Universidade do Minho
Escola de Engenharia

Nuno Alexandre Ramos de Carvalho

OML
Ontology Manipulation Language

Dezembro, 2008

Universidade do Minho
Escola de Engenharia

Nuno Alexandre Ramos de Carvalho

OML
Ontology Manipulation Language

Tese de Mestrado em Informática

Trabalho efectuado sob a orientação do
Professor José João Almeida

Dezembro, 2008

O M L
Ontology Manipulation Language

Nuno Alexandre Ramos de Carvalho
(smash@cpan.org)

Dissertação submetida à Universidade do Minho para obtenção do grau de Mestre em Informática, elaborada sob a orientação de
José João Almeida

Departamento de Informática
Escola de Engenharia
Universidade do Minho

Braga, 2008

Abstract

Ontologies are a common approach used in nowadays for formal representation of concepts in a structured way. Natural language processing, translation tasks, or building blocks for the new web 2.0 (social networks for example) are instances of areas where the adoption of this approach is emerging and quickly growing.

Ontologies are easy to store and can be easily build from other data structures. Due to their structural nature, data processing can be automated into simple operations. Also new knowledge can be quickly infered, many times based on simple mathematics properties. All these qualities brought together make ontologies a strong candidate for knowledge representation. To perform all of these tasks over ontologies most of the times custom made tools are developed, that can be hard to adapt for future uses.

The purpose of the work presented in this dissertation is to study and implement tools that can be used to manipulate and maintain ontologies in a abstract and intuitive way. We specify a expressive and powerful, yet simple, domain specific language created to perform actions on ontologies. We will use this actions to manipulate knowledge in ontologies, infer new relations or concepts and also maintain the existing ones valid. We developed a set of tools and engines to implement this language in order to be able to use it. We illustrate the use of this technology with some simple case studies.

Resumo

Ontologias são uma opção muito utilizada hoje em dia para representar formalmente conceitos de uma forma estruturada. Processamento de linguagem natural, tarefas de tradução, ou componentes associados à web 2.0 (redes sociais por exemplo) são instâncias de áreas onde a adopção desta aproximação está a emergir e a crescer rapidamente.

Ontologias são fáceis de armazenar e podem ser facilmente construídas a partir de outras estruturas de dados. Devido à sua natureza estruturada, o processamento de dados pode ser automatizado em operações simples. Além disso pode ser inferido novo conhecimento rapidamente, muitas vezes baseado em propriedades matemáticas simples. Todas estas qualidades em conjunto fazem das ontologias fortes candidatas para a representação de conhecimento. Na maior parte dos casos, para executar este tipo de operações, são desenvolvidas ferramentas costumizadas à medida que podem ser difíceis de adaptar para uso futuro.

O objectivo do trabalho apresentado nesta dissertação é estudar e implementar ferramentas que podem ser utilizadas para manipular e manter ontologias de uma forma abstracta e intuitiva. Especificamos uma linguagem de domínio específico simples, no entanto expressiva e poderosa para efectuar operações sobre ontologias. Vamos usar estas operações para manipular o conhecimento em ontologias, inferir novas relações ou conceitos e também para manter os existentes válidos. Foram desenvolvidas um conjunto de ferramentas e motores que implementam esta linguagem de modo a que possamos utilizá-la. Ilustramos o uso desta tecnologia com alguns casos de estudo simples.

Acknowledgments

I would like to thank to the following people that in many ways contributed for this work:

- My special thanks to my teacher and supervisor José João Almeida, for all the ideias, discussion and time spare.
- Many, many thanks to Alberto Simões for all the reading, discussion, inputs and help with everything.
- Thanks to José Almeida and Alberto Simões again, for all the tools and modules that were already available and were used in this work.

Preface

This document is a master thesis in Computer Science (area of Natural Language Processing) submitted to University of Minho, Braga, Portugal.

Document structure

Chapter 1 introduces the subject, defining the basic concepts and ideas used in the remaining document.

Chapter 2 presents some background on the concepts and approaches currently used and a brief overview of the state of the art concerning these subjects.

Chapter 3 describes the specification of the domain specific language that will be implemented to perform actions on ontologies.

Chapter 4 describes the tools and engines developed in order to implement the language specified, in Chapter 3.

Chapter 5 illustrates the use of the domain specific language with the tools developed and described in Chapter 4.

Chapter 6 concludes this dissertation, discussion and analysis of the work done. Explores some new tracks we can explore in future works.

Some complementary information is presented on the appendixes:

Appendix A shows the domain specific language grammar.

Appendix B presents a brief introduction to Camila notation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Knowledge is power.

Sir Francis Bacon

Knowledge representation has always been a challenge for science. From ancient philosophers to the most recent software engineers different ways were found to approach this problem, and several ways to represent knowledge over different domains emerged. From the 1970's neural networks or more heuristic question-answering systems to the later, more formal, computer language representation, different solutions have been adopted. Although they can all be very interesting and have their specific advantages, we are more interested here in the ones adopted by computer science.

Naturally more than one solution exists in nowadays to address the problem of knowledge representation. Several different areas use techniques suited to the specific problems they study. For example natural language processing tools for linguists do not represent knowledge in the same way that NASA satellites store information gathered by deep space observation. In other words, different goals usually demand different representations.

Today we have a wide variety of approaches to represent knowledge. And for each of this approaches we traditionally have a wide variety of solutions. Meaning that for different, well defined sets of domains, different representations can be used. It does not mean that there are good of wrong ways to represent things, it just means that there are representations more suitable to perform some kind of tasks than others. Another problem that naturally emerges here is when we start sharing knowledge. The continuous need to integrate heterogeneous systems in today's globalizing Internet aggravates the need of an easier interoperability between systems. This means that different systems need to understand, or at least understand the rules, that other systems use to represent information.

If we want to share our knowledge among others in the community, we must make sure that we are representing knowledge using the same rules, and in the same context. A couple of standards and drafts exist today, but there is not a general understanding in the community about which standard or language to use. What happens most of the times is that methods for representing knowledge and methods for maintaining and manipulate that knowledge, are custom made for most solutions. Although there is a clear interest in the community for the adoption of more standards and related technologies.

With the growing of these standards use, not only the standards achieve a more mature development stage, but also there is more motivation to work in other tools and solutions that use them. With better standards, and related tools, we hope that more developers will be adopting them on their solutions. This is also a clever way to work together for better and quicker interoperability between systems. Talking the same language is the first step for understanding between systems.

1.1 Introducing Ontologies

Ontologies is one of the many solutions that science uses to try to represent knowledge. Although, this study has started with Greek philosophers, today's ontology is a burgeoning field, involving researchers from the computer science, philosophy, data and software engineering, logic, linguistics, and terminology domains[28], transversely to many sciences. But there's no doubt that the term is one very important key word in today's computer science.

An ontology can be defined has an explicit specification of a conceptualization [10]. This definition can bring up a couple of more philosophical discussions, mainly because of the definition of conceptualization itself, so let us try to narrow this down to something more suitable to our needs. Let us assume that an ontology is an engineering artifact constituted by a specific vocabulary used to describe a certain reality, plus a set of explicit assumptions regarding the intended meaning of the vocabulary words. Therefore, in the simplest case, an ontology describes a hierarchy of concepts related by relationships[11]. Now, this is nearer to the reality we are aiming for.

Since, ontology is a comprehending concept and it is used among many sciences it can be hard to find an accurate definition, but by now we should have a clear idea of what we are talking about. We will continue this discussion and the benefits of using them in Chapter 2.

1.2 Motivation

There's an odd misconception in the computing world that writing compilers is hard. This view is fueled by the fact that we don't write compilers very often. People used to think writing CGI code was hard. Well, it is hard, if you do it in C without any tools.

Allison Randal

Our main goal during this work is contributing to the use of systems featuring ontologies or associated technologies. By studying and using existing solutions, and also creating new ones. Our intention is to deploy a complete system to work with ontologies. Where we could easily be able to accomplish the following tasks:

- Create new ontologies from other sources.
- Perform operations over ontologies.
- Maintain ontologies valid.
- Easily share and reuse ontologies.

Creating new ontologies can be a more or less trivial operation depending on the language we are using to write the ontology itself. The main problem here is being able to convert ontologies between formats. Most of the times the data is already in some kind of specific format, or some kind of markup language. So, the process of building an ontology from these data sources is always the same. If it is always the same then it can be automatically processed. This means that we can have a set of tools prepared and ready, to shift information from these data sources into out structured ontologies.

The second thing we are looking for, is a simple way to manipulate ontologies, by manipulating we understand execute some pre defined actions whenever we found a specific pattern in a ontology. Since ontologies are mainly build with concepts and relationships, they can grow very fast, thus the need to be able to maintain the ontology valid. Traditionally there are a set of rules that need to be enforced for each ontology so that the knowledge in that domain is still true. This maintenance task can be hard, we are sure that this manipulation approach can help in building more practical solutions for this family of problems.

Once we spent effort on creating ontologies and finding ways to maintain them valid, we want to share them between systems as often as possible. Or maybe reuse them across a different set of applications. There is no need to go through all that hard work again. We want to always consider the use of an ontology a valuable asset, not another source for extra work.

This complete system is our "carrot on a stick", and we definitively believe that this system can be useful for everyone working with this kind of technology. And a valuable asset to gain new enthusiasts for this approach to data representation.

Ontologies are already being used today in some interesting areas, which helps us prove their valuable contribution. Natural language processing is one, more traditional, example of these areas. A more modern example can be the new semantics web for the web 2.0. Social networks are a good example of system that more often relies on the use of structured knowledge. Whatever the problem is, this system can help to minimize the implementation burden of such complex solutions.

1.3 Methodology

This work intends to focus more on the maintaining and manipulation part of the system described in the previous section. In order to deploy the necessary tools for these tasks we will follow this methodology:

1. Create a domain specific language to describe operations.
2. Create engines and programs that would be able to calculate operations described in the specific language.
3. Create tools that can apply the results described in a program.
4. Use the created tools in a couple of case studies.

1.4 Outline

A brief outline of the remaining of this document, which is divided in five more chapters:

Chapter 2

In this chapter we try to discuss the needed concepts for correctly understanding this work. We also present other solutions to some of the problems talked about.

Chapter 3

In this chapter we describe the complete specification of the language created to manipulate ontologies.

Chapter 4

In this chapter we describe the tools and programs created to compile and execute programs written used the language described in Chapter 3.

Chapter 5

In this chapter we illustrate the use of the domain specific language in some concrete operations over some simple ontologies.

Chapter 6

In this final chapter we discuss the work done and results. We also enumerate some tasks to improve this work.

Appendix

The appendix shows us the complete grammar in BNF format for the language described in Chapter 2, and a paper with an introduction to Camila notation.

Chapter 2

Background

As we have seen in the previous chapter, some concepts we are going to discuss in this work can be hard to define. The aim of this chapter is to review most of the necessary concepts and definitions that are needed for a better understanding of this work. We also do a brief analysis of the current technology being used around this subject and related tools.

2.1 Ontologies

The term ontology has it's origin in the field of philosophy. Ontologies are one of the solutions found in computer science to represent knowledge about a well defined domain in a structured way. Ontologies can be used to represent knowledge about any kind of domain or area of interest. The use of the term ontology in computer science was first introduced in the area of artificial intelligence reasoning[18]. An ontology was used to represent the things that existed in a given domain. Actually, in a very abstract way, the idea still persists today. We use an ontology to represent our domain, and we do that by representing everything that exists in that domain.

Another important term that we have been using but have not yet defined is *domain*. An ontology is always an artifact on a given domain. Again, this term is used in a wide range of sciences which can make it harder to define. But, we can say that a domain is a way of refereeing a particular well defined area of knowledge. Sometimes this knowledge can may not be clearly bounded[14]. From the Oxford English Dictionary: ”A sphere of thought or action; field, province, scope of a department of knowledge, etc.”[1]

During this work we will assume the following definition: a domain ontology is an engineered artifact that informally defines concepts from a specific domain, representing and organizing them as conceptualizations which a set of systems working cooperatively with each other agree to share[14].

Sometimes we used other structures that can belong to the ontology family, but most of the cases they are quite distant cousins. Nevertheless they can still be very useful, when we need some kind of conceptualization, but not that rigid. Some examples of these structures are[17]:

- Glossaries are basically a list of terms and definitions.
- Thesaurus are networks of well defined interrelations, or associations, between terms. Given a particular term, a thesaurus will indicate which other terms mean the same, which terms denote a broader category of the same kind of thing, which denote a narrower category, and which are related in some other way.
- Taxonomies are traditionally structures that arrange terms into groups and subgroups based on predetermined rules.

When using this broad family of structures, most of the times, we can take advantage of tools that were designed to work with ontologies.

These are all very interesting structures conceptually, now we need to find ways to represent this structures in our traditional computer systems. Of course we could think on several ways to represent this, but that is not the idea. Since one goal and advantage of the use of this structures is to share knowledge, we must agree on rules to represent these ontologies. So that other systems can know how to make use of the stored information. We could always choose a specific representation for our ontology and distribute the structure used to represent the knowledge along with the ontology itself. This approach would result on thousands of heterogeneous representations, and still had the problem of sharing knowledge between different representations.

In the next chapter we will address this problem and illustrate some common ways to solve it.

2.2 The Art of Representation

Formal symbolic representation of qualitative entities is doomed to its rightful place of minor significance in a world where flowers and beautiful women abound.

Albert Einstein

There are several ways to represent and therefore be able to store, for later use, ontologies. Some of them are more suitable to some kind of particular tasks, other are well defined published standards. There are quite some publications trying to emerge a standard representation for ontologies. These are some examples of families of languages that can be used to describe ontologies or some well defined subsets. These are also the standards actually more used and well known.

You can also note that most of them use some kind of XML notation. This is mainly a portability issue, it makes information exchange between different systems easier.

2.2.1 OWL

The Web Ontology Language (OWL) is a family of languages for publishing and sharing ontologies on the World Wide Web[13]. This language is mainly developed and maintained by World Wide Web Consortium (W3C). The OWL specification includes the definition of three variants:

- OWL Lite, supports basic needs of a classification hierarchy and simple constrains.
- OWL DL (Description Logic), supports maximum expressiveness.
- OWL Full, meat for maximum expressiveness and syntactic freedom of RDF.

OWL is intended to provide a language that can be used to describe the classes and relations between them that are inherent in Web documents and applications[19].

