.

7.1.2 Relation properties

The introduction of relation properties has allowed further improvement of the ontology and, full representation of the original ontology provided by José Carlos Ramalho. That's because

the initial specification written in OntoDL did not allow for the declaration of relational properties. However, the ontology provided did possess some relational properties specified, which were translated during this iteration and are presented in listing 7.5.

```
relacoes {
  temDestino[domain: Ligacão, codomain: Cidade],
  temOrigem[domain: Ligacão, codomain: Cidade],
  éDestinoDe[domain: Cidade, codomain: Ligacão, props: inverseOf=temDestino],
  éOrigemDe[domain: Cidade, codomain: Ligacão, props: inverseOf=temOrigem]
}
```

Listing 7.5: OntoMapa Relation Properties

The introduction of properties allowed more usability for the Alloy translations, however, one of the key improvements becomes clearer with the introduction of the `inverseOf` property. Since the property provides automatic triple inference, the OntoDL+ compiler ceases to require triples with the relations `éOrigem` and `éDestino`. As a result, it was possible to further reduce the ontology, in both number of lines and characters. Length wise, the ontology has suffered a decrease of slightly over 10% of the lines of the original file. Character wise, the ontology possess only 2048 non spacing characters, corresponding to a reduction of almost 25% of the OntoDL specification and a reduction of roughly 10% of the triple grouping iteration.

7.1.3 Attributes Normalization

The following iteration, did not have any major impact in terms of characters written. However, the feature does improve readability of the specification by reducing line length in the concepts section, as well as by grouping common information enabling easier access.

```
atributos {
  distância: string,
  descrição: string,
  distrito: string,
  nome: string,
  população: string
}

conceitos {
  Ligacão[distância],
  Cidade[descrição, distrito, nome, população]
}
```

Listing 7.6: OntoMapa Attributes Normalization

As shown in listing 7.6, it was possible to reduce the volume of information concentrated around the concepts section.

7.1.4 *Expand*

The expands feature does not have direct implications in the length of the ontology specification. It was designed mainly to be used with large ontologies which require introduction of modularity and division of ontology sections throughout different files. However, an example using the expands was created and is available for the ontology OntoMapa.

```
Ontologia mapa

atributos {
    expand atributos.odlp;
}

conceitos {
    expand conceitos.odlp;
}

individuos {
    expand individuos.odlp;
}

relacoes {
    expand relacoes.odlp;
}

triplos {
    expand triplos.odlp;
} .
```

Listing 7.7: OntoMapa Expands Feature

Since the example size was small, the expand usage was shown by importing each section's text from an external file, as illustrated in listing 7.7.

7.1.5 *Processing*

The compilation of the ontology generates the 4 outputs required. It is possible to test the examples by following the instructions on the download page of the website. It is merely required to have Java 16.0.1 or a higher version installed, then by running the command `java -jar <path_to_OntoDL.jar> <targetfile>`, the output files are created.

The OWL translated ontology can be viewed, used and updated afterwards using environments such as Protégé. In Figure 7 it is possible to see the description of `temDestino` provided by Protégé.

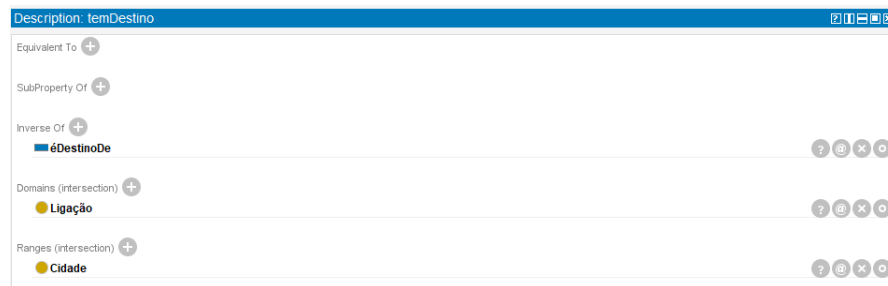


Figure 7: Protege `temDestino` description

The file created has over 7 times more lines of code than the OntoDL+ specification. Furthermore, having 28707 non-spacing characters, it requires over 10 times more code than any of the OntoDL+ iterations of the specification.

