Computer Languages, Systems & Structures 45 (2016) 35–52

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Ontological approach for DSL development

Maria João Varanda Pereira [b,*], João Fonseca [a], Pedro Rangel Henriques [a]

[a] Centro Algoritmi (CAlg-CTC), Dep. Informática, Universidade do Minho, Portugal
[b] Centro Algoritmi (CAlg-CTC), Dep. Informática e Comunicações Instituto Politécnico de Bragança Portugal

ARTICLE INFO

Article history:
Received 22 August 2015
Accepted 30 December 2015
Available online 22 January 2016

Keywords:
DSL
Ontologies
Grammars
Problem domain concepts

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 DSLs, 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 specific semantic actions. The Onto2Gra project is composed of three engines. The main one is OWL2DSL, the component that converts an OWL ontology into a complete Attribute Grammar for the construction of an internal representation of all the input data. The two additional modules are Onto2OWL, converts ontologies written in OntoDL into standard OWL, and DDesc2OWL, converts domain instances written in the new DSL into the initial OWL ontology.

Published by Elsevier Ltd.

1. Introduction

The use of Domain-Specific Languages (DSL) [16,27] 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 [18,20]. However, to create a DSL is a thankless task that requires the participation of language engineers, which are (usually) not experts in the domain for which the language is targeted. It is difficult to map the problem domain concepts into the language syntax.

The work hereby presented takes advantage of the processable nature of OWL [6,17] ontologies [8,26,30] to generate DSLs from the enclosed domain knowledge. This is expected to automatize, at a certain extent, the language-engineer's task of bringing program and problem domains together. 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.

* Corresponding author.
E-mail addresses: mjoao@ipb.pt (M.J.V. Pereira), jprophet89@gmail.com (J. Fonseca), pedrorangelhenriques@gmail.com (P.R. Henriques).

http://dx.doi.org/10.1016/j.cl.2015.12.004
1477-8424/Published by Elsevier Ltd.

DSL is a language designed to provide a notation based on the relevant concepts of an application domain. The objective is to improve the productivity and to allow the end-user to construct easily correct programs, but it implies some development costs. Several DSL implementation patterns are presented in a study by Kosar et al. [15], for instance:

- Preprocessing – the new DSL constructions are transformed into a language already implemented.
- Embedding – use of a domain specific operations library in the host language; the domain idiom is expressed using syntactic mechanisms of the host language.
- Compiler/interpreter – standard language processing techniques are used to implement the DSL from the scratch.
- Commercial Off-the-Shelf – use of existing tools that are applied to a specific domain (XML-based DSLs) to implement the new language constructions.

The downfall of these approaches is that they are complex because they require that the programmer have some expertize on the application domain, also on grammar engineering, and language implementation.

In this paper we propose an approach: to derive the grammar for the DSL from a formal definition of the domain based on an ontology.

That approach is useful for domain analysis phase where language engineer needs to identify concepts in a domain and relations among these concepts. When these are identified a grammar needs to be designed to reflect these concepts and relations. In this step concepts are usually mapped to non-terminals and relations are covered by right-hand side of grammar productions. Hence, the proposed approach automate this part of language development, which is usually done manually.

On one hand, a domain expert gets a formal language to write statements describing its expertize. On the other hand, a language engineer gets a grammar that properly defines a complete language about a domain that is not his expertize.

As said, the derivation process is rigorous and systematic so it can be automatized. A benefit is the efficiency, simplicity, of the development process; a drawback of the proposal may be the quality of the derived language, it can require a manual improvement a-posteriori.

Actually we produce a complete Attribute Grammar (AG) [14] with attributes and attribute evaluation rules for the extraction of all the data from the input sentences and the construction of a complete internal representation of that input data. The AG is written in the syntax of AnTLR [22] so a parser and a processor can be immediately generated from the output of our tool. In that sense, our approach contributes for the design and implementation of a DSL following the third implementation pattern above. In the present version of our system, the user only has to write his ontology, pass it to the generator, and then compile the output with AnTLR to get a processor for the new DSL programs; more semantic rules can be manually added to the generated AG to get the final desired behavior that is not specified in the given ontology.

