Converting Ontologies into DSLs

João M. Sousa Fonseca[1], Maria João Varanda Pereira[2], and Pedro Rangel Henriques[1]

1 **Centro de Ciência e Tecnologia da Computação (CCTC)**
Departamento de Informática, Universidade do Minho
Braga, Portugal
{jprophet89,pedrorangelhenriques}@gmail.com

2 **Centro de Ciência e Tecnologia da Computação (CCTC)**
Departamento de Informática e Comunicações,
Instituto Politécnico de Bragança
Bragança, Portugal
mjoao@ipb.pt

Abstract

This paper presents a project whose main objective is to explore the Ontological-based development of Domain Specific Languages (DSL), more precisely, of their underlying Grammar.

After reviewing the basic concepts characterizing Ontologies and Domain-Specific Languages, we introduce a tool, Onto2Gra, that takes profit of the knowledge described by the ontology and automatically generates a grammar for a DSL that allows to discourse about the domain described by that ontology.

This approach represents a rigorous method to create, in a secure and effective way, a grammar for a new specialized language restricted to a concrete domain. The usual process of creating a grammar from the scratch is, as every creative action, difficult, slow and error prone; so this proposal is, from a Grammar Engineering point of view, of uttermost importance.

After the grammar generation phase, the Grammar Engineer can manipulate it to add syntactic sugar to improve the final language quality or even to add semantic actions.

The Onto2Gra project is composed of three engines. The main one is OWL2DSL, the component that converts an OWL ontology into an attribute grammar. The two additional modules are Onto2OWL, converts ontologies written in OntoDL (a light-weight DSL to describe ontologies) into standard OWL, and DDesc2OWL, converts domain instances written in the DSL generated by OWL2DSL into the initial OWL ontology.

1998 ACM Subject Classification D.3.4 Processors

Keywords and phrases ontology, OWL, RDF, languages, DSL, grammar

Digital Object Identifier 10.4230/OASIcs.SLATE.2014.85

1 Introduction

The use of domain-specific languages (DSL) enables a quick interaction with different domains, thereby taking a greater impact on productivity because there is no need for special or deep programming skills to use that language [3]. However, to create a domain-specific languages is a thankless task, which requires the participation of language engineers, which are (usually) not experts in the domain for which the language is targeted [4].

The work hereby presented takes advantage of the processable nature of OWL[1] ontologies

[1] Web Ontology Language as defined at http://www.w3.org/TR/owl-features/

3rd Symposium on Languages, Applications and Technologies (SLATE'14).
Editors: Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões; pp. 85–92

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

to generate DSLs from the enclosed domain knowledge. This is expected to automatize, at a certain extent, the language engineer task of bringing program and problem domains together.

Ontologies are usually created as only a scheme of the domain knowledge. But this is far from being a complete ontology. Ontologies also support instances of the concepts and their relations, but populating such *database* is a tremendous manual and time consuming routine.

An additional outcome of the proposed work is the possibility of populating ontologies from text files written in the new and automatically generated DSL. This would combine the best of both worlds. In practice, it is desired to take advantage of modelling the domain as an ontology from where a DSL (and its processor) can be extracted. The DSL processor can be specialized to convert its input into new OWL files containing the concept instances described in the input.

In this paper we will demonstrate that, given an abstract ontology describing a knowledge domain in terms of the concepts and the relations holding among them, it is possible to derive automatically a grammar, more precisely an attribute grammar, to define a DSL for that same domain.

The rest of the paper is organized as follows. Section 2 is used to discuss work by other authors similar to ours. Section 3 is devoted to an overview of Onto2Gra, discussing its functionality and introducing its architecture. The two modules already implemented, Onto2OWL and OWL2DSL, are presented in Section 4 and Section 5, respectively. The paper ends in Section 6 with some concluding remarks and guidelines for future work.

2 Related Work

In [2], a DSL is defined as a language designed to provide a notation tailored to one aplication domain and is based on the relevant concepts and features belonging to that domain. By providing notations tailored to the application domain, a DSL offers substancial gains in productivity and turns easier the task of the end-user. The main disadvantage of DSL is the cost of their development, requiring both domain and language development expertise.