The generated DOT file stands at 7845 characters used, representing a reduction to nearly one third of the code used in the OntoDL+. The created DOT file is able to create the image representation of the specified graph. However, it proves to be a better use to filter the graph to include only parts of the ontology, creating graphical projections of the ontology. For illustration purposes, the ontology was commented to include only triples with connections which include `brg` (Braga) as either an origin or a destination, creating a filtered view of Braga's connections. The graph generated from said filter can be viewed in Figure 8.

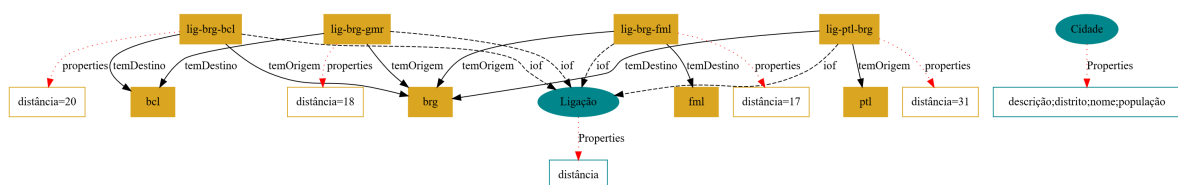


Figure 8: DOT Graph of Connections passing through `brg`

The Alloy file is considerably smaller than the OntoDL+ specification. A big reason for that has to do with the use of the language. Alloy is not a specification language, it is a modelling language. Therefore, it will be used to represent only the ontology structure and properties.

Using the Alloy tool as shown in Figure 9, it is possible to execute the generated file to create models. Such models can then be used to try to identify possible flaws in the ontology structure. For instance, Figure 10 shows a possible instance of OntoMapa which does not have any visible problem, however, Figure 11 shows a connection in which both

the beginning and the end of the connection belong to the same city, which might be an undesirable behavior.

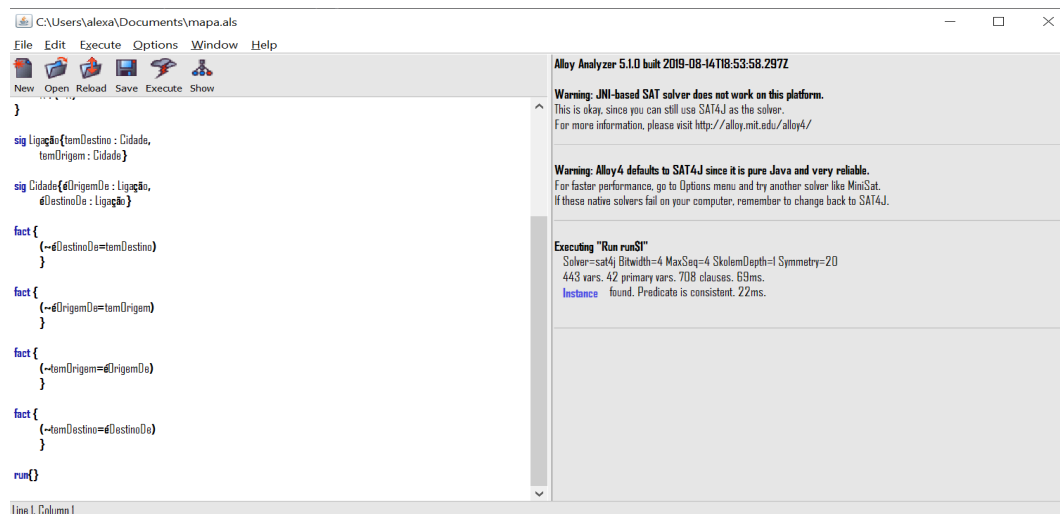


Figure 9: AlloyTool execution

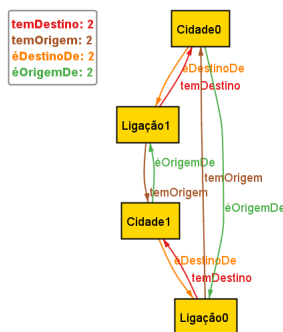


Figure 10: AlloyTool Instance 1

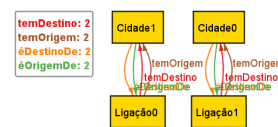


Figure 11: AlloyTool Instance 2

Finally there's the Prolog file, with only 3685 non spacing characters, the difference in efficiency when producing code is not as prevalent as in OWL and DOT. However, there is nonetheless reduction in the required code for the specification.

```

🐼 SWI-Prolog (AMD64, Multi-threaded, version 8.2.1)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
Warning: c:/users/alexa/documents/mapa.pl:2:
Warning: Singleton variables: [X]
% c:/Users/alexa/Documents/mapa.pl compiled 0.00 sec, 112 clauses
?- individuo(brg).
true.

?- not(individuo(brg)).
false.

?- █

```

Figure 12: Prolog consult

The Prolog file can be consulted using tools like SWI-Prolog. As shown in Figure 12, it can be used to query the ontology. Furthermore, the not function was introduced to allowed forced negation.