A small example of something expressed in OWL using abstract syntax:

```
Ontology(
 Class(pp:animal partial restriction(pp:eats someValuesFrom(owl:Thing)))
 Class(pp:duck partial pp:animal)
 Class(pp:cat partial pp:animal)
)
```

Another example, now written in RDF/XML syntax:

```
    <rdf:Description rdf:about="#Huey">
        <rdfs:comment><![CDATA[]]></rdfs:comment>
        <rdfs:label>Huey</rdfs:label>
        <rdf:type>
            <owl:Class rdf:about="#duck"/>
        </rdf:type>
    </rdf:Description
```

Remember that the examples illustrated here are not complete. OWL is a very complex language to use, and representations can quickly become complicated and confused. Of course this complexity translates in a much more accurate representation. Ontologies are use among many sciences, OWL might be hard to use for someone that is not trained in computer science. This could became an obstacle for the spread of the language between different communities. This complexity should not be considered as a disadvantage, it allows to implement complex data structures that can smoothly be exchanged between heterogeneous systems.

Just recently the OWL working group published the OWL 2 Web Ontology Language. A extension of the previous version adding even more features.

2.2.2 SKOS

SKOS or Simple Knowledge Organization Systems is another family of languages that can be used for expressing the basic structure and content of concept schemes. It is is published and maintained by the W3C Semantic Web Best Practices and Deployment Working Group. SKOS can be used to easily create a very familiar subset of ontologies, some examples are thesaurus, taxonomies and terminologies[21].

A simple example of this language representation, also in RDF/XML:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:skos="http://www.w3.org/2004/02/skos/core#">

  <skos:Concept rdf:about="http://www.my.com/#canals">
    <skos:definition>A feature type category for places
such as the Erie Canal</skos:definition>
    <skos:prefLabel>canals</skos:prefLabel>
    <skos:altLabel>canal bends</skos:altLabel>
    <skos:altLabel>canalized streams</skos:altLabel>
    <skos:altLabel>ditch mouths</skos:altLabel>
    <skos:altLabel>ditches</skos:altLabel>
    <skos:altLabel>drainage canals</skos:altLabel>
    <skos:altLabel>drainage ditches</skos:altLabel>
    <skos:broader rdf:resource="http://www.my.com/#hydrographic%20structures"/>
    <skos:related rdf:resource="http://www.my.com/#channels"/>
    <skos:related rdf:resource="http://www.my.com/#locks"/>
    <skos:related rdf:resource="http://www.my.com/#transportation%20features"/>
    <skos:related rdf:resource="http://www.my.com/#tunnels"/>
    <skos:scopeNote>Manmade waterway used by watercraft
     or for drainage, irrigation, mining, or water
power</skos:scopeNote>
  </skos:Concept>
</rdf:RDF>
```

SKOS creates very extensive and overwhelming representations. It provides a framework for expressing knowledge structures in a machine understandable way. SKOS is a very powerful vehicle already being used in many situations instead of OWL. A good example is the new directory environment being developed in the UK: SWED[1]. Which uses SKOS to represent some thesauri. In fact, this site also uses OWL to publish some ontologies which can show that both language can be used together[20].

2.2.3 Topic Maps

Topic Maps is a specification that provides a grammar and a model for representing the structure of information resources[24]. A simple example of this representation:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE topicMap SYSTEM "xtm1.dtd">
<topicMap xmlns="http://www.topicmaps.org/xtm/1.0/" xmlns:xlink="http://www.w3.org/1
    <topic id="st-music">
        <baseName>
            <baseNameString>Music</baseNameString>
        </baseName>
    </topic>
</topicMap>
```

[1]http://www.swed.org.uk

This approach represents information using topics, a topic can represent any fact or concept (for example, cities, countries, etc.). Besides topics associations and occurrences are also used to represent information. Associations, as the name points out, are used to represent relations between topics. Every one of this constructors can have types.

2.2.4 Biblio::Thesaurus

This module was initially created to provide a set of tools to maintain thesaurus files. We already discussed how a thesaurus can be defined as a sub set of an ontology. But this module has grown and now is prepared to work with more abstract and complex structures, like ontologies for example. It still maintains the name, but that is bound to change in the future.

The internal representation for the ontology follows a subset from ISO 2788. Which means that can interact with other sources that follow the same standard. Note that the module was changed to work with more complex structures, and the standard defines standard features to be found on thesaurus files. Which means that things may not work right from the start. An example of the ISO representation looks like:

```
Animal
NT cat, dog, cow
   fish, ant
NT camel
BT Life being
```

This module has already been successfully used to translate other resources into ontologies. The same module can already be used to manipulate information, but we will see that in the next section. [27]

A well defined API allows the manipulation and access to various information in a very simple way. Adding or deleting information can be as simple as:

```
$ontology->addTerm('term');
$ontology->addRelation('term','relation','term1',...,'termn');
```

Clearly is a very different approach form the other representations discussed before, which has advantages and disadvantages. We will choose this representation to query and access ontologies when running programs in our domain specific language. We will explain this choice in Chapter 4.

2.2.5 RDF

RDF is World Wide Web Consortium data model that most often is used with XML [3]. Being a model means it needs a transport language. XML is the most common choice because is a flexible, portable and expandable language. An example of RDF/XML notation:

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/"
  xmlns:dc="http://purl.org/dc/elements/1.1/">
<rdf:Description rdf:about="http://en.wikipedia.org/wiki/Tony_Benn">
<dc:title>Tony Benn</dc:title>
<dc:publisher>Wikipedia</dc:publisher>
                <foaf:primaryTopic>
                     <foaf:Person>
                          <foaf:name>Tony Benn</foaf:name>
                     </foaf:Person>
                </foaf:primaryTopic>
</rdf:Description>
</rdf:RDF>
```

The idea is to use this pair as an general-purpose language for representing information in the Web. This is mainly used in current semantic web applications.

2.2.6 CycL

One of the firsts, if not the first, language to aim for knowledge representation was CycL. This formal language is mainly used by the Cyc knowledge based. Cyc is a project to create a comprehensive ontology and knowledge base of everyday common sense knowledge. The language itself is very peculiar, for example, the predicate:

```
 (#$genls #$Tree-ThePlant #$Plant)
```

states that *"All trees are plants"*. The applications for a common sense database are unimaginable, since user behavior models to simulations.[16]

2.3 Converting Representations

Another interesting problem that naturally emerges among so many different representation possibilities, is the tools that implement conversions between representations. In this section we briefly introduce some of these tools.

2.3.1 Generating RDF Models from LDAP directories

LDAP[2] is a common database technology for storing information on a directory system. This paper[6] presents a methodology for creating RDF models form LDAP directories. The semantic web world being able to pull information from LDAP directory resources is a big motivation for this kind of work. Although the author does not give a conclusion on the conversion results, it clearly states how easy it was to implement a small proof of concept. We might assume that the converting results were, at least, satisfactory.

2.3.2 Thesauri to SKOS

Mos of the conversion research and tools belong to this family: given a simpler structure like a thesauri, XML or database, for example, build a more complex representation. This work[31] is one of these examples, it tries to convert thesauri into a SKOS representation. Although, some lacks of the SKOS's model, according to the authors, most case studies went very well, which helps to prove that such conversions can be done.

2.3.3 Biblio::Thesaurus

This module can be used to to some simple conversions. For example, very often such things like taxonomies are stored in plain text file and use indentation to subclass terms. This simple text notation is very simple to parse and is possible to automatically create a more complex structure, an ontology for example, to store the same data. These family of tools can be quickly implemented with this module. Of course it is also possible to parse more complicated structures, or even a thesauri and convert it to an ontology.[2]

[2]Lightweight Directory Access Protocol

2.4 Manipulation Approaches

Since there are several ways to represent ontologies, there are also different approaches to manipulate them. Several software packages offer methods to change and manipulate information in a ontology. Once we achieve agreement on which representation to use for our ontology, we immediately have the benefit of using previously developed tools. This is a clear advantage over having to develop from scratch, all the actions and changes that we need to perform on our specific ontology.

We now give some examples of this manipulation tools.

2.4.1 Protégé

Protégé[3] is an open-source platform that provides a suit of tools for building knowledge driven applications based on ontologies. It has a specific extension to work with OWL. This extension allows for a visual editing of information. Figure 2.1 illustrates the this visual OWL editor.

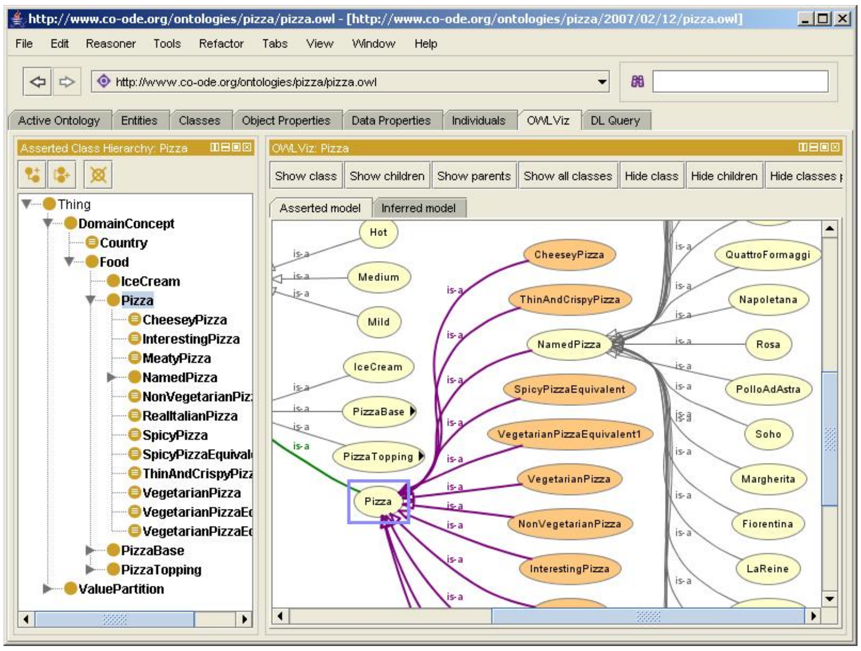


Figure 2.1: Protégé OWL editor.

[3]http://protege.stanford.edu/

Currently stable version of Protégé is version 3.3.1. The OWL editor is very complete and the user interface is rich and filled with options. There are a lot of very interesting features. Browsing the ontology in a graphical interface, and a lot of validations and OWL specific operations and class creation are some examples[12]. This editor is actually very user friendly and has prove it can be very useful. Of course it can be hard to use this software if you are aiming for computer generated operations. There is also a project in Google Code for a plugin for Protége to edit and create some artifacts represented in SKOS. This plugin is only available for version 4.

This project is quite settled and has a quite large community of users, which shows that new releases are to be expected in the near future, with new features. In can prove to be a go bet for a OWL editor for humans to use. Also runs on all the most common platforms including Linux based systems.

2.4.2 Jena Framework

Jena[4] is a framework for Java for building semantic web applications. This framework include a wide set of classes for using in Java development. Among many things, this framework has an interface called *OntModel* that can be used with other tools in the framework as interfaces to underlying models, written in OWL for example. Although, we did not use this framework for any development yet. The documentation promises very interesting features that can be really useful.

There are quite a lot of examples of research works using this technology. For example, ontology reusability[30] is a good example of a good use for this framework.

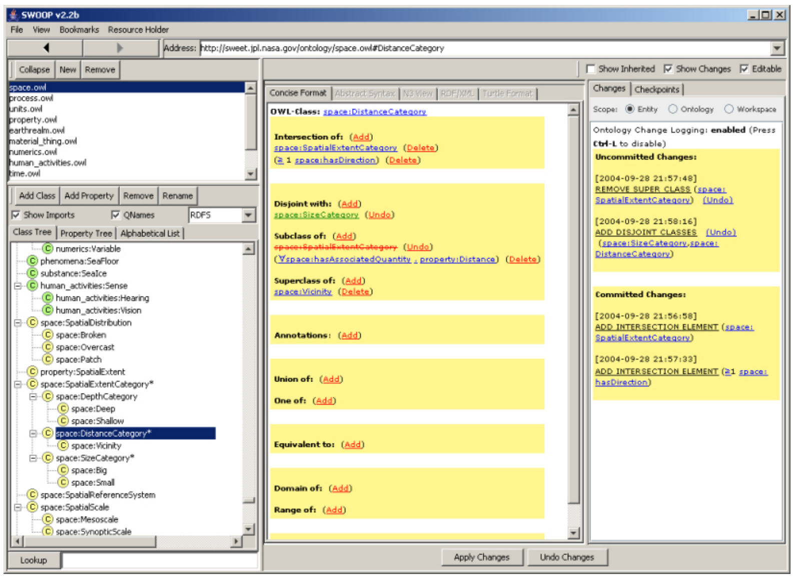
2.4.3 SWOOP

SWOOP[5] is another tool for creating and editing OWL ontologies. This project is also hosted on Google Code. This tool has a nice look and feel and it is very intuitive. Simple interface with concise operations over ontologies and a plug-in option for quick development of new features. Figure 2.2 illustrates this tool.

Unfortunately the current release is still in beta version.

[4]http://jena.sourceforge.net/

[5]http://code.google.com/p/swoop/



Address: http://sweet.jpl.nasa.gov/ontology/space.owl#DistanceCategory

☐ Show Inherited ☑ Show Changes ☑ Editable

Scope: [● Entity] ○ Ontology ○ Workspace

OWL-Class: space:DistanceCategory

Intersection of: (Add)
space:SpatialExtentCategory (Delete)
(≥ 1 space:hasDirection) (Delete)

Disjoint with: (Add)
space:SizeCategory (Undo)

Subclass of: (Add)
~~space:SpatialExtentCategory~~ (Undo)
(∀space:hasAssociatedQuantity . property:Distance) (Delete)

Superclass of: (Add)
space:Vicinity (Delete)

Annotations: (Add)

Union of: (Add)

One of: (Add)

Equivalent to: (Add)

Domain of: (Add)

Range of: (Add)

Ontology Change Logging: **enabled** (Press **Ctrl-L** to disable)

Uncommitted Changes:

[2004-09-28 21:57:48]
REMOVE SUPER CLASS (space:SpatialExtentCategory) (Undo)

[2004-09-28 21:58:16]
ADD DISJOINT CLASSES (Undo)
(space:SizeCategory,space:DistanceCategory)

Committed Changes:

[2004-09-28 21:56:58]
ADD INTERSECTION ELEMENT (space:SpatialExtentCategory)

[2004-09-28 21:57:33]
ADD INTERSECTION ELEMENT (≥1 space:hasDirection)

Figure 2.2: SWOOP ontology editor.

2.4.4 ThManager

ThManager[6] is an open source tool that is able to manage thesauri stored in SKOS, allowing their visualization and edition[15]. Figure 2.4 illustrates the ThManager thesaurus concept editor

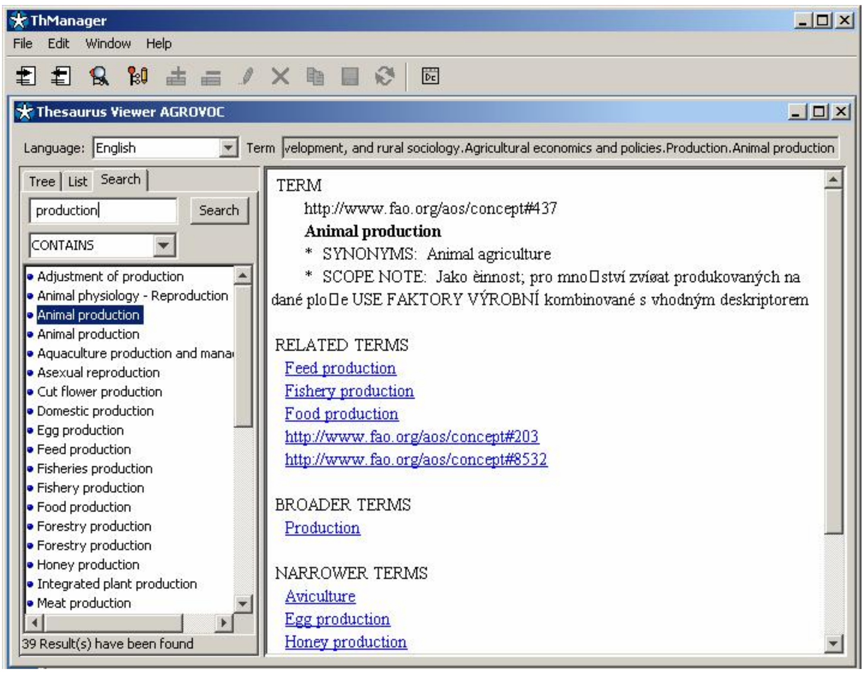
This is an open source projected, developed in Java for Windows or Unix. It has multilingual support. It allows to browse the different terms in hierarchical or alphabetical order, there is also a searching tool. Although it is not what we might call eye candy editor it sure implements the basic features needed for many operations. Also, it is very simple to use.

2.4.5 OWL Visual Editor

OWL Visual Editor[7] is, as the name clearly shows, a visual editor for OWL.

Unfortunately we were not able to start this editor to take a look. A problem with Java, so we will have to settle with the screen shoot.

[6]http://thmanager.sourceforge.net/
[7]http://owlve.sourceforge.net/



Language: English Term velopment, and rural sociology.Agricultural economics and policies.Production.Animal production

production

CONTAINS

- Adjustment of production
- Animal physiology - Reproduction
- Animal production
- Animal production
- Aquaculture production and mana
- Asexual reproduction
- Cut flower production
- Domestic production
- Egg production
- Feed production
- Fisheries production
- Fishery production
- Food production
- Forestry production
- Forestry production
- Honey production
- Integrated plant production
- Meat production

39 Result(s) have been found

TERM
http://www.fao.org/aos/concept#437
Animal production
* SYNONYMS: Animal agriculture
* SCOPE NOTE: Jako èinnost; pro mno□ství zvíøat produkovaných na dané plo□e USE FAKTORY VÝROBNÍ kombinované s vhodným deskriptorem

RELATED TERMS
Feed production
Fishery production
Food production
http://www.fao.org/aos/concept#203
http://www.fao.org/aos/concept#8532

BROADER TERMS
Production

NARROWER TERMS
Aviculture
Egg production
Honey production

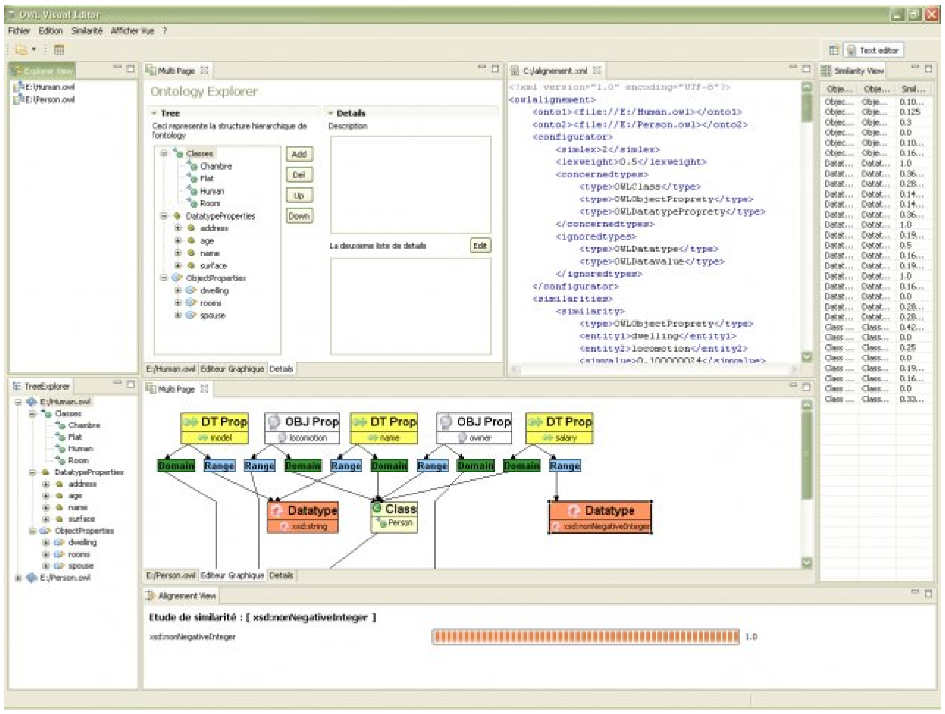
Figure 2.3: ThManager thesaurus concept editor.

2.4.6 Biblio::Thesaurus

We are also introducing this module as a manipulation tool. In the same way that we talked earlier, about using Java classes to deploy interfaces for OWL models access, we can also use this module to access an ontology. We can only access ontologies in a representation that the module understands. But the module has a very rich API[8] that can be used to do many elaborated operations over ontologies. Another plus for these approaches, in opposition to the graphical tools, is that operations can be automatically created and executed without human intervention.

An example of some of this module skill to manipulate ontologies is demonstrated during this work, since this is the module used by our domain specific language to perform actions over ontologies.

[8]Application Public Interface