The rest of the paper is organized as follows. Section 2 is used to briefly present other works related with DSL development. In Section 3 the formalisms involved in this proposal are described: Section 3.1 introduces the formal definition of ontology and other related features; and Section 3.2 discusses grammars as a formalism to specify and implement languages. In Section 4 a set of translation rules are presented. These rules are used to map the ontology to grammar productions, and attributes. Section 5 is devoted to an overview of Onto2Gra, discussing its functionality and introducing its architecture; the complementary modules, Onto2OWL and DDesc2OWL—providing extra features to process the input ontology and the DSL generated—are briefly presented. The main module OWL2DSL is described in Section 6. In Section 7 the overall project evaluation is discussed, and then we focus on the DSL quality assessment. The paper ends in Section 8 with some concluding remarks and guidelines for future work.

2. Related work

Lots of work has been done in the last years regarding Domain Specific Languages development [13,32]. The initial phase of DSL development is the definition of the domain, its terminology, the concepts and their interdependencies, the so called *Domain Analysis Phase*. For the definition of these artifacts models should be used. Using ontologies in this phase there is no need to start from scratch the DSL development.

In this direction, the work presented in [29] uses ontologies to support the domain analysis and to get the domain terminology for the DSL creation. Ontologies help in the initial phase of domain understanding and can be combined with other formal domain analysis methods during the development of a DSL. These authors propose a UML-based approach to convert ontologies into class diagrams. The authors also suggest that class diagrams can be transformed into a grammar. However just manual transformation rules are proposed for both processes.

The authors in [33,34] also emphasize the advantages of using ontologies to support the development of DSLs. For that purpose, the authors developed a framework, OntoDSL. It is a development framework for DSLs that uses ontologies during the design phase. The developer constructs a visual program, step by step, guided by an ontology that defines all the possible combinations between elements, relations and instances. The definition of such ontology is made by a domain expert in order to guide the DSL programmer. The consistency of the program is checked, concepts are suggested, incomplete parts and redundancies are detected. The DSL designer counts on an expressive language (based on Java) that allows for modeling logical constraints over DSL metamodels.

Although very similar to our work, the main difference is that Walter's framework is integrated in a model-driven approach to software development [5] and they followed the OMG's four layered architecture. They are mainly concerned with the development and exploitation of formal domain-specific models. In our case, included in a grammar engineering context, the DSL grammar is automatically generated from the ontology and the generated program templates already guide a DSL user; the errors will be detected by the DSL processor (implemented in AntLR using the generated grammar). Ceh et al. presented in [2] 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 authors also created a framework, called Ontology2DSL, 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 on those data structures. The result is a grammar that is inspected by 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 on the 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.[1] The final grammar can later be used for the development of DSL tools that are developed with the use of language development tools (i.e. LISA [19], VisualLISA [21], etc.).

Our proposal has the same objectives as those of Ontology2DSL, however we did some improvements: small number of steps along the generation process; reduction of the user dependency; validation of the resulting grammar; use of an output format for the new grammar that is compatible with commonly used compiler generators; generation of semantic actions associated with the context free grammar in order to process the new DSL sentences (programs) and generation of a DSL program template. We want to emphasize that the system automatic generates not only a context free grammar but also an attribute grammar because semantic actions are added to each production to collect input data.

3. Formalisms involved

This work is based on well-known formalisms, like ontologies and grammars. These formalisms will be briefly revisited in the following sections aiming at settle down terminology (naming) and notations used in the rest of the article.