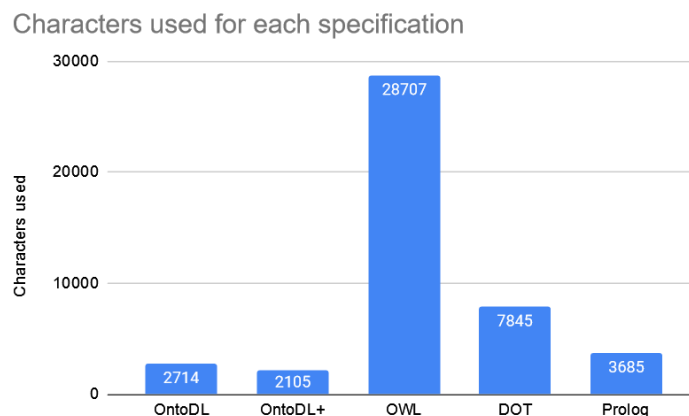


Figure 13: OntoMapa Characters used in each language specification

The graph from Figure 13 shows the comparison of characters used per specification using the languages OntoDL, OntoDL+, Prolog, DOT and OWL. Through it, it's possible to understand that OntoDL+ has fixed the redundant syntax present in OntoDL, proving to be the language which allows for the smallest file while containing the same information.

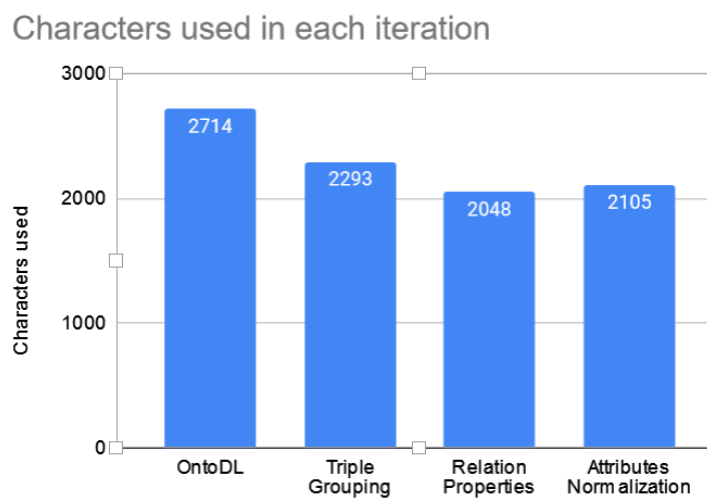


Figure 14: OntoMapa Characters used in each iteration of the specification

Furthermore, the graph from Figure 14 highlights the size differences for each iteration of the ontology created, enabling a better understanding of the effects of each iteration.

7.2 ONTOJOGO

OntoJogo is centered on the subject of game representation. The ontology was used to model the domain knowledge of an application. It allows the description of various information creating a classification methodology for games, which will be used for game suggestion in the application.