```
<?xml version="1.0" encoding="UTF-8"?>
<owlalignement>
    <onto1><file://E:/Human.owl></onto1>
    <onto2><file://E:/Person.owl></onto2>
    <configurator>
        <simlex>2</simlex>
        <lexweight>0.5</lexweight>
        <concernedtypes>
            <type>OWLClass</type>
            <type>OWLObjectProprety</type>
            <type>OWLDatatypeProprety</type>
        </concernedtypes>
        <ignoredtypes>
            <type>OWLDatatype</type>
            <type>OWLDatavalue</type>
        </ignoredtypes>
    </configurator>
    <similarities>
        <similarity>
            <type>OWLObjectProprety</type>
            <entity1>dwelling</entity1>
            <entity2>locomotion</entity2>
            <simvalue>0.100000024</simvalue>
```

Etude de similarité : [xsd:nonNegativeInteger]
xsd:nonNegativeInteger 1.0

Figure 2.4: OWL Visual Editor.

2.4.7 SquishQL

We are not sure this language qualifies as a manipulation tool but sure is interesting enough to be mentioned here. SquishQL is a language, modeled after SQL, that can be used to query RDF providers[22].

An query example written in SquishQL:

```
SELECT ?title, ?description, ?name
WHERE
(?libby, <foaf:mbox>,
<mailto:libby.miller@bristol.ac.uk>) ,
(?paper, <dc:contributor>, ?libby),
(?paper, <dc:title>, ?title) ,
(?paper, <dc:description>, ?description) ,
(?paper, <dc:contributor>, ?someone) ,
(?someone, <foaf:name>, ?name)
USING foaf for <http://xmlns.com/foaf/0.1/>
```

Regarding the graphical editors we could go on illustrating more editors but it would be hard to find any feature that would not exist in the ones already visited. With more or less features or tools, they are all very alike. Table 2.1 summarizes the tools described for easier reference.

Tool	**Version**	**Description**	**Platforms**	**Formats**
Protége	3.3.1	graphical editor	All	OWL+SKOS
Jena	2.5.6	Java classes	All (w/ Java)	-
SWOOP	beta	graphical editor	All	OWL
ThManagaer	2.0	graphical editor	All (w/ Java)	SKOS/RDF
OWL VE	1.1.0	graphical editor	Linux/Source	OWL
SquishQL	-	SQL-ish language	-	RDF

Table 2.1: Summary of ontologies editing tools.

2.5 Interesting Case Studies

This section aims to present some successful and interesting research topics or work, around the use of ontologies or related technologies. This helps us see how far the use of ontologies can go, and that this artifact knows no science boundaries. Of course, it all ends up to computer related sciences to actually implement them, but the use of them is vast.

A methodology for the creation of entailment-base question answering system can easily take advantage of ontologies. In this particular work ”An user-centred ontology-and entailment-based Question Answering system” [9] an domain ontology was created and populated with cinema information. The data was stored in OWL/RDF format. Then random users were asked to query the ontology data base and questions were grouped based on similar information requests. These groups are then added to the information data base and a textual implication model uses this information and grammatical deductions to answer new queries.

Now to get a bit away from the linguistic examples, let us analyze a very interesting use of ontologies. ”The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines” is a project to develop clinical guidelines. The domain ontology for this case specifies concepts, such as drugs, diseases, patient findings, tests, and clinic visit types. A special purpose language was created for physicians to query the data base. Protégé, which we talked about a bit earlier, was used to create intuitive tools that health care operators can use[26].

One of ontologie's valuable asset is the ease of information share. Ontolingua takes this serious, it is a distributed collaborative environment to browse, create, edit, modify, and use ontologies[8]. This system allows for the achieving consensus on common shared ontologies between geographically distributed groups. Creating big ontologies, big because of complex domains that can be represented, can be a time-consuming process, this family of services can help share this work load between many users and to benefit everyone involved in the process.

Not only within the research scope ontologies are used. it's notorious benefits already reached the commercial world. Lists of wine properties are already available from commercial Web sites such as http://www.wines.com that customers can use to browse wine characteristics. Another fancy commercial ontology can be seen at http://www.unspsc.org/, which is used to classify products or services [23].

Our final case study shifts the subject again, the human genome is one of the 21st century technologies. The Gene Ontology Consortium[9] is a collaborative effort to create several ontologies representing various biological related information. Although this specific case works in representing roles of the gene products within an organism, there are several resources for biologists that are using ontologies or related artifacts. The Schulze-Kremer ontology for molecular biology or The TAMBIS Ontology[10] are other examples. Biology research rarely starts from scratch, previous knowledge is always used before starting new investigations. Ontology based systems are being used within the community to provide knowledge input to databases and applications. Also, the kind of data we are talking about is very complicated and complex, ontologies make it easy to provide services that information sharing[29].

There are a lot more examples out there of successfully ontologies use, either still in development or already at at commercial level. This situation only motivates even more the study and research around these artifacts. In the next chapter we start describing a language that was created in the scope of this work to help express ontology manipulation operations.

[9]http://www.geneontology.org/

[10]http://www.cs.man.ac.uk/ stevensr/tambis/

Chapter 3

OML Specification

The goal was to specify a language that will be able to act on ontologies. For now we are not interested in implementation details, we are more concerned on the language specification which we can use to manipulate and maintain ontologies. We will call this language OML[1]. The main idea here is that we want to look for a specific pattern on the ontology, and then execute some action. We will call a pair consisting of a pattern and an action a rule. The patterns can be very simple, but can quickly grow and became less intuitive. Actions are simply operations we want to execute over terms or relations, maybe add or remove a specific relation. Or, in a more complicated rule, execute some arbitrary code that can do produce an arbitrary side effect.

Please note, that the remaining of this chapter aims to be the complete spec for the language but it is presented in a very illustrative way, with lots of examples. This is done with the intent to present a wide range of language statements that can be used, and make it easier to read. Also during this chapter we will be using a simple ontology with knowledge in the domain of geography for these examples. For a more detailed description of the information in this ontology refer to Chapter 5.

3.1 Design Goals

Some design goals were taking into consideration when crafting the grammar and when language signals were chosen:

- The language needs to be simple: simple to use, not to complicated expressions or statements, simple to understand.

[1]Ontology Manipulation Language

- It needs to have a clear and well defined syntax, no need to overwhelm the user with lots of complicated symbols or signals, just the essential ones.

- It needs to be powerful enough to express complicated patterns. Although, the language should be simple it must allow the representation of complicated patterns which often need to be found.

- The statements need to be expressive, we should be able to write statements that are close to natural language and are easy to understand by themselves.

- Easy to understand, finally we did not want to complicate the language in such a way that it would be more complicated to learn the specification itself than the patterns we need to describe.

Since this kind of technology is transverse to many sciences, we must keep in mind that many people outside the computer science scope might be using this language.

Other fact that led to the choice of some of the design goals enumerated was the analysis of some of the representations discussed in chapter 2. Some of them can be very syntax obfuscating, there are many syntax details that nned more effort to write than the information we want to store itself. Although these details often enrich the language and allow for interesting features, they can easily become overkill and obfuscate the information outside the syntax scope, that should be the center point of view for the end user. Hence force, the continued struggle for a clean, simple and yet, expressive syntax.

3.2 Specification

First of all our language needs to have some basic simple notations. The most basic building block for ontologies are terms. We can then represent relations between terms. We can use this relations to build patterns. Patterns that can exist in a given ontology. Sometimes it is possible to have more than one instance for a given pattern, but we will get back to this later. After patterns are found we want to execute actions. We group patterns and actions to describe a set of operations to be executed. We can put together an arbitrary number of related patterns and actions to create a program. The following subsections describe every one of this components in detail. But before that, a brief description on data types regarding OML.

3.2.1 Data Types

Before starting describing statements or expressions let us take a brief look on OML's data types. There is no need to complicate things regarding this issue when defining the language. All terms, relations containers, etc. syntax nneds to follow two rules:

- Named variables or relations are described using strings, a sequence of alphanumeric characters, if there is the need to include any white spaces single or double quotes need to be used. For example:

  ```
  name
  'another name'
  "yet another name"
  ```

- Terms or relations containers, variables, always have names that do not have any white spaces, and are strings.