3.1. Ontologies – knowledge formal representation

An ontology is a formalism used to specify a knowledge domain by describing objects and categorize them. It can be defined as *a formal naming and definition of the types, properties, and interrelationships of the entities that really or fundamentally exist for a particular domain of discourse*. The basic elements are a set of concepts, a set of relations and a set of triples (concept, relation, concept). Each concept can have attributes, or can have instances. Relations can be split into two categories.

The definition of an ontology [25] also includes axioms of the knowledge domain. In this way it is possibility to reason over that domain, i.e., to check the consistency of the relations or infer new knowledge.

So, an ontology, *Onto*, can be defined as a tuple

$$Onto = < Con, Rel, Prop, Ax >$$

where

Con is the set of all Concepts, including classes (general concepts) or class instances (individuals);
Rel is the set of all Relations, both taxonomic (or hierarchical) and non-taxonomic. Usually the hierarchical relation is denoted by the name 'is _ a'. Each relation $r \in R$ defines a set of tuples of the form $<$ subject, predicate, object $>$, where subject and object are concepts and predicate is the relation r;
Prop is the set of all Properties that characterize each concept.
Ax is the set of all axioms and inference rules. Axioms are defined as predicates over the relations or the concept properties. Inference rules are defined as logical formulae that allow to deduce the consequent when the antecedent holds, this is, that combine known facts to produce new facts.

There are many formal notations (languages) to describe ontologies but one of the most used is Ontology Web Language (OWL), specially in the context of the Semantic Web. So our choice for the project here reported was obviously OWL; actually our grammar generator accepts as input ontologies written in OWL dialect.

[1] A free, open-source platform, that provides a suite of tools to create and maintain knowledge based applications based on ontologies, and available from Stanford University from ⟨http://protege.stanford.edu/products.php⟩.

There are also some specific environments to edit and process ontologies. The most used is Protégé. In Protégé concepts are called classes, instances are individuals, the relations are denoted as object-properties (relate objects to other objects), and concept properties are denoted as data-properties (relate objects to datatype values). Relations [11] can be functional or inverse functional, transitive, symmetric or asymmetric, reflexive or irreflexive. Data-properties can be associated to a concept or just to an instance of that concept. A concept can have a set of pre-defined instances that can be exclusive or not. It is also possible to set up restrictions: quantifiers (*existential* or *universal*), cardinality constraints (*max*, *min* or *exact*), and hasValue (to restrict an attribute to a specific value). Protégé was also used in our work.

3.2. Grammars – language formal representation

An attribute grammar is a formalism to specify rigorously the syntactic and the semantic rules of a programming language. The programmer that needs to write a sentence in that language must follow the grammar rules to create a valid sentence, and the compiler must also follow the same grammar to interpret it, this is, to extract the sentence's meaning.

A Context Free Grammar (CFG) is a formalism used to specify the syntax of a (computer) language. It describes the language elements or vocabulary, and how to relate them in order to build correct (or valid) sentences [3,10]. Formally a CFG is a tuple

$$CFG = < T, N, S, P >$$

composed of a set of terminal symbols, T, a set of nonterminal symbols, N, a start symbol (axiom), S, and a set of derivation rules (or productions), P. Each $p \in P$ has the form $LHS \rightarrow RHS$, where the left-hand side, LHS, is one non-terminal symbol, and the right-hand side, RHS, is a sequence of terminal and non-terminal symbols (notice that the sequence can be empty).

A terminal symbol is a token (a sequence of characters built according to lexical, or orthographic rules) and a non-terminal symbol is a concept denoted by an identifier.[2] A sequence of terminal symbols is a *valid sentence* of the language defined by a given CFG if and only if that sequence derives from the axiom applying the grammar derivation rules.

An Attribute Grammar (AG) is an extension of a CFG that allows to define also formally the language semantics [7]. An AG is a tuple

$$AG = < CFG, A, CR, CC, T\,R >$$

composed of a context free grammar, CFG, a set of attributes (associated to the terminal and nonterminal symbols), A, a set of computation rules that define how to evaluate the attribute values, CR, a set of contextual conditions that define the static semantics, CC, and a set of transformation rules, $T\,R$.