The ontology was mostly created during the research phase of OntoDL+, therefore, most of it was written using OntoDL syntax. There is not much use of the language extensions provided, however, the ontology does use some new features. For this example, most of the ontology was kept as it was originally. The concepts, and individuals were kept unchanged, and all the relationships were included as well. The triples however were reduced to ease comprehension, including only two of the specified games and its triples, as shown in listing 7.8. For this purpose two of the instantiated games were used: Chess and LightBot.

```
triplos {
  ...

  Chess = iof => Game [ name= 'Chess', description= 'Chess is a two-player
    board game
      played on a checkered board, where the goal is to kill the king of
        the opponent.', age_rating=3, storyline='null'];
  Chess = is_available => Non-Digital;
  Chess =with_ending=> Singular;
  Chess =has_mode=> Competitive;
  Chess =belongs_to_genre=> Board ;
  Chess =belongs_to_genre=> Strategy ;
  Chess =has_information=> Perfect;
  Chess =has_player_number=> Multiplayer;

  LightBot = iof => Game [ name= ' LightBot: Code Hour' , description= '
    Lightbot is a
mobile game for learning software programming concepts.', age_rating=3 ,
storyline= 'null' ] ;
  LightBot =is_available => Digital-Offline ;
  LightBot =involves_character_choice=> Default ;
  LightBot =with_ending=> Singular ;
  LightBot =has_mode=> Casual ;
  LightBot =belongs_to_platform=> Mobile ;
  LightBot =belongs_to_genre=> Educational ;
  LightBot =belongs_to_genre=> Puzzle ;
  LightBot =with_input=> Touch ;
  LightBot =has_player_number=> Single_Player ;
  LightBot =has_perspective=> Third_Person ;
```

```

LightBot =has_progression=> Level_Based ;
LightBot =has_progression=> Linear
}

```

Listing 7.8: OntoJogo Triples Example

Furthermore, during development, there have been some slight changes regarding the proposed design of the restrictions, these can be seen in listing 7.9. Therefore to make full use of them, they were updated to fix some use issues. Namely, the previously defined restrictions used variables with equal names to already defined concepts or individuals, creating errors during translation.

```

restrictions {
  G =belongs_to_platform=> P :-
    G =is_available=> A, (A==Digital_Offline; A==Digital_Online) .

  G =with_input=> I :-
    G = is_available => A , (A == Digital_Offline; A == Digital_Online) .

  G = is_available=> A :-
    ( A == Non_Digital , A \= Digital_Offline , A\=Digital_Online);
    ((A==Digital_Offline; A== Digital_Online) , A\=Non_Digital) .
} .

```

Listing 7.9: OntoJogo Restrictions

The new restrictions keep the intended meaning of the original, however, through the use of variables, they gain usage in a querying system.

7.2.1 Processing

As happened with the ontology OntoMapa, compiling OntoJogo will generate 4 different outputs. The OWL generated can be used and checked with Protégé as can be seen in Figure 15 and Figure 16.

Furthermore, as can be seen in Figure 17, while the OntoDL+ specification requires 4069 non-spacing characters, the OWL specification uses 39320 characters in its description. This secures yet again much more productivity when using OntoDL+, given that it is required to write nearly 10 times less code. Similarly, a full DOT specification stands at 9341 characters used, more than double the code required in the OntoDL+ specification.

However, the DOT graph is much more useful with the creation of filtered views. By commenting every individual and triple except the ones used in the specification of one