Tairas et al. in [4], the authors explore the use of ontologies to perform domain analysis. A domain model is created by defining the scope of the domain, the terminology, concepts description and features model describing commonalities and variabilities. An ontology can be used to define the domain model. The information contained in the ontology will influence the language shape contributing for the early development stages. With this approach there is no need to start from scratch the DSL development. So, this work uses ontologies to validate the domain analysis and to get the domain terminology for the DSL creation.

Ceh et al. presented in [1] a concrete tool for ontology-based domain analysis and its incorporation on the early design phase of the DSL development. In this work, the authors identified several phases that a DSL development need. The most important are the decision, domain analysis, design, implementation and deployment.

The domain analysis is a process that uses several methodologies that differ on the level of formality and on the information extraction approach. The objective is to select and define the domain focus and collect the important information and integrate within a domain model. However, little attention is being paid to the analysis and design phase comparing with the efforts done in the implementation phase for instance. The methodologies have proven to be too complex (too much work) and they do not provide guidelines about how to use that information in the design phase. Subsequently the domain analysis is often performed informally.

The main idea of this work is to use ontologies to define the domain model. OWL classes, which define basic concepts, may be organized into a hierarchy. OWL defines two kinds of classes: simple named and predefined (the "Thing" and "Nothing"). The second component, the properties, is a binary relation, which associates values with individuals. The two main kinds of properties in OWL are object properties and datatype properties. Object properties relate objects to other objects. Datatype properties relate objects to datatype values. The third component, the individuals, are members of the user-defined ontology classes.

The authors of [1] also created a framework, called Ontology2OWL, to enable the automatic generation of a grammar from a target ontology. This framework accepts OWL files as input and parses them in order to generate and fill internal data structures. Then following transformation patterns, execution rules are applied over those data structures. The result is a grammar, acquired automatically, that is inspected for a DSL engineer in order to verify and find any irregularities. The engineer can either correct the constructed language grammar or change the transformation pattern or the source ontology. If the change was done on the ontology or transformation pattern, then a new transformation run is required. The framework can then use the old transformation pattern on the new ontology, the new transformation pattern on the old ontology, or the new transformation pattern on the new ontology. The DSL engineer can edit the source ontology with the use of a preexisting tool, such as Protégé. The final grammar can later be used for the development of DSL tools that are developed with the use of language development tools (e.g., LISA, VisualLISA).

This framework has the same objective as the one presented in this paper but some features were improved: reduction of the generation process steps, reduction of the user dependency, validation of the resulting grammar, transformation of the developed DSL to a form that is compatible with compiler generators, and generation of semantic actions associated with the grammar.

3 Onto2Gra: General Overview and Architecture

The purpose of Onto2Gra project here reported is to create automatically a Domain Specific Language based on an ontological description of that domain.

Figure 1 – the block diagram that depicts the architecture of Onto2Gra system – represents how, given an abstract ontology describing a knowledge domain in terms of its concepts and the relations among them, it is possible to derive automatically a grammar to define a new DSL for that same domain.

In a first stage the objective was to create a tool that allows to upload into the Protégé System an ontology in an acceptable OWL format. The original ontology shall be described in a special purpose DSL, a kind of natural language specifically tailored for that purpose, called OntoDL – Ontology light-weight Description Language. This tool was named Onto2OWL and what it does is to take an OntoDL file, that is an ontology specification file, and to convert it into OWL, that is the standard format for ontology descriptions. The aim of this tool is to offer an easy way to build a knowledge base to support the next phase. However, it is important to notice that this phase is not mandatory – this step can be skipped if the source ontology is already available in OWL format (or even in RDF/XML format). In that case, the user of Onto2Gra system can go directly to the second stage.