  ```
  $container
  $variable3
  ```

The only restriction to this situation is when you use a `sub { ... }` block in the action section of a rule. The use of this block will be explained later. But if this block is used the standard rules for data types apply inside the block. Since the language that can be used inside the `sub` statement is Perl, Perl's language syntax needs to be used.

3.2.2 Programs

A program is a block that can be executed, which means a program is a list of rules. Programs are written in plain text files. Every rule is executed in order but the results of each rule will only be visible at the end of the program execution. The rules are executed in the same order that they were written in the program. A rule consists of four elements in the following order:

1. A pattern section, this section of the rule describes the pattern that we will be looking for in the ontology.
2. The special sign `=>`, which is a equals sign (`=`) followed by a greater than sign (`>`).

3. An action section, this section describes the actions to be performed when the rule's pattern is found.
4. A rule is always finished with a single dot (`.`).

To the left of the special signal (`=>`) we can find the pattern and the action to the right. A rule looks like:

```
<pattern> => <action> .
```

We can have any number of rules in a program. And can also use the sign `#` in the begging of the line to mark lines as comments. Lines marked as comments are not processed.

```
# this line is not processed
```

Next we will be describing how to write patterns in detail.

3.2.3 Patterns

Patterns represent one or more relations, or terms, or any combination of both that can be found in the ontology. First things first, we can be searching the ontology for a single term:

```
term(<term>)
```

Which means that this pattern will be considered found if exists at least one term named `<term>`. For example, in our geography ontology we could use the following pattern:

```
term(Braga)
```

To verify the existence of a term called `Braga`.

We could also be looking for a single relation:

```
rel(<relation>)
```

Which means that this pattern will be considered found if exists at least one relation name `<relation>`. For example, in our ontology we could use the following pattern:

```
rel(city-of)
```

To verify if a relation name `city-of` existed. As most ontologies relate terms using relations we can search patterns that look like:

```
<term1> <relation> <term2>
```

The pattern represented here will be considered found if there is a relation named `<relation>` that relates the terms named `<term1>` and `<term2>`. For example:

```
Braga city-of Portugal
```

is a pattern that evaluates true if the term `Portugal` is related to `Braga` by a relation named `city-of`.

In the examples used so far, we always gave names to things, a relation named `<relation>` or a term named `Braga`. But we can use containers instead of named terms to specify patterns. A container term, which basically is a variable that can take any term value, is distinguished by a named term, by starting with the dollar signal (`$`). Named term:

```
Braga
```

Named container:

```
$city
```

Using named containers we can specify patterns that can match more than once in the ontology. For example we can write patterns like this:

```
$city city-of Portugal
```

This patterns represents all the relations in which the term named `Portugal` is related to any term by a relation named `city-of`. In a more natural language this pattern represents all the cities in Portugal.

More than one named container can be used at the same time. The following pattern for example:

```
$city city-of $country
```

represents all the relations that exist between the list of terms in named containers `$city` and `$country` that are related by a relation name `city-of`. It is the list between all possible combinations of cities for each country that exists in the ontology.

Besides using containers for terms, we can also use named containers for relations. This means we can write something like this:

```
Braga $relation Portugal
```

This represents all relations that exists between the named terms `Portugal` and `Braga`. And even more dangerous, you can mix terms containers with relation containers, this way you can end up with patterns that look like this:

```
$term1 $relation $term2
```

This represents all the possible relations, in this ontology, between all possible terms.

Remember that we are still talking about patterns, which means that we only looked at ways of matching subsets of the ontology to take some action. Before looking at what we can do in the action block, let us look at some operators that can be used to combine terms, relations or both.

Entities	**Named**	**Containers**
terms	term(<term>)	term($term)
relations	rel(<relation>)	rel($relation)
facts	<term> <relation> <term>	$term1 $relation $term2

Table 3.1: Summary of basic patterns.

Binary Operators

There are two binary operators: `AND` and `OR`. Both operators can be used between any of the three different type of patterns talked before.

The `AND` operator can be used to collect a list of named terms that all need to exist to find a pattern:

```
term(<term1>) AND term(<term2>) [ AND term(<term3>) ... ]
```

For example, the pattern:

```
term(Braga) AND term(Guimaraes)
```

is evaluated has found only when terms named `Braga` and `Guimaraes` exist. This operator can be used for relations:

```
rel(<relation1>) AND rel(<relation2>) [ AND rel(<relation3>) ... ]
```

The `AND` operator can also be used for a list of relations between terms.

```
t1 r1 t2 AND t3 r2 t4 [AND ....]
```

This means that the pattern is only considered as found if the list of relations is found. In the following example:

```
Braga city-of Portugal AND Guimaraes city-of Portugal
```

the illustrated pattern will only be found if the ontology relates the term `Braga` with the term `Portugal` by a relation named `city-of`, and the term `Guimaraes` with the term `Portugal` by a relation named `city-of`.

The `OR` operator can be used to join a list of terms, relations or relations between terms using the same syntax as the `AND` operator. The only difference is that for the given list only one of the elements needs to be found for the entire pattern to be considered found. For example:

```
Braga city-of Portugal OR Guimaraes city-of Portugal
```

is a pattern found if the term `Braga` is related with the term `Portugal` by a relation named `city-of` or the term `Guimaraes` is related with the term `Portugal` by a relation named `city-of`. This same principle applies to a specific term or relation.

Both binary operators can be used between named containers. Which means that we can write patterns that look like:

```
$city1 city-of Portugal AND $city2 city-of Spain
```

This pattern represents all the patterns that have a term related with the term `Portugal` by a relation named `city-of`, and are related with the term `Spain` by a relation named `city-of`.

Unary Operators

The only unary operator that exists is the `not` operator. This operator can be used before any of the three already discussed instructions. The following pattern represents all the terms that are not the term named `term`.

```
not(term(<term>))
```

For example, the pattern:

```
not(term(Braga))
```

represents all the terms that are not named `Braga`. The same behavior for name relations:

```
not(rel(<relation>))
```

This pattern represents all the relations that are not named `<relation>`. Finally, we can use this operation to negate a relation between two terms:

```
not ( <term1> <relation> <term2> )
```

This pattern represents all the relations that do not relate the named terms `<term1>` and `<term2>` by a relation named `<relation>`.

This operator can also be used before expressions using any of the binary operator. This means that we can write a pattern like:

```
not(<term> <relation> <term> AND <term> <relation> <term>)
```

or

```
not(<term> <relation> <term> OR <term> <relation> <term>)
```

For a better understanding of this patterns use the following transformations:

```
NOT ( x AND y ) ==> (NOT x) OR (NOT y)
NOT ( x OR  y ) ==> (NOT x) AND (NOT y)
```

With this simple transformation we end up with patterns that were already illustrated and work exactly in the same way as before.

3.2.4 Actions

After being able to specify the patterns we are looking for in the ontology we need to describe the actions that are going to be executed if the pattern is actually found. The actions section of the rule is everything between the special sign `=>` and the terminating dot. This section is a list of operations that are going to be executed:

```
(operation1|sub1) [ (operation2|sub2) ... ]
```

You can notice that we are distinctly saying that any of my actions can be an `operation` or a `sub`. This is because there can be two exclusive types of operations:

- We choose to execute an operation from the list of operations that are already available. In this particular case the syntax to use is:

  ```
  <operation> ( <list of arguments> )
  ```

  This feature will be explained in more detail later, in chapeter 4.

- Or we choose to define our own operation, writing the complete code of the operation to be performed. In this case the syntax to use is:

  ```
  sub { <code> }
  ```

  `<code>` must be written in Perl. For anyone familiar with this language, what we are actually doing here is defining a new function that will be called later if the pattern did match.

A simple example of an action adding an operation from the defined table could look like:

```
add(Portugal official-lang-is Portuguese)
```

One example of an action using code to produce any side effects:

```
sub { print "I found a relation.\n"; }
```

This action would simple print a message informing that a relation was found. But we could get really complicated here, and start producing all kinds of side effects, for example:

```
sub {
  use DBI;

  my $dhb = connect(...);

  $dbh->do("INSERT INTO relations(...) VALUES(...)";
}
```

In this example, and remember the code here is not complete, it is for illustration purpose only, would connect to a data base and insert some data in a table regarding a relation that was found. Remember that our `sub` actions are actually Perl code, meaning that you can do whatever Perl allows you to do, which is pretty much about everything. You can even call another OML program as an action, and execute another set of rules on the same or new ontology, or change the existing ontology in execution time. Basically almost everything is possible here, one can always argue about the advantages or disadvantages of such freedom, but that is how it works, at least for now.

Another important aspect is that, when named containers are used those variables and names and instances are propagated to the action block. This means that a rule like this can be written:

```
$city city-of Portugal => add($city official-lang-is Portuguese).
```

In this rule we are looking for a pattern that represents all the terms that are related with the term `Portugal` by a relation named `city-of`. The container `$city` which contains this list of terms propagates to the action block were it can be used. This way, in the action block, we are executing an operation that adds a new relation to the ontology, that relates the term `Portuguese` with every term found by the pattern by a relation named `official-lang-is` This rule is self explanatory, we are adding information about the official language in every city found in Portugal.

We will illustrate more actions and analyze more actions examples in chapter 5.

3.3 The Grammar

We created a grammar that formally defines the syntax described through out all this chapter. The complete grammar for OML can be found in Appendix A in, BFN notation. The BNF notations can be used to define free context grammars where entities are defined in terms of other entities. Entities can be defined by combination of other entities, either by alternation or sequences of entities. The non-terminal symbols are all written in lower case, and terminal symbols are written in upper case. The axiom for our grammar is *pTree*.

Chapter 4

OML Implementation

We now have the domain specific language which we can use to specify operations we want to execute in a given ontology. In this next chapter, we will discuss the development of tools that allow the execution of programs written in this language.

So, first things first. In the optic of this work we created a package that used together with other packages, blindly to the user, can execute programs. A package is bundled with a great deal of many different things, some of these things are more important than others, so we will focus on each one at different stages.

Please note, that there were tools and modules that were started before the scope of this dissertation. Sometimes, those tools or modules were use to fill in gaps. And even improved, or features were added because of some specific needs.

`Thesaurus::ModRewrite` is the package responsible for running programs written in OML which was described in the previous chapter. The modules in this package were written in Perl, and follow the traditional object oriented paradigm. We will call it simply `ModRewrite` for the remaining of this document. Also, do not let the prefix `Thesaurus` deceive you, this module is to use in ontologies, not only thesaurus.

We choose to write these modules in Perl because of several reasons:

- There were already some tools and other modules that we will used to build our system, and these tools were already written in Perl. Writing all of them in the same language gives a clean, simple and free integration among every tool.
- Perl is a natural language, with great tools already implemented to cre-

ate new languages (a version of yapp for example). Yapp is a powerful tool for creating new compilers.

- Is widely available in most common operating systems with out out-of-the-box installations. Support for new modules installation is a core feature of the language, so you can't immediately installing new modules.

- It has a good support for modules, and related tools, like distribution.

For our ontology representation and manipulation we choose to use the module `Biblio::Thesaurus`, because of the following reasons:

- Is developed in Perl, as stated before, integration with other tools is easier.

- This module can be used to store ontologies, manipulate and even convert between some formats.

- The module has a rich API, that provides a set of useful functions that are going to be needed to solve some of the implementation problems.

- This module is very flexible, so we would be able to tweak some features, if needed, to implement some specific functionality.

- This would also be a good opportunity to test this module and see if there were any implementation flaws that were compromising the knowledge representation.

Figure 4.1 illustrates a very brief overview of this package architecture. We feed a program and an ontology to our module and after some compiling stages a final result is produced.

$$\begin{tikzcd}
 & Ontology \arrow[d] & \\
Program \arrow[r] & \boxed{ModRewrite} \arrow[r] & Result
\end{tikzcd}$$

Figure 4.1: Architecture overview.

We will formally define it as:

$$ModRewrite : program \times ontology \longrightarrow result \tag{4.1}$$

Our module, given an *ontology* and a *program* written in OML, will be able to execute it and produce a final *result*.

4.1 Design Principles

Before describing the architecture of the implementation in detail, we would like to review some of the design principles that were kept in mind during development.

- The solution itself needs to be very modular, every consisting part needs to be well defined by itself. This is useful because, we can later change the way one of this modular components works without having to change other modules. This allows for easy mechanics manipulation in core modules, and makes easier the job for third parties code contributions.

- Keep it simple, do not complicate tasks more than needed. Break big tasks in small tasks always as possible. This also helps to maintain a modular package as described in the previous point.

- Sometimes development was test driven. The package contains a well defined test suit, which can be used at all times if any of the components is broken or misbehaving. Sometimes tests were written before implementation, this was also a good indicator to measure feature development.

- A sense of abstraction is in order, we don't want to simply have a tool that runs and execute code, we want to have a serious of modules and engines that can be brought together to accomplish different tasks. In other words, we are not trying to build a compiler, we are trying to put together a set of pieces that can be easily used together to compile programs.

And of course we try to follow the most know common best practices as often as possible[4].

4.2 Architecture

The main module in the package is responsible for the most complex task: using all the other components together to execute a program written in OML. This core problem was divided in smaller tasks:

1. Parse the program and calculate a parsing tree. The parsing tree contains a set of patterns to look for, and actions to take if patterns are found.
2. Analyze the parsing tree rule by rule, and calculate the sets of patterns that are found. If a pattern is found and specifies more than one solution, calculate all the possible solutions. Build a new tree, that contains the possible solutions for the patterns found, and the actions to run.
3. Analyze the new tree, for each rule found, iterate every possible solution and run the appropriate actions.

In any case we can tell the module to output the corresponding tree, by setting the corresponding switches before program compilation.

We can say that our main module given a *program* and an *ontology* calculates a final *result*. The final result can be an ontology by itself, but since arbitrary side effects can be produced by the *program*, different types of results can be calculated.

$$\begin{aligned} ModRewrite &: program \times ontology \longrightarrow result \\ ModRewrite &= reactor \circ expander \circ parser \end{aligned} \tag{4.2}$$

This way the final result of our main module *ModRewrite* is the result of calling the *reactor* function after the *expander*, and the *expander* after the *parser* function. In the following sections we will describe the different components responsible for each task. We will also define the *parser*, *expander* and *reactor* functions.

4.2.1 Internals

So, how does it works internally? The core of this solution is divided in modules or specific sets of functions, responsible to compute the different stages described in the architecture section. Internally we have divided everything into well defined sections depending on which task is being resolved:

- A parser, which is responsible for executing task number one described in the architecture section. This is an independent module that can be re-factored if needed as often as possible.
- A set of well defined functions that are responsible for converting a parse tree into a new tree that can be used to execute actions. We will call this new tree a *diTree*.
- A reaction engine, which is mostly responsible for actually executing the actions defined in the program.
- A set of tools that use the functionality provided by the previous items to run programs.

As stated in the last item, everything is combined together in a couple of high order tools that allows us to run programs from a single invocation.

4.2.2 The Parser

The parser is one of the core modules and is responsible for parsing the program source. It behaves as most of the parsers do, it takes the source code in and creates a parsing tree. This task is described in figure 4.2.

$$\begin{tikzcd} program \arrow[r] & \boxed{Parser} \arrow[r] & pTree \end{tikzcd}$$

Figure 4.2: Parser overview.

This module, given a *program* builds a parsing tree, we will call the parsing tree build by the parser a *pTree*. We define this *pTree* as a list of *statements*:

$$pTree \;=\; statement^{\star}$$

$$statement \;=\; condBlock \times actionBlock$$

$$condBlock \;=\; nil + condition + binOp + unaryOp + term + relation$$

$$binOp \;=\; condBlock \times op \times condBlock$$

$$op \;=\; AND + OR$$

$$unaryOp \;=\; NOT \times condBlock$$

$$condition \;=\; term \times relation \times term$$

$$term \;=\; STRING + VAR$$

$$relation \;=\; STRING + VAR$$

$$actionBlock = action^{\star}$$

$$action = operation + SUB$$

A *pTree* is a list of *statement*s. Each *statement* can be several things:

- Nothing, a *statement* can be empty.
- A *condition*, in this case the pattern we are looking for is a simple fact. A *condition* is a three elements list: a *term*, a *relation* and another *term*.
- A *binOp*, which is a three elements list: a *condBlock*, followed by a binary operator (which can be an *AND* or an *OR*, followed by another *condBlock*.
- A *unaryOp*, is a *NOT* followed by a *condBlock*.
- A *term*, a *statement* can be a single *term*.
- A *relation*, a *statement* can be a single *relation*.

A *term*, as well as a *relation* can be one of two things:

- A *STRING*, in this case this string of characters represents the actual name of the *term* (or *relation*).
- A *VAR*, in this case we have a container (a variable) that can represent a set of *term* (or *relation*s) and the string of characters represents the name of the container.

An *actionBlock* represents the set of operations to be executed and is a list of *action*s. An *action* can be one of two things:

- An *operation*, in this case we are going to execute an operation from the pre defined operations table.
- A *SUB*, in this case the operation to execute is defined in a function supplied by the user who wrote the program.

In both these cases, arguments need to be passed to the functions that run the operation. We do not need to go into that the detail, just imagine the arguments are stored in the tree as children of the corresponding operations.

We have defined the necessary types to represent the structure which we need to build our *pTree*.

To build the parser we used the `Parse::Yapp`[5] module for Perl. This module is the Perl implementation of the traditional Bison[1] (also known as yacc in some architectures). Bison is used to build parsers from grammars. We have our grammar defined, we have our data structure, so the only thing we need now is to define the function that will run for each rule in the grammar. Our *parser* function given a *program* builds a *pTree*:

$$parser \ : \ program \ \longrightarrow \ pTree$$

The parser also uses a separated component, a function called *lexer*,

$$lexer \ : \ programText \ \longrightarrow \ symbol$$

The *lexer* is responsible for tokenizing the source fed to the parser. Given a piece of *programText* this function returns the next *symbol* found. When called, the *lexer* reads the code until it matches a previously defined regular expression and returns the token found. The program is being consumed during the process, so every time the lexer is called it continues the match from the last point that it returned. This function is called until the program ends.

Our *parser* function uses the *lexer* to know the sequence of symbols found in the program. While building the derivation tree. This derivation tree is build using the rules defined in the grammar.

Now we need to implement the parser for the grammar that is going to use the *lexer* function and the data types defined earlier. To undergo this complex, yet completely automated task, we will use *Parse::Yapp*. This module will build a new module that can be used to parse programs in our specific language. This is not near complete, our parser can only calculate if for a given program is there one and only one possible tree that can be built by deriving the non-terminal rules. The lexer is called by this module to return the next token found in the program source. By now our parser is not much more than a state machine. As we have discussed earlier, the goal of the parser is to build a *parsingTree*. To do this we add code to our grammar to be executed whenever a rule is used, this way we can build our tree during tree derivation. This way every time the state machine chooses a rule based on the next token returned by the lexer we execute some code. Most of the times this code is used to create a new node in the resulting tree,

[1]http://www.gnu.org/software/bison/

and also add some needed information to the newly create node. Let us look at a little example to bring some more light into the subject. Our grammar clearly states that a term, which is a non-terminal rule, can derived into two terminal symbols: a string or a container. This is described in BNF notation in our grammar as:

```
term : STRING | VAR
```

In our grammar module we started by defining this rule as:

```
term : STRING
     | VAR
     ;
```

This is correctly defined. Also anyone can state that the notation is a very similar to BNF, just to make the task of writing parsers easy. But this it is not a solution for the problem because we still need to return a parsing tree for everything to work. So, we will add some code to create those specific nodes, the term nodes:

```
term : STRING { +{'term'=>$_[1]} }
     | VAR { +{'var'=>$_[1]} }
     ;
```

The code that is going to be executed when each rule is reduced is written between `{` and `}` for each rule. Also, the last evaluated expression is returned. Therefore we just need to create an internal representation for this node and return it's reference so it can be added to the tree.

Finally our top rules return the whole parsing tree which is internally represented with and hash table with an element, the key for this element is the keyword `pTree` and the value of this element is the rest of the tree. This is done in our top rule `program`:

```
program : statement_list { +{ pTree=>$_[1] } }
        ;
```

The new hash table is the last evaluated expression, and `program` is the top rule, so this tree is the returned result after calling the parser. This tree is then used by the next module to continue computation.

The following example illustrates the building of the parsing tree. The next program, which consists in only one rule, adds to the ontology a relation named `city-of` between a container name `$city` and the term `Europe`, for each `$city` that is related with the term `Portugal` by a relation named `city-of`.

```
$city 'city-of' Portugal => add ($city 'city-of' Europe).
```

In figure 4.3 we can see a simplified version of the parsing tree that is build when running this program. The square boxes are all the non-terminal symbols that we need to follow before getting to the terminal symbols, which are represented here by the round corner boxes. The double line square box represents the starting symbol.

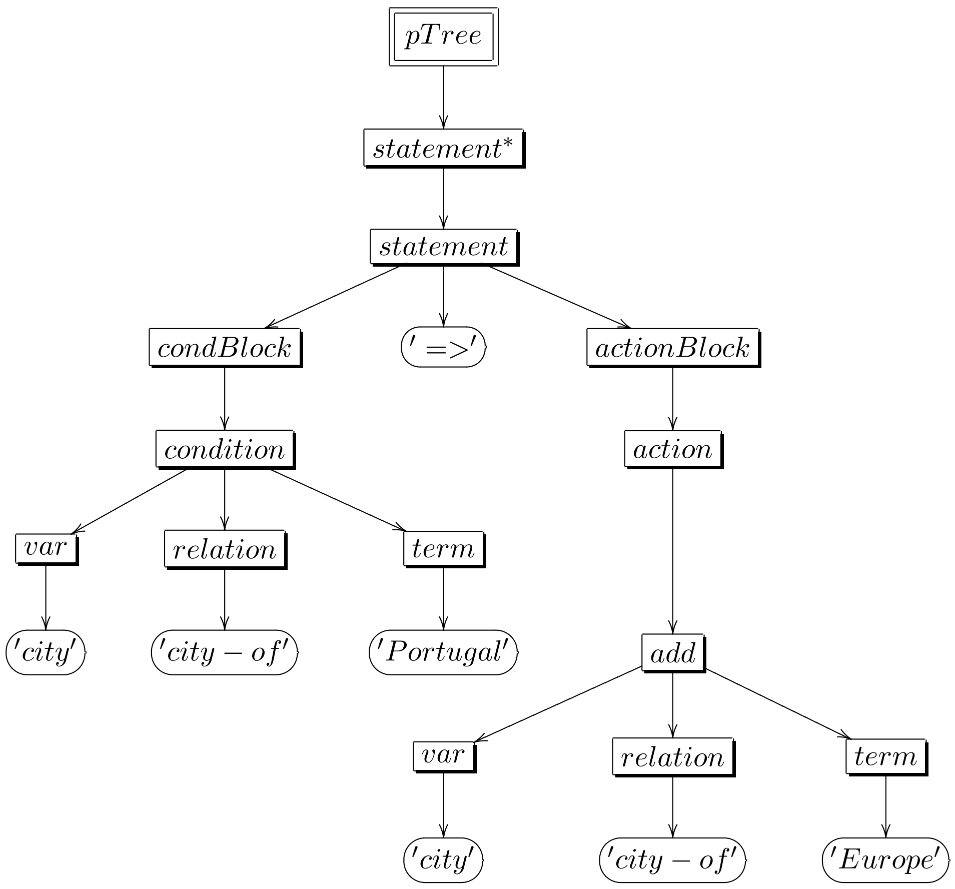


Figure 4.3: Parsing tree example for rule *$city 'city-of' Portugal => add ($city 'city-of' Europe).*.

So, from figure 4.3 we can see that from the non terminal symbol *pTree* we can build the tree that represents

Of course we now need an internal representation for this tree. This is illustrated next. This is the actual internal representation of our parsing tree, where the `{ }` represent anonymous feature sets and the `( )` represent anonymous sets. This is a simplified version, some information concerning the order of the rules for example was removed in order to be easier to read.