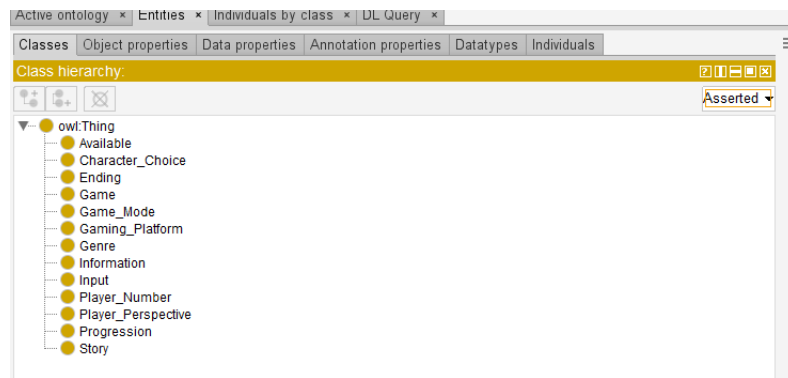


Figure 15: Protégé Classes

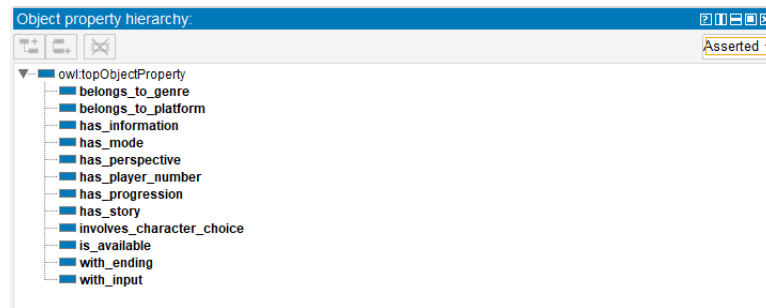


Figure 16: Protégé Object Properties

Characters used for each specification

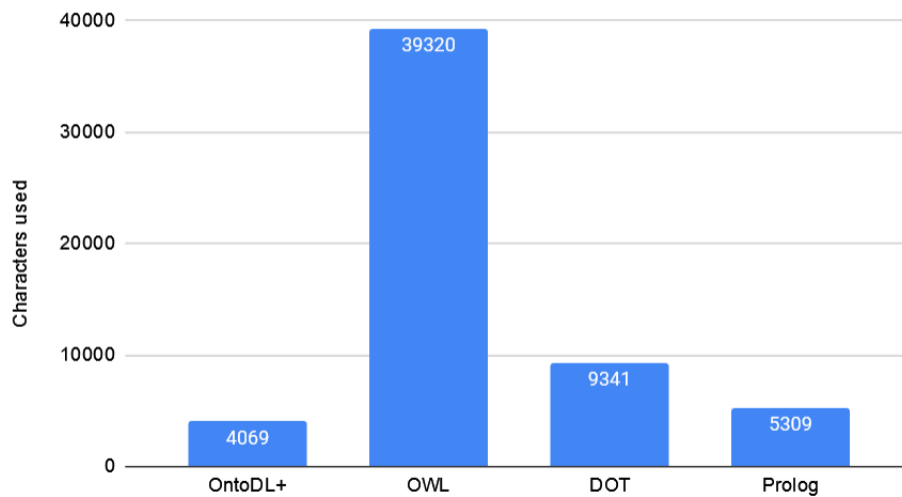


Figure 17: Characters Used in Each Language Specification

game (for instance, Chess), it's possible to create smaller game oriented graphs, as shown in Figure 18.

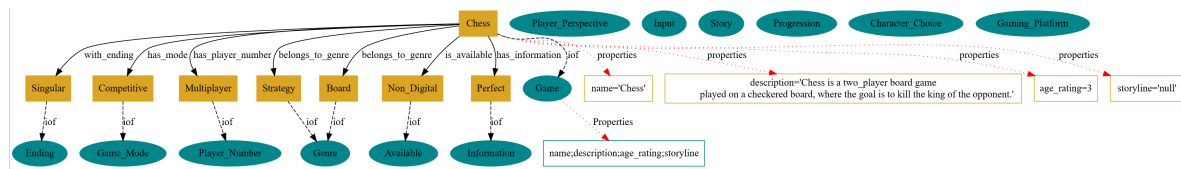


Figure 18: Chess DOT Graph

Given the higher complexity of the ontology structure, Alloy will generate more complex models, harder to analyse, as highlighted in Figure 19. Given that there are more concepts and relationships, with next to no rules defined, the models created are considerably larger than previously. For this reason, Alloy also provides tools to enable easier understanding of them. Using projections or creating themes as shown in Figure 20, it is possible to make a more in-depth analysis of the model.