The second block is the most important on this proposal, and also the core of Onto2Gra. It is composed of a tool, OWL2DSL that makes the conversion of an OWL file into a grammar. This grammar is created systematically from a set of rules that will be explained in Section 5. From the OWL ontology description, OWL2DSL is able to infer: the non-terminal and

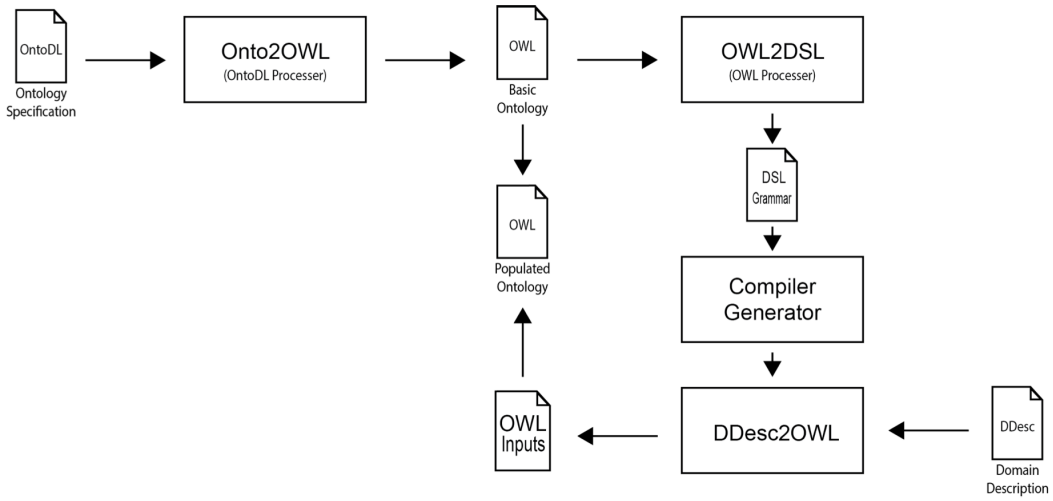

Figure 1 Onto2Gra Architecture.

terminal symbols; the grammar production rules; the symbol attributes and their evaluation rules[2]. Besides that, OWL2DSL generates a set of Java classes that are necessary to create an Internal Representation of the concrete ontology[3] extracted from each sentence of the target language[4]

The Grammar generated by OWL2DSL is written in such a format that can be compiled by a Compiler Generator (specifically in our case we are using the AnTLR compiler generator) in order to immediately create a processor for the sentences of the new DSL. AnTLR builds a Java program to process the target language; we call that processor DDesc2OWL and it is precisely the engine in the center of the third block.

Finally in the last block of the Onto2Gra architecture, the above referred tool DDesc2OWL, will read an input file, with a concrete description of the Domain specified by the initial ontology, and will generate an OWL file that, when merged into the original OWL file, will originate a specification that populates the original ontology creating a network of knowledge.

4 Onto2OWL Module

Onto2OWL is the first module of Onto2Gram system. The objective is to offer the possibility creating a specification file in the standard notation for ontologies specification that is OWL/XML from a file with a different and simple ontology specification language.

This module is compose of two parts. The first is a parser for the input files written in a DSL, called OntoDL, we have specially designed to describe ontologies. The second part is a transalator (a Java class) that manipulates the information gathered by the parser in order to generate the OWL file. In the next subsections it will be explained how these two parts work together.

[2] In the future we will also be able to derive the contextual conditions.
[3] An ontology with instances.
[4] The new Domain Specific Language defined by the generated Grammar.

4.1 The Parser for OntoDL Files

This parser is generated from a grammar that was created to specify OntoDL language. It recognizes all the basic components of an ontology described in OntoDL language.

After analyzing the problem, we found that only four parts are essential for a basic description of an ontology: Concepts, Hierarchies, Relations and Links. This supported the definition of OntoDL language as schematized below.

Listing 1 Template for a OntoDL File.

```
Ontology{
        Concepts[List of concepts]
        (,Hierarchies[List of hierarchies])?
        (,Relations[List of relations])?
        (,Links[List of links])?
}
```

An ontology is a specification of a certain domain. In order to describe the domain objects the ontology uses Concepts, or Classes. A Concept has a name, a description and a list of attributes. An attribute has a name and a type that can be a 'string', 'int', 'boolean' or 'float'.

After the Concepts specification, it is possible to define the hierarchy between two Concepts, the first is the father Concept(the super-class) and second is the son Concept(the sub-class). If one of the Concepts is not previously specified, the program ignores the Hierarchy that is being specified and issues a warning message.