Attribute grammars not only provide a formal specification of a language but also allows to construct processors for that language. Language Processing is the analysis and translation (or transformation) of sentences of a given language.

4. Mapping ontologies to grammars

In order to convert an ontology into a grammar, a set of translation rules must be followed. So, this section defines the core of our work: the mapping of concepts into grammar symbols, and the mapping of relations into grammar productions; the generation of attributes and their evaluation rules are also described, in the last section. The rules used in the proposed generative process will be introduce by examples, and summarized in a table.

4.1. CFG generation

In order to specify the translation rules we consider the definitions presented in Section 3.1. Namely, the sets Rel and Prop that correspond to the hierarchical and non-hierarchical relations.

The first rule is concerned with the Thing concept and its hierarchical relations. The grammar derivation process always starts by this rule. Thing is a super production that connects all the concepts with no *father*, this is, concepts that are super classes. Considering A and B as the main concepts of the domain described by the ontology the first rule is the following:

```
A is_a Thing, B is_a Thing    ===>    Thing -> ( A | B )+
```

Adding some syntax sugar, the first rule was replaced by the following:

```
A is_a Thing, B is_a Thing    ===>    Thing -> ( A | B ) (,( A | B ))*
```

In a second rule other hierarchical relations can be defined:

```
B is_a A, C is_a A    ===>    A -> ( B | C )*
```

Concerning the non-hierarchical relations, a third rule must be defined. In that case the name of the relation that links 'A' to 'F', lets say 'rel', is used as a keyword.

```
A rel F    ===>    A -> ('rel' F)*
```

[2] Notice that this insight regarding the definition of Nonterminal is of uttermost importance for our proposal to deduce a CFG from an Ontology.

Table 1
Rules to generate the CFG.

Ontology	CFG
A is a Thing, B is a Thing	Thing→(A|B) (,(A|B))*
B is a A, C is a A	A→(B|C)*
A is related with F by the relation 'rel'	A→('rel' F)*
A has an attribute 'att' of type'att type'	A→('att''att type')?
A is related with min 1F	A→('rel' F)+
A is related with min 2F	A→('rel' F)('rel' F)+
A is related with max 1F	A→('rel' F)?
A is related with max 2F	A→ε | ('rel' F)('rel' F)?
A is related with min 1 max 1F	A→'rel' F
A is related with min 1 max 2F	A→('rel' F)('rel' F)?
A is related with min 2 max 2F	A→('rel' F)('rel' F)
A is a concept ∧ $ (A, 'rel', X)	A→'A'|'a'

When a concept has data properties, another translation rule must be used. Considering the property named 'att' of type 'ttt' associated with concept A, the property name and the property type will appear in the RHS of a production with A in the LHS. This fourth rule is:

```
A -> ('att' ttt)?
```

These rules can be combined and it is possible to translate concepts with data properties, hierarchical and non-hierarchical relations, in the same production. The RHS of the production should begin with a name that uniquely identifies the concept. Considering again a concept A, this identifier could be AId, and this is the only mandatory element on the RHS. The composition rule that will be used produces the following production:

```
A -> AId (',' ('att' ttt)? ('rel' F)* ( B | C )*)?
```

Still concerning the third rule it is possible to represent the cardinality of the non-hierarchical relations. The cardinality is given by two parameters presented on the TripleRange class (considering the OWL notation), 'min' and 'max'. These variables define how the relation will be translated into a grammar rule. The variants are simple to explain. Listing 1 shows the cardinalities that are recognized by our system and how those cardinalities are translated to the grammar. If there is a combination of minimum and maximum the grammar generated will represent that same cardinality in the productions.

```
A rel min 1 F          ===> A -> ('rel' F)+
A rel min 2 F          ===> A -> ('rel' F)('rel' F)+
A rel max 1 F          ===> A -> ('rel' F)?
A rel max 2 F          ===> A -> ε | ('rel' F)('rel' F)?
A rel min 1 max 1 F ===> A -> 'rel' F
A rel min 1 max 2 F ===> A -> ('rel' F)('rel' F)?
A rel min 2 max 2 F ===> A -> ('rel' F)('rel' F)
```

Listing 1. : Cardinality rules

A last rule, a kind of default rule (not exemplified), states that any concept that is not the origin of any relation (never appears as a Subject of a triple) derives into a Terminal.