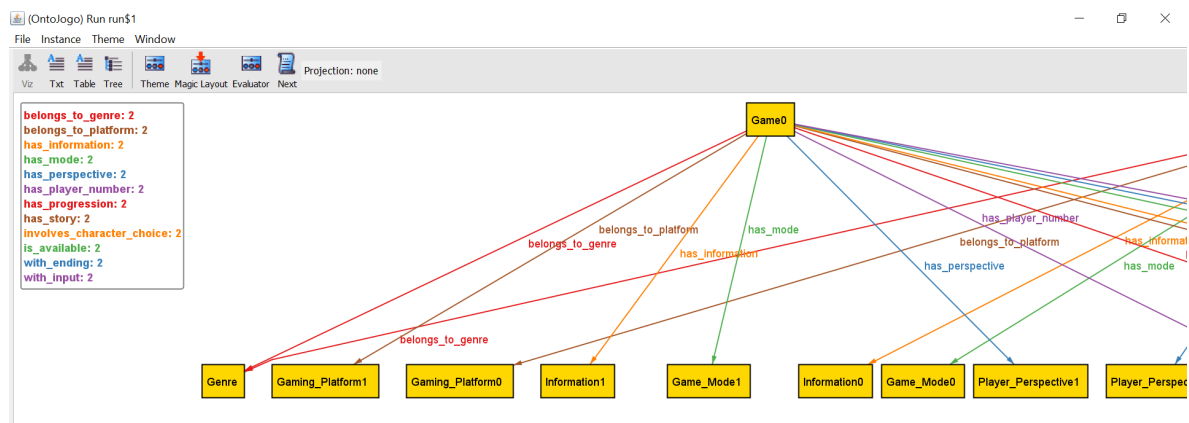


Figure 19: Alloy model

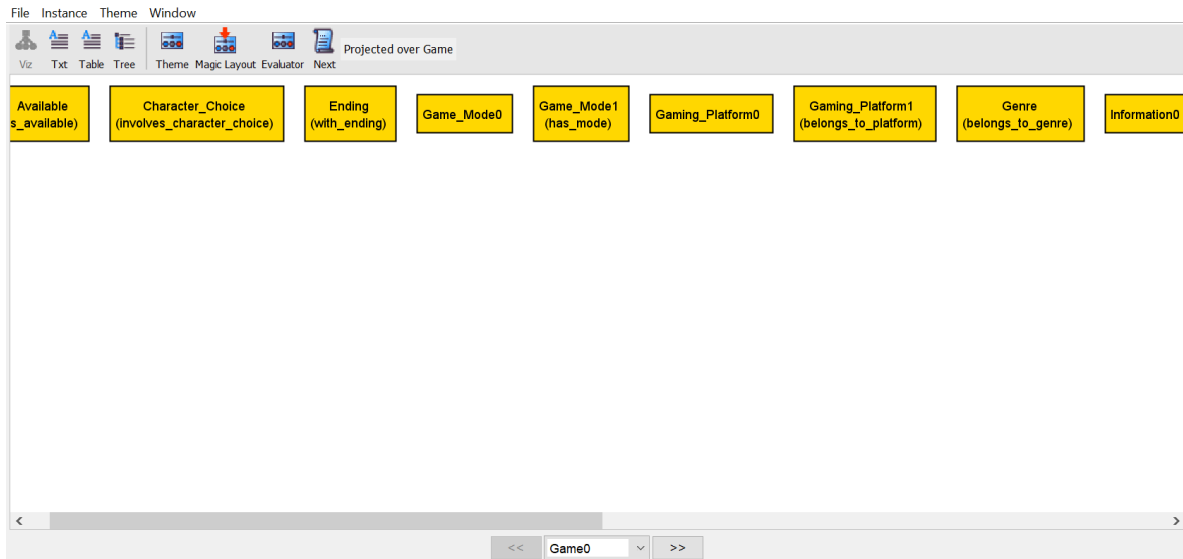


Figure 20: Alloy Projection

Finally the generated Prolog file can be consulted using SWI-Prolog. Using the corrected restrictions specification, it is possible to see in Figure 21 that the restrictions can be queried and verified using Prolog.

```

SWI-Prolog (AMD64, Multi-threaded, version 8.2.1)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (threaded, 64 bits, version 8.2.1)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/alexa/Documents/OntoJogo.pl compiled 0.00 sec, 174 clauses
?- restricao(is_available(chess,digital_offline)).
false.

?- restricao(is_available(chess,non_digital)).
true .

?- restricao(belongs_to_platform(lightbot,mobile)).
true .

?- restricao(belongs_to_platform(lightbot,pc)).
false.

```

Figure 21: Prolog Restrictions

7.3 SUMMARY

Along this chapter OntoDL+ compiler was explored to explain its usage. The improvements in productivity provided by OntoDL+ were highlighted, showing a decrease of nearly 10 times in the code required for specification. The advantages in using Alloy as a modelling language were also shown highlighting how some mistakes can be easier to detect, and how larger ontologies can be analysed and modelled. Finally the purpose of using Prolog as an inference system was also underlined by expressing some of the queries which can be used.

CONCLUSION AND FUTURE WORK

In this final chapter of the Master Thesis, the conclusions taken from the results of the previous chapters are presented. The final section of this work presents future work that can be followed to take this project further on.

8.1 CONCLUSIONS

In this Master Thesis it was proven that it was possible to improve the usability of OntoDL through language extensions and the implementation of new features.

Extending the language for attribute normalization has improved the readability of the language. The extension is also useful in avoiding errors during attribute declaration and has improved the ease of access to the attributes declared in an ontology.