After defining the hierarchical relations holding among Concepts, it is necessary to define the non-hierarchical Relations that will be used to connect Concepts. A Relation is a bridge between Concepts and it brings semantic value to the domain graph. With that in mind it was added a production rule to describe a Relation.

At last, we need to define the Links to identify the Concepts and the Relation that connect them. A Link is composed of two Concepts and one Relation. If one of the three items is not previously specified, the Link is ignored and a warning is displayed on the console. Listing 2 is an example of an OntoDL file to describe a small ontology called BookIndex (we will use this as a running example along the paper).

Listing 2 OntoDL File for the BookIndex example.

```
Ontology{
        Concepts[{Book},{Page},{Title},{SpecialTerm}]
        Hierarchies[]
        Relations[{has},{contains}]
        Links[{Book has Page},{Book has Title}, {Page contains
            SpecialTerm}]
}
```

From OntoDL grammar it is possible to generate a parser to process OntoDL sentences (ontology descriptions). This parser will store the information extracted from the input file in a set of java classes that was designed to accommodate the needs of the translator, as explained in the next subsection.

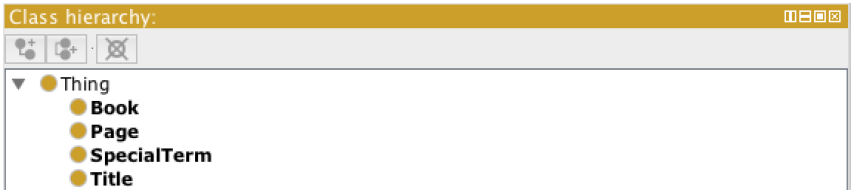

Figure 2 The generated OWL BookIndex ontology opened in Protégé.

4.2 The OWL File Generator

The OWL file generation has the objective of taking the information that was retrieved by the parser and stored in the java classes in order to generate the final product that is an OWL/XML file.

The parser returns an object of the class "Ontologies". Listing 3 shows how this class is organized.

Listing 3 Ontologies Class.

```
public class Ontologies{
    public ArrayList<Concepts> concept;
    public ArrayList<Hierarchies> hierarchy;
    public ArrayList<Relations> relation;
    public ArrayList<Triples> triple;
    public void gerarowl(String input){...}
}
```

As the listing above shows, the object that is returned by the parser already has all the information required to generate the OWL file. This information is stored in four important Array List. The first is the array of Concepts. This array stores all the Concepts that are processed by the parser. The second list stores the Hierarchies between concepts. This object is very simple, it only saves the name of the super-class and of the sub-class. The next ArrayList is also important because it stores all the non-hierarchical connections between Concepts. If a relation is not present in this list its name can not be used to create Links. As was referred above, the final array is the one that stores the Links or Triples. The links state what Concepts are connected and with what Relations.

To generate the final OWL/XML file it is enough to run the method with the name "gerarowl(String name)" that belongs to the Ontologies class. This method receives a string parameter. This parameter specifies the name of the OWL file; normally this name is the same as the OntoDL file, sent as input to the parser.

After the generation of the OWL/XML file we can work more on the ontology, to explore or to edit it. For that purpose we can load it into any tool that supports OWL/XML, like Protégé[5]. Then it is also possible to add information to enrich the ontology. Figure 2 represents the OWL version of the BookIndex ontology produced from the OntoDL description by the Onto2OWL module.

Concluding, this module was created with the objective of generating OWL files from simple descriptions of the domain. It can be used separately and independent of the other components, but it was important to generate input files for the second module of this project, OWL2DSL.

[5] `http://protege.stanford.edu/`

5 OWL2DSL Module

This section introduces the OWL2DSL module, that is presented in Figure 1 , and explains how it is possible to generate a grammar specification and generate a parser for a DSL to describe elements for the domain.

This module combines two phases the OWL or RDF parser and the CodeGenerator. This first phase, is a simple OWL or RDF parser that retrieves a Ontology Object (OO).This OO is crucial because it will enable the creation of the desired grammar.The CodeGenerator receives as input this OO and produces: the grammar and a set of Java classes necessary to implement the next module DDesc2OWL. The output grammar is composed of simple productions that obey a set of transformation rules. Those rules, look for OWL patterns to transform them systematically into grammar elements (symbols, and productions).