```
'pTree' => {
  'cond' => [
    { 'var' => 'city' },
    { 'relation' => 'city-of' },
    { 'term' => 'Portugal' }
  ],
  'action' => {
    'add' => [
      { 'var' => 'city' },
      { 'relation' => 'city-of' },
      { 'term' => 'Europe' }
    ]
  }
}
```

4.2.3 Tree Transformation Engine

The next module is responsible for converting the parsing tree in a tree with more information. Since in this case we are looking for patterns this engine calculates, for each pattern, two things:

- If the pattern was found in the ontology or not.
- If the pattern was found, calculate the list of solutions for the given pattern. This is needed because if named containers are used in the pattern, there can be more than one solution.

This module final result will be a new tree that we will can domain instantiated tree (*diTree*). In this tree all the patterns defined in program are replaced with the possible solutions that we found for those patterns. Therefore we define the function that does this transformation *expander*, that given a *pTree* returned by the parser module and an *ontology* builds a new *diTree*.

$$expander : pTree \times ontology \longrightarrow diTree$$

In the new *diTree* the patterns will be replaced for sets of instances that represent the solutions for that pattern. We define this new tree as:

$$diTree = distatement^{\star}$$
$$distatement = instBlock \times actionBlock$$
$$instBlock = instance^{\star}$$
$$instance = VAR \hookrightarrow STRING$$

A *diTree* is a list of *distatement*s. Each *distatement* is a pair that consists of a *instBlock* which represents the instances of the patterns we were looking for, and an *actionBlock* which is kept unchanged during this transformation. Although. for the *actionBlock* we still use the definition presented in the last section, we need to define the *instBlock*. An *instBlock* is a list of *instance*s. If the pattern was found in the ontology this list represents the list of instances that matched for the given pattern. Later we will illustrate this with an example.

We also need to define an ontology. Although much was discussed in chapter 2 we will for now adopt a rather simple formal definition:

$$ontology = fact^{\star}$$
$$fact = term \times relation \times term$$

An *ontology* is a simple list of *fact*s. Each fact is a three list element: a *term*, a *relation* and another *term*. Although this is far from the definitions discussed in earlier chapter, it is enough for the model we are illustrating here.

The *expander* module will iterate over all the rules in the *pTree* examining all the patterns. For each rule's pattern it creates a new node in the *diTree* that contains the *instBlock* which represents the solutions for the *condBlock* in the *pTree*. The *diTree* also stores the *actionBlock* from the *pTree*, this block is stored unchanged.

We define the *expander* function as:

$$expander : pTree \times ontology \longrightarrow diTree$$
$$expander(P, O) \overset{\text{def}}{=}$$
$$<distatement(setCalc(condBlock(p), O), actionBlock(p)) \mid p \in P>$$

The *expander* function creates a new *diTree*. To do this it creates a new *distatement* for each *statement* in the *pTree*. To create a new *distatement* the *condBlock* needs to be converted to a new *instBlock*. Which means that the pattern in the rule is going to be replaced with the instances for the pattern in the ontology. This calculation is made by the *setCalc* function:

$$setCalc : condBlock \times ontology \longrightarrow instBlock$$

$$setCalc(C, O) \overset{\text{def}}{=}$$

$$\begin{cases} \text{is-}nil(C) & \Rightarrow & <> \\ \text{is-}term(C) & \Rightarrow & handleTerm(C, O) \\ \text{is-}relation(C) & \Rightarrow & handleRel(C, O) \\ \text{is-}condition(C) & \Rightarrow & handleCond(C, O) \\ \text{is-}binOp(C) & \Rightarrow & handleBinOp(C, O) \\ \text{is-}unaryOp(C) & \Rightarrow & handleUnaryOp(C, O) \end{cases}$$

The *setCalc* function creates a new *instBlock*, this is done based on the argument *condBlock*. Actually this function does only the dispatch to the correct function to do the calculation based on which alternative is the *condBlock*. If the pattern only contained a single *term* then the *handleTerm* is called, for a single *relation* then the *handleRel* and so one for each of the possible alternatives for *condBlock*.

$$handleTerm : term \times ontology \longrightarrow instBlock$$

$$handleTerm(T, O) \overset{\text{def}}{=}$$

$$\begin{cases} \text{is-}string(T) & \Rightarrow & \begin{cases} T \in O & \Rightarrow & <\left(\begin{pmatrix} \texttt{"term"} \\ T \end{pmatrix}\right)> \\ T \notin O & \Rightarrow & <> \end{cases} \\ \text{is-}var(T) & \Rightarrow & <\left(\begin{pmatrix} T \\ \pi_1(f) \end{pmatrix}\right) \mid f \in O> \end{cases}$$

The *handleTerm* function calculates an *instBlock* for a given *term* and an *ontology*. The pattern we are looking for only contains a term, which means that there are only two possible solutions for the new *instBlock*:

- The term is a named term, this means that the final block will include the term if the term exists in the ontology.

- The term is a container, which means that the pattern represents all the terms for the ontology. This list is put together to return the instance block.

$$\begin{array}{l} \mathit{handleRel} : \mathit{relation} \times \mathit{ontology} \longrightarrow \mathit{instBlock} \\ \mathit{handleRel}(R, O) \overset{\text{def}}{=} \\ \quad \left\{ \begin{array}{rcl} \text{is-}\mathit{string}(R) & \Rightarrow & \left\{ \begin{array}{lcl} R \in O & \Rightarrow & <\left(\left(\begin{array}{c} \texttt{"rel"} \\ R \end{array}\right)\right)> \\ R \notin O & \Rightarrow & <> \end{array} \right. \\ \text{is-}\mathit{var}(R) & \Rightarrow & <\left(\left(\begin{array}{c} R \\ \pi_2(f) \end{array}\right)\right) \mid f \in O> \end{array} \right. \end{array}$$

The *handleRel* function works the same way that the *handleTerm* function, but instead oh handling terms handles relations.

$$\begin{array}{l} \mathit{handleCond} : \mathit{condition} \times \mathit{ontology} \longrightarrow \mathit{instBlock} \\ \mathit{handleCond}(C, O) \overset{\text{def}}{=} \\ \quad \left\{ \begin{array}{rcl} \textit{at least one is } \text{VAR} & \Rightarrow & \mathit{select}(\pi_1(C), \pi_2(C), \pi_3(C), O) \\ \textit{are all strings} & \Rightarrow & \left\{ \begin{array}{lcl} C \in O & \Rightarrow & <C> \\ C \notin O & \Rightarrow & <> \end{array} \right. \end{array} \right. \end{array}$$

The *hangleCond* function follows the same principle, the different here is that we have a three list elements of terms and a relation and any of these three elements can be a *STRING*, which means that the we know the name of this *term* (or *relation*). Or, it can be a *VAR*, which means that we have a container for this *term* (or *relation*). If all the elements are strings the resulting *instBlock* contains that *condition*, if the fact exists in the ontology, otherwise the resulting *instBlock* is an empty list. If any of the elements is a *VAR* we need to do a SQL style query to the ontology and select the relations that instantiate the pattern, this is handle by the *select* function.

$$\begin{array}{l} \mathit{select} : \mathit{term} \times \mathit{relation} \times \mathit{term} \times \mathit{ontology} \longrightarrow \mathit{instBlock} \\ \mathit{select}(T1, R, T2, O) \overset{\text{def}}{=} \\ \quad \textit{select the list of instances in the ontology for this relation} \end{array}$$

Think of the *select* function as an actual `SELECT` from a database, where the terms and relations are columns, and the names that are containers are replaced by a `*`. The function is not defined here because it would complicate and extend more this description than would help to define the implementation. Also, to make the model easier to illustrate and explain, some details were omitted, without which this function definition would be rather difficult.

$$\begin{array}{l} \mathit{handleBinOp} : \mathit{condition} \times \mathit{ontology} \longrightarrow \mathit{instBlock} \\ \mathit{handleBinOp}(C, O) \overset{\text{def}}{=} \\ \quad \left\{ \begin{array}{rcl} \text{is-}\mathit{and}(\mathit{op}(C)) & \Rightarrow & \mathit{cartesian}(\mathit{condBlock}(C, O), \mathit{condBlock}(C, O)) \\ \text{is-}\mathit{or\_}(\mathit{op}(C)) & \Rightarrow & \mathit{union}(\mathit{condBlock}(C, O), \mathit{condBlock}(C, O)) \end{array} \right. \end{array}$$

The *handleBinOp* function calculates a new *instBlock*, when the *condBlock* contains a binary operator. There are two possible situations here:

- The binary operator is an *AND*, in this case we need to intersect the two *condBlocks*. This is a particular join and is defined in the *cartesian* function.
- The binary operator is an *OR*, is this case we need to unite the two *condBlocks*. This is a traditional union and is defined in the *union* function.

$$\begin{aligned}
&union : instBlock \times instBlock \longrightarrow instBlock\\
&union(A, B) \stackrel{\text{def}}{=}\\
&\quad A \cup B
\end{aligned}$$

The *union* function simply joins two *instBlocks* using the traditional union set.

$$\begin{aligned}
&cartesian : instBlock \times instBlock \longrightarrow instBlock\\
&cartesian(A, B) \stackrel{\text{def}}{=}\\
&\quad \{cartAux(a, b) \mid a \in A, b \in B\}
\end{aligned}$$

$$\begin{aligned}
&cartAux : instance \times instance \longrightarrow instBlock\\
&cartAux(A, B) \stackrel{\text{def}}{=}\\
&\quad \begin{cases} dom(A) \cap dom(B) = \emptyset & \Rightarrow \quad A \dagger B\\ dom(A) \cap dom(B) \neq \emptyset & \Rightarrow \quad \underline{let}\ K = dom(A) \cap dom(B)\\ & \qquad \underline{in} \begin{cases} \forall x \in K \wedge A(x) = B(x) & \Rightarrow \quad A \dagger B\\ \forall x \in K \wedge A(x) \neq B(x) & \Rightarrow \quad \emptyset \end{cases} \end{cases}
\end{aligned}$$

The *cartesian* joins two *instBlocks*. For a given list of *instances* it calculates all the possible instance combinations, this way it returns a new *instBlock* with all the possible solutions for the pattern provided in the rule. The *cartAux* is just an auxiliary function for the *cartesian* function.

As described in the design principles the component that is used to calculate this tree is independent so it can be easily replaced. One thing that needs to be noted here is that this component needs to ask questions to the ontology. To do this we used the API functions provided by the module `Biblio::Thesaurus`. Examples of these queries are mostly the existence of a given relation or term in the ontology. We can change the ontology source that is being used as long as we are able to query the new source with the questions needed to implement the functions described in this chapter.

Let us now illustrate this calculation with a small example. A program that has only one rule which as the following pattern:

```
$city city-of $country => ...
```

can be used to calculate the container `$city` that is related with the container `$country` by a relation named `city-of`. In a more natural language this program will execute something, that we are not interesting in (`...`), for every city named `$city` that is a city on any `$country`. The figure 4.4 illustrates only the condition block that would be created for this pattern in the parsing tree.

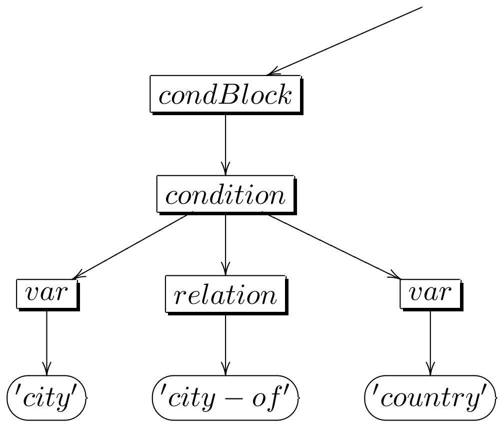


Figure 4.4: Condition block for condition *$city city-of $country.*

The tree returned by the parser will represent this information as:

```
...
  'cond' => [
    { 'var' => 'city' },
    { 'relation' => 'city-of' },
    { 'var' => 'country' }
  ],
...
```

We can see that the relation name is well defined `'city-of'`. But there are two variable containers named `$city` and `$country`. This is also indicated in the feature set show by the use of the keyword `var`, used as key. This engine is responsible for replacing this particular node with a new node that actually represents the instances for this pattern. The node that would represent that list of instances looks like:

```
...
  'inst' => [
    { 'city' => 'Braga', 'country' => 'Portugal' },
    { 'city' => 'Guimaraes', 'country' => 'Portugal' },
    { 'city' => 'Lisboa', 'country' => 'Portugal' },
    { 'city' => 'Porto', 'country' => 'Portugal' }
    ...
  ],
...
```

A list of all the possible solutions for each named container. In this particular example there are two containers: `$city` and `$country`. The *condBlock* that represented the pattern was replaced in the new *diTree* by a *instBlock*. This block represents the list solutions for the containers in the pattern. Each of this solution is a feature sets with the instances for all the containers found in the pattern.

After going through all the nodes in the parsing tree and for each one of them create the corresponding node in the domain instantiated tree, this engine returns this newly created tree. Our main module then handles control to the next engine which is responsible for actually executing the actions defined in the program for each rule. This engine is described in the next section.

4.2.4 Reaction Engine

In this stage we will use the *diTree* created during the transformation described in the previous section. This engine iterates through the tree and for each rule executes the actions defined. For each rule we can now have two possible options in the pattern section:

- An empty list which means that the pattern described in the rule was not found in the given ontology.
- The other option is to have a set. This indicates that the pattern was found and could have originated more that one solution, and so the action block needs to be executed once for each possible instance for the variables in the pattern.

The main function that implements this reaction engine (*reactor*) is defined as:

$$reactor : diTree \times ontology \longrightarrow result$$

$$reactor(T, O) \stackrel{\text{def}}{=} \begin{cases} \text{is-}nil(T) & \Rightarrow \quad nil \\ \quad \underline{else} & \Rightarrow \quad \underline{let} < h : t > = T \\ & \qquad\quad h1 = runAction(instBlock(h), actionBlock(h), O) \\ & \quad \underline{in} \;\; reactor(t, O) \end{cases}$$

The *reactor* function given a *diTree* and an *ontology*, for each rule in the *diTree* executes the *actionBlock*.

$$runAction : instBlock \times actionBlock \times ontology \longrightarrow result$$

$$runAction(I, A, O) \stackrel{\text{def}}{=} \begin{cases} \text{is-}nil(I) & \Rightarrow \quad nil \\ \quad \underline{else} & \Rightarrow \quad \underline{let} < h : t > = I \\ & \qquad\quad h1 = execute(h, A, O) \\ & \quad \underline{in} \;\; runAction(t, A, O) \end{cases}$$

The *runAction* function, for a given *instBlock*, an *actionBlock* and an *ontology* executes the *actionBlock* for each instance of the pattern found in the *instBlock*.

$$execute : instance \times actionBlock \times ontology \longrightarrow result$$

```
$execute(I, A, O) \stackrel{\text{def}}{=}$
    $\underline{let}$ $args = I$
        $<h : t> = A$
        $h1 = runCode(h, args, O)$
    $\underline{in}$ $runCode(t, args, O)$
```

$$runCode : action \times args \times ontology \longrightarrow result$$

$runCode(A, L, O) \stackrel{\text{def}}{=}$
execute action A passing L as arguments

The *execute* function runs each action in the *actionBlock* for an *instance*. This function is also responsible for fetching the needed variables from the *instance* to be used as arguments to the action being run.

The *runCode* function is the function responsible for actually executing the necessary code to perform the describe action. The only thing to note is that before executing the code we need to remember that have pre defined functions and user defined functions in the program. In either case it executes the code passing along the necessary parameters. The actions are always executed in the order they were written in the program. So, each of these actions can be any of two distinct types:

- A pre defined operation was specified, this means that the action chosen is from the table 4.1. For example, to add or delete a relation. There is a callback table defined in the package that is used to call the function for each operation. In this callback table the code that implements each function, can be changed according to any specific needs or another callback table can be used. This code can even be change in runtime.

- As described in the previous chapter, the other option is to have a standalone subroutine. This means that variables are set as needed, specially the ones concerning the iterated solution and the code written in the program is executed as normal code.

Operation	**Target**	**Arg. Number**	**Arg. List**
add	ontology	3	term,relation,term
del	ontology	3	term,relation,term

Table 4.1: Summary of pre defined operations.

Picking up in the example we have been using figure 4.5 illustrates the *condBlock* for an example rule.

We can see that the rule represented in this parsing tree used the pre defined operation *add*. We can also verify that the correct number of arguments were parsed. The add operation needs three arguments as we can see in table 4.1. So for this *actionBlock* we would run the *add* operation for each instance in the *instBlock*.

Using the last *instBlock* we have calculated:

```
'inst' => [
  { 'city' => 'Braga', 'country' => 'Portugal' },
  { 'city' => 'Guimaraes', 'country' => 'Portugal' },
  ...
],
```

as an example, we can see that the *actionBlock* will run as many times as instances exists. The first time it runs the arguments passed to the action will be a feature set like:

```
{ 'city' => 'Braga', 'country' => 'Portugal' }
```

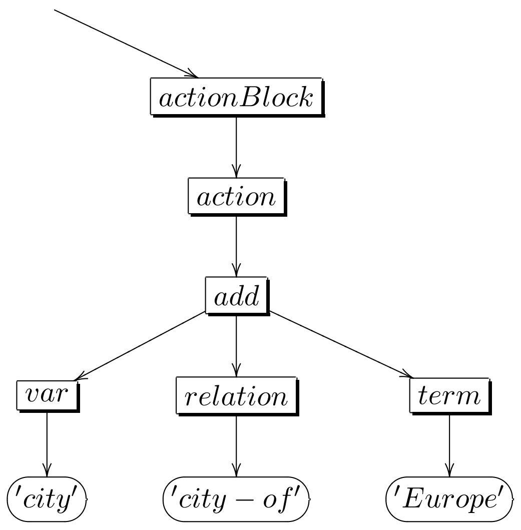


Figure 4.5: A *condBlock* example.

The second time it runs the arguments will be the feature set:

```
{ 'city' => 'Braga', 'country' => 'Portugal' }
```

The functions illustrated in this chapter were written in Camila. These functions aim to build a model to illustrate the current implementation of the OML interepreter, but remember it is simplified. Some details were omitted in order to keep it simple for a better and easier understanding. Regarding the notation used, there is a artile in Appendix B *Simple Camila Notation* that may help to undestand functions definitions.

4.3 Package

The package contains many things that were talked about earlier. In summary:

- A module written for yapp which implements the parser.
- A set of functions that are used to transform the tree as described in the architecture section.
- A set of function that are used to execute actions over the ontologies or arbitrary side effects.

- Some scripts that use all the engines together to execute programs and manipulate ontologies.
- An examples directory with a couple of examples.
- The test suite that can be used to validate that everything still works as expected.
- And finally every file has it's own documentation in POD format.

The package can be freely downloaded from CPAN[2], the main archive for Perl modules available today. The module can be installed using CPAN specific utilities, like `cpan` for example:

```
$ cpan Biblio::Thesaurus
```

Or, the package can be installed manually on any system that runs a Perl interpreter:

Download and unzip the package file:

```
$ wget .....Biblio-Thesaurus.tgz
$ tar zxvf Biblio-Thesaurus.tgz
```

Configure for building:

```
$ perl Makefile.PL
```

Do the building:

```
$ make
```

Run the test suit to make sure everything build correctly:

```
$ make test
```

Install the package:

```
$ make install
```

Note that in any of these two cases, there are some modules dependencies that need to be installed. If you use the CPAN tools to install the module this dependencies can be automatically handled for you, in the second case they need to be manually installed.

[2]http://www.cpan.org

4.4 Distribution

Some people tend to extrapolate that code produced during investigation projects in universities, often stays in universities and it is never released to the public. In this case, all the code and documentation included in the package can be freely downloaded under GPL (GNU General Public Licence)[3] from CPAN (Comprehensive Perl Archive Network)[4].

Keep in mind that this implementation is still in development and should not be used at production level. The direct link to the package page can always be found in:

`http://search.cpan.org/~smash/`.

[3]http://www.gnu.org/copyleft/gpl.html
[4]http://www.cpan.org

Chapter 5

OML by Example

5.1 Geography Ontology

To illustrate our OML implementation we created a very simple ontology. The knowledge stored in this ontology identifies a small set of information about geography, European geography mostly. Information about cities and some of their properties was more than enough to build some easy to read examples. A very small subset of the information in this ontology is illustrated in figure 5.1, just to give an idea of the kind of things we are talking about.

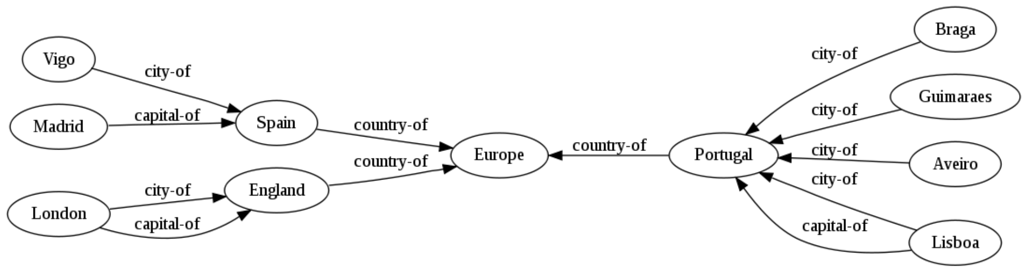
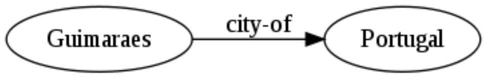


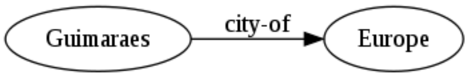
Figure 5.1: Geography ontology.

Although the reader of this work by now should already have a fairly accurate idea on how to use OML, there is nothing like a set of examples to consolidate how everything works.

Starting with simple things, for example, imagine you have a simple relation like described in figure 5.2(a). Which states that the term `Guimaraes` is related to the term `Portugal` by a relation named `city-of`.



(a) Existing relation.



(b) New relation.

Figure 5.2: Simple example.

In a more natural scope we would read that as *Guimaraes is a city of Portugal*. Given this we can also say that *Guimaraes is a city of Europe*, since Portugal is a country in Europe. We can write a small program in OML to add the new relation shown in figure 5.2(b), as shown in program 1.

Program 1 Add a simple relation.

```
Guimaraes 'city-of' Portugal => add(Guimaraes 'city-of' Europe).
```

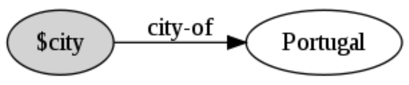
Program 1 will add the new relation described in figure 5.2(b) if the relation described in figure 5.2(a) exists. We can also execute this operation for any number of cities, for example, in a single program we could do something like, take a loot at program 2.

Program 2 Add more simple relations.

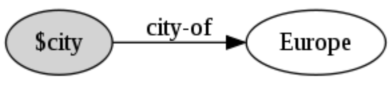
```
Guimaraes 'city-of' Portugal => add(Guimaraes 'city-of' Europe).
Braga     'city-of' Portugal => add(Braga 'city-of' Europe).
Porto     'city-of' Portugal => add(Porto 'city-of' Europe).
```

Although this is very simple it can be very overwhelming. The work involved in this approach would be the same as adding all the needed, or wanted, relations by hand. So, let us try something a little bit more elaborated. If Portugal is a country in Europe, then we can infer that all the cities that are in Portugal are also in Europe. Basically what we are looking for are relations that look like the relation show in figure 5.3(a).

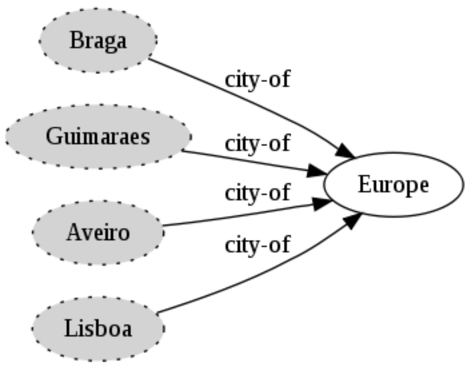
The shaded circle means represents a term container, which means that we will be looking for every term that is related to the term `Portugal` by



(a) Matching relations.



(b) New relations.



(c) Added relations.

Figure 5.3: Container example.

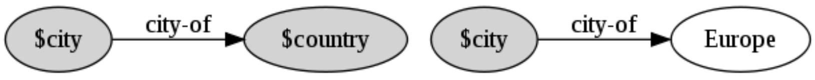
the relation `city-of`. We will call this term container `$city`. Now, the idea is for each `$city` that matches this pattern we will add a new relation that states that every city found is also a city of `Europe`, as shown in figure 5.3(c). Program 3 illustrates how do define this operation.

Program 3 Add a relation with a container.

```
$city 'city-of' Portugal => add($city 'city-of' Europe).
```

Program 3 is much more interesting that the earlier one, program 1. Since we do not care how many cities exist in our ontology, if that city exists and is in Portugal, then it is in Europe also. Also, we can run this program later just to make sure if new cities were added, they also have this relation.

This is a more sophisticated method of adding relations. But we can still improve our program. There are more countries besides Portugal in the ontology, and for each country there can be many cities. We can create a program to add relations for every of this cities. The pattern we will be looking is illustrated in figure 5.4(a). In a more natural language this pattern says that for every `city` in every `country`. And for every pattern like this that we found we want to add a new relation for that city, something like shown in figure 5.4(b).



(a) Matching relations. (b) New relations.

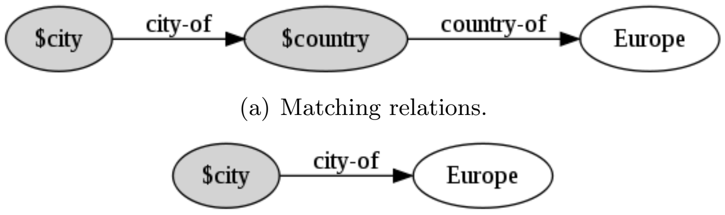
Figure 5.4: Container example.

Program 4 shows how to write this operation in OML. We will add a relation that states that `$city` is in Europe for each `$city` for every `$country` in our ontology.

Program 4 Add a relation for every city in every country.

```
$city 'city-of' $country => add($city 'city-of' Europe).
```

This program can make our life easier, because we can add the new information to the ontology in an automatic way. This is lees prone to human errors and also easier to maintain, since i can run the program every time a new country or city is added. We could have a problem if we started adding countries that not belong in Europe. In this case we would need to narrow down our solution range in our pattern. So, if someone was to add other continent for example the pattern we would be searching for would be something like the one described in figure 5.5(a).



(a) Matching relations.

(b) New relations.

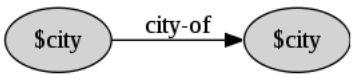
Figure 5.5: Container example.

Which means that we would be looking for every `$city` that belongs to every `$country` but with a restriction. That this `$country` is related with the term `Europe` by a relation named `country-of`, which naturally means that this `$country` is in Europe. And, for every `$city` add a new relation as shown in figure 5.5(b), which means that `$city` is in Europe. Program 5 shows how to implement this.

Program 5 Add a relation for every city in every country in Europe.

```
$city 'city-of' $country AND $country 'country-of' Europe
=>
add($city 'city-of' Europe).
```

There a lot of tricky operations that could be executed. We could loose fate in our information integrity, or maybe if new data was inserted by humans there could be mistakes. We could implement simple operations to do various tests. For example, a city can not be city of itself, which means that we could never have a `$city` related with itself by a relation named `city-of'`. For each case that we found with this pattern we could print a warning, or remove the relation on the fly. This pattern is shown in figure 5.6(a).



(a) Matching relations.

Figure 5.6: Relation for the same term.

This way we can search the ontology for invalid relations, we can run this program every day for example to verify the ontology.

Program 6 Check for invalid relations.