The extensions in the declaration of triples have removed redundancies in the specification, resulting in a reduction of over 15% of the code required in OntoDL. The language was further extended to incorporate relation properties and inference, increasing the code reduction from 15% to 25% comparatively to OntoDL.

The adaptation of the context validator was also executed ensuring that all the desired semantic verifications from OntoDL were implemented in OntoDL+. To ensure that, some errors previously present in the context validator were also fixed during the code refactoring stage.

The compiler expansions also ensure that the generated ontology outputs can be used by outside applications. For instance, the OWL files created by the compiler can be opened, analysed and changed using applications such as Protégé, which also allows the automatic conversion to other ontology syntax formats, such as Turtle. Such conversion further allows users to benefit from other tools, such as GraphDB, which permits the import and graphical visualization of the ontology after conversion to Turtle. Similarly, the Alloy, DOT and Prolog specifications can be used and processed by other applications such as AlloyTool, Graphviz and SWI-Prolog, respectively.

The compiler expansions have also introduced the translation to Prolog, which in addition to SWI-Prolog can be used to verify the restrictions introduced in the language. The system can also be used as a querying system for knowledge recognition and the user can further developed the generated file to define composed relations allowing the inference of implicit knowledge. The conversion of the ontology to Alloy, enables the use of the Alloy tool to perform ontology static model checking. Such expansion has allowed the creation of models based on the ontology rules, providing a fast and easy method to perform ontology verification.

Finally, a Website to publicize the application was created, providing tutorials, examples and templates along with the compiler to enable an easier start for new users of the language.

8.2 FUTURE WORK

In the future it would be important to perform an usability test of the language and its compiler. Since it would allow the retrieval of more metrics to evaluate the performance of the application.

It could also be created a compiler application to perform the translation from the generated formats back to OntoDL+. That way, it would be possible to use OntoDL+ as an intermediary language for translation. Already existing ontologies created in languages covered by the compiler could be translated and maintained using OntoDL+. It would enable already existing system to have a smoother transition to OntoDL+, as well as provide more means to work with ontologies by connecting languages with different capabilities. The language could also be extended in order to include the declaration of axiomatic expressions.

BIBLIOGRAPHY

- António Abelha, Cesar Analide, José Machado, José Neves, Manuel Santos, and Paulo Novais. Ambient intelligence and simulation in health care virtual scenarios. In *Working Conference on Virtual Enterprises*, pages 461–468. Springer, 2007.
- Krzysztof R Apt. Logic programming. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, 1990:493–574, 1990.
- Cristiana Araújo. Building the Museum of the Person Based on a combined CIDOC-CRM/ FOAF/ DBpedia Ontology. Master’s thesis, Universidade do Minho, Braga, Portugal, December 2016. MSc dissertation.
- Anindya Basu, Mark Hayden, Greg Morrisett, and Thorsten von Eicken. *A language-based approach to protocol construction*. Cornell University, 1997.
- Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, 1(1999):75–88, 1999.
- Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.99. Available online: <http://xmlns.com/foaf/spec/>, 2014. 20 February 2021.
- William J Brown, Raphael C Malveau, Hays W McCormick III, and Thomas J Mowbray. Refactoring software, architectures, and projects in crisis, 1998.
- David Bruce. What makes a good domain-specific language? apostle, and its approach to parallel discrete event simulation. *Kamin [43]*, pages 17–35, 1997.
- Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. The electrom analyzer: model checking relational first-order temporal specifications. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 884–887, 2018.
- Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and uml class diagrams annotated with ocl. *Software & Systems Modeling*, 14(1):5–25, 2015.
- Greg Dennis and Rob Seater. Session 1: Intro and logic. https://alloytools.org/tutorials/day-course/s1_logic.pdf, a. Accessed on 18.05.2021.
- Greg Dennis and Rob Seater. Session 2: Language and analysis. https://alloytools.org/tutorials/day-course/s2_language.pdf, b. Accessed on 18.05.2021.