To start the generation process, we create a production with the axiom, `Thing` and we insert as many alternatives as the number of Concepts that are hierarchically connected to `Thing`. The listing below shows the production diagram generated by the application of that first transformation rule.

Listing 4 Thing Production.

```
thing: 'Thing['(Tproduction1|Tproductioni|TproductionN)(','(
    Tproduction1|Tproductioni|TproductionN))*']' ;
```

After the creation of this main production, all the Concepts that appeared on its RHS are iterated aiming at creating all the subsequent productions for those concepts. This task is accomplished by a recursive function `sub-production`. The function `sub-production` generates almost all the complementary Java classes that add dynamic semantics to the generated Grammar. As this `sub-production` function is recursive, it explores all the sub-Concepts and returns all the productions to the CodeGenerator Class.

The productions corresponding to the concepts also are generated according to a set of systematic transformation rules. These productions are composed of four parts. The first part is an ID that will be used to identify each instance of that concept. The attributes are specified using a non-terminal symbol preceded by a keyword (attribute name). Then, for each attribute, one new production is added to the grammar in order to return its value. After the list of attributes we have the third part, a set of sub-productions. Each sub-production contains on the RHS a sub-concept that is connected to the main Concept by a *is-a* relation. The last part of the production is used to specify the non-hierarchical relationships for the non-hierarchical relationships that connect these Concepts with other concepts.

The listing below shows some of the productions of the grammar generated from the BookIndex ontology displayed in Figure 2.

Listing 5 Example of a Generated Grammar: BookIndex.

```
thing: 'Thing['(book | page | title | specialterm) (','(book | page |
     title | specialterm))* ']';
book:'Book{' bookID(','''['(book_has)(','(book_has))*']')?'}';
bookID: STRING;
book_has:'{''has' STRING'}'; //Book has Page or Title use the STRING
    to link them
```

Summing up, it was demonstrated that a complete grammar for the new DSL can be generated from the ontology that describes that domain applying a small set of transformation rules. The upcoming grammar is completely human readable and can be adapted to any

special purpose. Adding to the grammar the Java Classes, also generated systematically and explained above, it is possible to obtain a processor to cope with the sentences of the new DSL. Listing 6 is a short example of a sentence written in the language BookIndex defined by the generated grammar shown in the previous listing.

Listing 6 Example of a DDesc input file.

```
Thing{
        Book{ "Ref_002" ,[{has "Game of Thrones"},{has "Page1"}, {has
             "Page2"}]},//the reasoner will infer the types on the
            has relations
        Page{ "Page1"},
        Page{ "Page2"},
        Title{ "Game of Thrones"}
}
```

6 Conclusion

As it was presented in this paper, the ontologies are a very useful formalism to specify domain models. Moreover a direct translation to a grammar avoids manual and informal methodologies in the design phase of a new language. This work is focused on the offline properties that an ontology can offer and define a set of transformation rules to convert the ontological information into an AntLR grammar. It was possible to automatize this process in such a way that the user has no need to interfere. Besides the grammar, associated attribute evaluation is also generated in order to fill internal data structures that can be used in a further step. At the end, it will be possible to describe a specific domain using the new created DSL and to use the information extracted from each description (written in that DLS) to populate the initial ontology.

Acknowledgements. We are indebted to Marjan Mernik and his team for the preliminary discussion on this topic. This work is funded by National Funds through the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project PEst-OE/EEI/UI0752/2014.

References

1 I. Čeh, M. Črepinšek, T. Kosar, and M Mernik. Ontology driven development of domain-specific languages. *Comput. Sci. Inf. Syst.*, 8(2):317–342, 2011.

2 Tomaz Kosar, Pablo Martinez Lopez, Pablo A. Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Inf. Softw. Technol.*, 50(5):390–405, April 2008.

3 Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.

4 Robert Tairas, Marjan Mernik, and Jeff Gray. Using ontologies in the domain analysis of domain-specific languages. In Michel R. Chaudron, editor, *Models in Software Engineering*, pages 332–342. Springer-Verlag, Berlin, Heidelberg, 2009.