```
$city 'city-of' $city
=>
sub{ print "$city has an invalid relation"; }.
```

Program 6 shows how to implement this.

We hope that this set of illustrated examples has given you a clear idea on how to write programs in OML. In the next chapter we will discuss some conclusions and future work possibilities.

Chapter 6

Final Notes

6.1 Conclusion

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

During this dissertation, work was done in order to achieve a complete system to manipulate ontologies. A new domain specific language called OML was created. The tools required to run programs written in OML were also implemented. The results of using programs written in OML were very satisfactory. Although it is a very simple language, programs tend to be very expressive and can be used to implement a wide range of heterogeneous operations.

Regarding the new domain specific language OML and related tools several achievements were accomplished. Some conclusions that can be taken:

- We successfully specified a domain specific language that can be used to describe patterns that can be matched against an ontology. These patterns can represent more than one solution, because variables can be used in the pattern to serve as containers for any term or relation. Patterns can be very expressive and can go from very simple facts representation to complicated relations between many facts or terms.

- OML also allows the representation of actions that can be performed on ontologies. There can be a a range of pre defined operations, or arbitrary user code can be executed for each operation. Pre defined tables are interesting because they solve most of the traditional problems. But being able to produce any kind of side effect, by writing user code broader a lot the applications of the technology.

- Grouping patterns and actions allows us to form rules. Rules allows us to manipulate information in an ontology. These rules can be used to add the new information, by adding new facts or relations. New information can be used to enrich our ontology and can be simply calculated from existing data. OML programs allow for this kind of operation to be executed. Rules can also be used to write programs that check if some list of proprieties is valid in the ontology. This is also a very important point. Most ontologies represent a domain, the domain information needs to be validated most of the times against a set of proprieties. Rules can be created to automatically enforce this properties regularly, even without any kind of human effort. This lessens the effort of maintaining ontologies a lot.

- The programs created are very easy to understand, there is not much overhead in learning the language. Syntax is very simple. Not much previous programming knowledge should be required to use it. This technology can be quickly dominated by anyone without any computer science specific training. This is very important because the continuous growth of ontology adoption among many different sciences, and not necessarily all computers science related.

- An independent module was created to parse this new language. This module is responsible for creating a parsing tree for a given program. The parser module is easy to understand and extend. No rigid or strict rules were enforced. Also, it can be entirely replaced with any other parser as long as s parsing tree as described in chapter 4 is created. The implementation of this module was very interesting because of all the traditional grammar based language parsers typical problems.

- We wrote a module that can build *domain instantiated trees*. These trees represent the instances found in an ontology for a given pattern. Besides using the parsing tree, this module also uses the ontology to calculate this tree. This was by far the most time and effort expensive module to build. It starts by a couple of simple cases but the use of containers for names and relations quickly raises the complexity of the code. Again, this module can be re-factored or replaced by another approach as long as Ir can calculate a *domain instantiated tree* given a parsing tree and an ontology.

- We implemented an engine capable of executing operations in an ontology, or producing any arbitrary side effects. This engine uses the *domain instantiated trees* and the ontology to execute the described operations. Alongside, dispatch tables were created, this allows for the use of pre defined operations in a program. Dispatch tables contain operations that can be directly called from the language. This is a very interesting asset because allows the creation of dispatch tables with most common operations for a given area or subject. You can also create a new function to run in real time. This is also interesting because allows actions that do anything you might need. Perl is used to write these user defined functions.

- Finally, we put everything together to execute programs written in OML in a simple set of example tools. Since everything is modular and can be used separately we can now build high order functions that use these ones as building blocks. We put together some interesting tools, but we are sure that there are still ways that the modules can be used that we have not yet think off, we are guessing that this will happen when the right problem comes along. This is the beauty of it, we can keep crafting more complex and complex applications as problems need them, not the other way around. Meaning we start with the simple things and grow in complexity from there. We do not start with complex tools and languages to tinker simple applications to solve simple problems.

The study and analysis of current technologies in this area contributed for some of the decisions made during the language specification and implementation. From this study some conclusions were taken:

- There are actually a wide range of means for ontologies computer representation. OWL, SKOS, Topic Maps and Biblio::Thesaurus are instances of solutions that can be used to represent knowledge in ontologies, or even some close subset of structures (thesauri or taxonomies for example).

- OWL, SKOS and Topic Maps can create very complex representations. Although this complexity allows for many things to be accomplished it can be overwhelmed for the end user. Biblio::Thesaurus allows a much simpler representation, of course it does not allow for some things to be accomplished so easily but it is much more simple to get started with.

- Converting between different representations is doable. There are some approaches for solving this problem, and there is a clear effort in research within this family of problems. Conversion may not be easy because of the complexity notation for most of the syntax.

- We felt there was a lack of solutions for manipulating technologies. There are a few software around, with interesting and intuitive GUIs,[1]. These solutions are powerful, but they all share the same problem, they lack expressiveness. And, most of the times, they lack a certain Independence of human intervention. Some operations could be easily completely automatic. Even more, some operations themselves should be automatically constructed from other sources. This king of automatizing it hard with visual clients, we need something that works on a lower level.

During the entire work we were in close contact with this technologies and we were able to take some more conclusions:

- The use of structures from the ontology family tree is growing. The requirements for today's state of the art solutions requires the use of structured information. This fact is most of the times stated by persons with any computer science skill.

- Many people today using ontologies do not have any computer science skills. Linguists are a good example of this set of people. Biologists are another community were the use of this technologies is growing.

- Ontologies can contribute for a unified set of communication services. Ontologies can take information share between different systems, their strong structured nature, and flexibility at the same time allow for complex data exchange.

We were very pleased with the final result and are hoping to adopt this approach in some real case scenarios in a near future. We are also hoping to have contributed to increase to the use of ontologies in the future. There is no doubt that this knowledge representation approach will play an important part in the journey we are taking into the semantic world of the web2.0.

[1]Graphic User Interface

6.2 Future Work

Some tasks that can be done in the future to improve this work:

- Use more case studies to further test the domain specific language. Mainly to verify that the current state of the language allows to describe the patterns required to solve problems.
- In case we find patterns that can not be described with the current language, add the operators necessary to write these patterns. If necessary, extend the language grammar.
- Finish the implementation of the unary operator (`not`) which is still not mature, and needs some more testing. Specially in some edge cases.
- Also, it may be possible that the operations defined in the callback table, and that are used to execute actions, are not enough. Different sets of problems may require different operations. Maybe design a couple of domain sets of callback tables. So, if you are working in some specific area, you should be using one of these specific callback tables.
- Take performance into consideration. Do some tests with really big ontologoies, and very complicated expressions, in order to measure performance. Improve code in order to improve performance.

Bibliography

[1] *Oxford English Dictionary, Second Edition.* 1989.

[2] José João Almeida and Alberto Simões. T_2O — recycling thesauri into a multilingual ontology. In *Fifth international conference on Language Resources and Evaluation, LREC 2006*, Genova, Italy, May 2006.

[3] D. Beckett and B. McBride. RDF/XML Syntax Specification (Revised). *W3C Recommendation*, 10, 2004.

[4] D. Conway. *Perl Best Practices.* O'Reilly Media, Inc., 2005.

[5] Francois Desarmenien. Parse::yapp. http://search.cpan.org/perldoc?Parse::Yapp.

[6] S. Dietzold. Generating RDF Models from LDAP Directories. In *Proceedings of the SFSW*, volume 5.

[7] M.J. Dominus and ScienceDirect (Online service). *Higher-order Perl: Transforming Programs with Programs.* Morgan Kaufmann Publishers, 2005.

[8] A. Farquhar, R. Fikes, and J. Rice. The Ontolingua Server: a tool for collaborative ontology construction. *International Journal of Human-Computers Studies*, 46(6):707–727, 1997.

[9] Ó. Ferrández Escamez, R. Izquierdo Beviá, S. Ferrández Escamez, V. González, and J. Luis. An user-centred ontology-and entailment-based Question Answering system. 2008.

[10] T.R. Gruber. A translation approach to portable ontology specifications. *KNOWLEDGE ACQUISITION*, 5:199–199, 1993.

[11] N. Guarino and Istituto (Roma) Consiglio nazionale delle ricerc. *Formal ontology in information systems.* IOS Press, 1998.

[12] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. A Practical Guide To Building OWL Ontologies Using The Protege-OWL Plugin and CO-ODE Tools Edition 1.0. *University Of Manchester*, 2004.

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

[14] D. Jin. *Ontological Adaptive Integration OF Reverse Engineering Tools.* PhD thesis, Queen's University, 2004.

[15] J. Lacasta, J. Nogueras-Iso, F.J. Lopez-Pellicer, P.R. Muro-Medrano, and F.J. Zarazaga-Soria. ThManager: An Open Source Tool for Creating and Visualizing SKOS. *INFORMATION TECHNOLOGY AND LIBRARIES*, 26(3):39, 2007.

[16] D.B. Lenat. CYC: a large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[17] A. Maratheftis and N.I.T. Consulting. Knowledge Management.

[18] J. McCarthy, Computer Science Dept, Stanford University, and Stanford Artificial Intelligence Laboratory. Circumscription-A Form of Non-Monotonic Reasoning. 1980.

[19] D.L. McGuinness, F. van Harmelen, et al. OWL Web Ontology Language Overview. *W3C Recommendation*, 10:2004–03, 2004.

[20] A. Miles, B. Matthews, D. Beckett, D. Brickley, M. Wilson, and N. Rogers. SKOS: A language to describe simple knowledge structures for the web. In *XTech 2005 Conference Proceedings*, 2005.

[21] A. Miles, B. Matthews, M. Wilson, and D. Brickley. SKOS Core: Simple Knowledge Organisation for the Web. In *Proceedings of the International Conference on Dublin Core and Metadata Applications*, pages 12–15, 2005.

[22] L. Miller, A. Seaborne, and A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 423–435, 2002.

[23] N.F. Noy and D.L. McGuinness. Ontology Development 101: AGuide to Creating Your First Ontology. *Disponible en http://www. Ksl. stanford. edu/people/dim/papers/ontology-tutorial-noy-mcguinessabstract. html [consulta: diciembre de 2005].*

[24] S. Pepper. The TAO of Topic Maps: Finding the Way in the Age of Infoglut. In *Proceedings of XML Europe 2000 Conférence.*

[25] A. Randal, D. Sugalski, and L. Tötsch. *Perl 6 and Parrot Essentials.* O'Reilly, 2004.

[26] Y. Shahar, S. Miksch, and P. Johnson. The Asgaard project: a task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence In Medicine*, 14(1-2):29–51, 1998.

[27] Alberto Manuel Simões and José João Almeida. Library::* — a toolkit for digital libraries. In *ElPub 2002 - Technology Interactions*, 2002.

[28] B. Smith, W. Kusnierczyk, D. Schober, and W. Ceusters. Towards a Reference Terminology for Ontology Research and Development in the Biomedical Domain. In *Proceedings of KR-MED*, pages 57–66, 2006.

[29] R. Stevens, C.A. Goble, and S. Bechhofer. Ontology-based knowledge representation for bioinformatics. *Briefings in Bioinformatics*, 1(4):398–414, 2000.

[30] M. Szymczak, M. Gawinecki, M. Vukmirovic, and M. Paprzycki. Ontological reusability in state-of-the-art semantic languages. *Proceedings of the XVIII Summer School of PIPS (to appear).*

[31] M. van Assem, V. Malaise, A. Miles, and G. Schreiber. A Method to Convert Thesauri to SKOS. *LECTURE NOTES IN COMPUTER SCIENCE*, 4011:95, 2006.

Appendix A

The Grammar

In this appendix we present the complete grammar for OML, the language specified in Chapter 2, in BNF notation.

```
pTree : statement_list

statement_list : statement_list statement DOT
               |

statement : cond_block ARROW action_block
          | action_list

cond_block : condition
           | condition oper cond_block
           | NOT OPEN cond_block CLOSE

condition : term relation term
      | TERM OPEN term CLOSE
      | REL OPEN relation CLOSE

term : STRING | VAR

relation : STRING | VAR

oper : AND | OR

action_block : action_list
```

```
action_list : action_list action
            |

action : ACTION OPEN condition CLOSE
       | SUB CODE
```

Appendix B

Simple Camila Notation

Simple CAMILA Notation

José João Dias de Almeida

November 27, 2008

1 Type Constructors

Most frequent notation for type construction:

$set(A)$	*set of A*
$A \rightharpoonup B$	*mapping*, A to B correspondence
A^*	*sequence of A*
$A \rightarrow B$	*function from A to B*
$A \times B$	*products*
$field1 : A \times field2 : B$	*product* with field names
$A + B$	*alternatives*
any	*universal type*
1	*singleton type*

A new type definition can include a predicate about its values in order to restrict the set holder — *invariant.* If we want to define a type *date*, even in it's most simplified version, it is easier to define a set holder (a three integer product, for example) and constrain the values with an invariant function that validates the triple values.

$$\begin{array}{lcrcl} date & = & day & : & int \ \times \\ & & month & : & int \ \times \\ & & year & : & int \end{array}$$

$$inv(d) \overset{\text{def}}{=} day(d) > 0 \wedge day(d) \leq 31 \wedge month(d) > 0 \wedge month(d) \leq 12 \wedge ...$$

Some functions associated with types:

Description	**Notation**
expression type	$type(e)$
expression compatible with type t	is-$t(e)$

2 Functions — $A \rightarrow B$

The CAMILA language allows several ways for functions definition. These definitions may include arguments, the result type, a pre condition definition or a state definition. Also, it is possible to define anonymous functions.

Description	**Notation**
compact function definition	$f(x) \stackrel{\text{def}}{=} y$
anonymous function definition	$\lambda(x) f(x)$
compact function with types signature	$\begin{array}{l} f : t1 \times t2 \longrightarrow t \\ f(x1, x2) \stackrel{\text{def}}{=} g(x1, x2) \end{array}$
function with pre-condition	$\begin{array}{l} f : t1 \times t2 \longrightarrow t \\ f(x1, x2) \stackrel{\text{def}}{=} \\ \quad \underline{pre} \quad p(x1, x2) \\ \quad \underline{in} \quad g(x1, x2) \end{array}$
display functions with state	

$$\begin{array}{l} f : t1 \times t2 \longrightarrow t \\ f(x1, x2) \stackrel{\text{def}}{=} \\ \quad \underline{post} \quad s' = h(x1, x2, s) \\ \quad \underline{in} \quad g(x1, x2) \end{array} \tag{1}$$

It is possible to define higher-order functions.

3 Generic expression constructors

The CAMILA language offers some of the usual mechanisms in specification languages, like *let* and conditional expressions (with partial pattern matching) and natural language *or*.

Description	**Notation**
conditional expression	$\left\{ \begin{array}{rcl} c1 & \Rightarrow & v1 \\ c2 & \Rightarrow & v2 \\ \underline{else} & \Rightarrow & vn \end{array} \right.$
conditional with pattern matching	$\left\{ \begin{array}{rcl} v1\ \mathit{is}\text{–}< h : t > & \Rightarrow & f(h, t) \\ v2\ \mathit{is}\text{–}<> & \Rightarrow & g \\ v3\ \mathit{is}\text{–}\{e : s\} & \Rightarrow & h(e, s) \\ v4\ \mathit{is}\text{–}\{\} & \Rightarrow & i \\ \underline{else} & \Rightarrow & j \end{array} \right.$
let expression	$\begin{array}{rl} \underline{let} & a = e1 \\ & b = e2 \\ \underline{in} & f(a, b) \end{array}$
let with pattern matching	$\begin{array}{rl} \underline{let} & <a, b> = e \\ \underline{in} & f(a, b) \end{array}$
natural language *or*	$a\ or\ b$

The `or` operator is not common in specification languages so it requires a little explanation. This operator (close to the natural language *or*) gives as result the first argument except if it is empty or undefined. It is used as an binary infix operator.

$$Or(exp1, exp2) \stackrel{\text{def}}{=}$$
$$\begin{cases} exp1 = \emptyset & \Rightarrow & exp2 \\ exp1 = () & \Rightarrow & exp2 \\ exp1 = <> & \Rightarrow & exp2 \\ undefined(exp1) & \Rightarrow & exp2 \\ \underline{else} & \Rightarrow & exp1 \end{cases}$$

4 Booleans

Booleans common logic functions and quantifiers exist in CAMILA.

Description	**Notation**
negation	$\neg a$
conjunction	$a \wedge b$
disjunction	$a \vee b$
implication	$a \Rightarrow b$
universal quantification	$\forall x \in setexp \wedge p(x)$
existential quantification	$\exists x \in setexp : p(x)$
unary existential quantification	$\exists^1 x \in setexp \wedge p(x)$

5 Mappings — $A \rightharpoonup B$

Uniform mappings can use the following predefined functions:

Description	**Notation**
mappings by enumeration	$\left(\begin{pmatrix} a1 \\ b1 \end{pmatrix} \begin{pmatrix} a2 \\ b2 \end{pmatrix}\right)$
mappings by comprehension	$\begin{pmatrix} f(a) \\ g(a) \end{pmatrix}_{a \in setexp}$
	$\begin{pmatrix} f(a) \\ g(a) \end{pmatrix}_{a \in setexp \wedge p(a)}$
domain	$dom(f)$
range	$rng(f)$
application	$f(x)$
domain constrain	$f \mid s$
domain subtraction	$f \setminus s$
rewrite, rewriting f with g	$f \dagger g$

6 Sequences — A^*

Sequences of type A can use the following base functions:

Description	**Notation**
sequences by enumeration	$<a1, a2, ... >$
sequences by comprehension	$<f(a) \mid a \in setexp>$
	$<f(a) \mid a \in setexp \wedge p(a)>$
head	$head(s)$
tail	$tail(s)$
element in position i	$s(i)$
first element	$\pi_1(x)$
second element	$\pi_2(x)$
soncatenation	$s \frown r$
append element	$<x> \frown s$
	$<x : s>$
	$s1 \frown s2 \frown ... \frown sn$
distributed concatenation	$\frown(<<... >, ... >)$
elements set	$\{x \mid x \in s\}$
	$elems(x)$
existing indexes	$inds(s)$
inverse	$reverse(s)$
length	$length(s)$
sorted sequence	$sort(s)$
custom sorted sequence	$sort2(f, s)$

Sequences can be heterogeneous (Example $S = \mathbf{any}^*$).

7 Sets — $set(A)$

Sets can use the following functions:

Description	**Notation**
sets by enumeration	$\{a1, a2, ... \}$
sets by comprehension	$\{f(a) \mid a \in setexp\}$
	$\{f(a) \mid a \in setexp \wedge p(a)\}$
non-deterministic choice	$choice(c)$
union	$c1 \cup c2$
interception	$c1 \cap c2$
sets difference	$c1 - c2$
belonging to a set	$e \in c$
not belonging to a set	$e \notin c$
number of elements	$\mathsf{card}\, c$
distributed union	$\bigcup(\{\{... \}, ... \})$
sorted set	$sort(s)$
custom sorted set	$sort2(f, s)$

Sets can be heterogeneous (Example $S = set(\mathbf{any})$).