- Greg Dennis and Rob Seater. Session 3: Static modeling. https://alloytools.org/tutorials/day-course/s3_static.pdf, c. Accessed on 18.05.2021.
- Arie Van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.
- Sabina Falasconi, Mario Stefanelli, and NJI Mars. A library of medical ontologies. In *Proceedings of ECAI94 Workshop on Comparison of Implemented Ontologies*, pages 81–92. Citeseer, 1994.
- Miguel Alexandre Ferreira and José Nuno Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In *Brazilian Symposium on Formal Methods*, pages 153–169. Springer, 2009.
- Anuradha Gali, Cindy X Chen, Kajal T Claypool, and Rosario Uceda-Sosa. From ontology to relational databases. In *International Conference on Conceptual Modeling*, pages 278–289. Springer, 2004.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Design Patterns. *Elements of reusable object-oriented software*, volume 99. Addison-Wesley Reading, Massachusetts, 1995.
- Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.
- Thomas R Gruber. Toward principles for the design of ontologies used for knowledge sharing? *International journal of human-computer studies*, 43(5-6):907–928, 1995.
- Robert M Herndon and Valdis A Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14(6):803–809, 1988.
- Richard B Kieburtz, Laura McKinney, Jeffrey M Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society, 1996.
- Dimitrios S Kolovos, Richard F Paige, Tim Kelly, and Fiona AC Polack. Requirements for domain-specific languages. In *Proc. of ECOOP Workshop on Domain-Specific Program Development (DSPD)*, volume 2006, 2006.
- Robert Kowalski. Predicate logic as programming language. In *IFIP congress*, volume 74, pages 569–544, 1974.

- William Kuechler, Vijay Vaishnavi, and William L Kuechler Sr. Design [science] research in is: a work in progress. In *Proceedings of the second international conference on design science research in information systems and technology (DESRIST 2007)*, pages 1–17, 2007.
- David A Ladd and J Christopher Ramming. Two application languages in software production.
- Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, et al. Dbpedia—a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 6(2):167–195, 2015.
- Luís Lima, Paulo Novais, Ricardo Costa, José Bulas Cruz, and José Neves. Group decision making and quality-of-information in e-health systems. *Logic Journal of the IGPL*, 19(2): 315–332, 2011.
- Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 373–383, 2016.
- Neil J Manson. Is operations research really research? *Orion*, 22(2):155–180, 2006.
- Mika Mantyla, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 381–384. IEEE, 2003.
- Ricardo Giuliani Martini and Pedro Rangel Henriques. Bridging the gap between bdme and ontome. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI'16)*, volume 1, pages 487–491, Oct 2016. doi: 10.1109/WI.2016.79.
- Ricardo Giuliani Martini, Giovanni Rubert Librelotto, and Pedro Rangel Henriques. Formal description and automatic generation of learning spaces based on ontologies. *Procedia Computer Science*, 96:235–244, 2016.
- Luís F Martins, Cristiana Araújo, and Pedro Rangel Henriques. Digital collection creator, visualizer and explorer. In *8th Symposium on Languages, Applications and Technologies (SLATE 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- Vijay Menon and Keshav Pingali. A case for source-level transformations in matlab. In *Proceedings of the 2nd conference on Domain-specific languages*, pages 53–65, 1999.
- Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

- Emerson Murphy-Hill and Andrew P Black. An interactive ambient visualization for code smells. In *Proceedings of the 5th international symposium on Software visualization*, pages 5–14, 2010.
- José Nuno Oliveira. Program design by calculation. <https://www4.di.uminho.pt/~jno/ps/pdbc.pdf>. Accessed on 05.06.2021.
- Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, and Kalyanmoy Deb. Multi-criteria code refactoring using search-based software engineering: An industrial case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3): 1–53, 2016.
- José Carlos Ramalho, 2020. URL <https://www4.di.uminho.pt/~jcr/Transfers/prc2020/mapa.ttl>.
- Ivo Serra and Rosario Girardi. A process for extracting non-taxonomic relationships of ontologies from text. 2011.
- Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.
- Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of systems and software*, 56(1):91–99, 2001.
- David B Stewart. Twenty-five most common mistakes with real-time software development. In *Proceedings of the 1999 Embedded Systems Conference (ESC'99)*, volume 141, 1999.
- Maria de La Salette Teixeira. Adequa, a platform for choosing games suitable to students' profile. Master's thesis, University of Minho, 2021.
- Arie Van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.
- Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
- Vangelis Vassiliadis, Jan Wielemaker, and Chris Mungall. Processing owl2 ontologies using thea: An application of logic programming. In *OWLED*, volume 529, 2009.