Summing up a final set of rules, that are used to generate the CFG, is presented in Table 1. These rules can be combined; actually several translation rules are composed to generate one grammar production.

A simple example: The Book Index.

In Listing 2, using a notation called OntoDL created by us, we specify an ontology describing a simple and small domain concerned with the problem of creating a *book index*.[3] To automatize the construction of the index, we need to define the book title and a set of pages. Each page has an associated list of the terms that are defined in that page.

```
Ontology {
        Concepts[ {Book}, {Title}, {SpecialTerm},
              {Page, Attributes [{number INT}{text STRING}]} ]
        Hierarchies[]
        Relations[{has}, {contains}]
        Links[{Book has Page}, {Book has Title}, {Page contains SpecialTerm}]
}
```

Listing 2. : OntoDL File for the BookIndex example

In this first example there are no hierarchical relations. However there are two kind of non-hierarchical relations: has and contains. The pairs of concepts associated by each relation are defined in the Links block. Notice that the concept Page is characterized by two data properties (or attributes), number and text, respectively of types integer and string.

Applying the rules explained above it is possible to generate the grammar productions. For the sake of space only some of them are listed below:

[3] Considering the number of elements.

```
thing : 'Thing[' (book | page | title | specialterm) (',' (book | page | title |
    specialterm))* ']'
    ;
book : 'Book{' bookID (',' '[' (book has) (',' (book has))* ']')? '}'
    ;
bookID : STRING
    ;
book has : '{' 'has' STRING '}'
    ; //STRING represents a page or a title
page : 'Page{' pageID ('number' number)? ('text' text)? (',' '[' (
    page contains specialterm)* ']')? '}'
    ;
pageID : STRING
    ;
page contains specialterm : '{' 'contains' STRING '}'
    ; //STRING represents a special term
```

Listing 3. : Example of a Generated Grammar: BookIndex

An example of a program written in this new DSL is shown in the next listing:

```
Thing {
    Book{" Ref002", [{has "Game of Thrones"},{has "Page1"}, {has "Page2"}]},
    Page{" Page1" number 5 text "Once upon a time", [{contains "time"}]},
    Page{" Page2" number 7 text "The end of the story", [{contains "end"}{contains "
        story"}]}
}
```

Listing 4. : Example of a DSL input file

This program is simple to understand because it is written in a DSL using only terms that directly follow the concepts and relations involved in that problem domain.

The generated grammar can be adapted to reflect the syntactic preferences of the end user.

The rules applied to generate the semantic part are presented in the next section.

4.2. Attribute grammar generation

By default attributes and semantic evaluation rules will be automatically added to the CFG symbols and productions according to the following generation schema.

First of all symbols are associated with one attribute:

- Each terminal symbol has an intrinsic attribute text of type String.
- Each non-terminal symbol A has a synthesized attribute syn, that can be called value or struct depending on the RHS.

1. if the RHS contains only a terminal, then A will return an attribute value of type String evaluated according to the semantic rule below

   `A -> t ===> { A.value = t.text; }`
2. if the RHS contains only a non-terminal with an attribute value, then A will return an attribute value of type String evaluated according to the semantic rule below

   `A -> B ===> { A.value = B.value; }`
3. if the RHS contains more than one symbol with an attribute syn, then A will return an attribute struct of type ClassA (where ClassA is a java class generated according to the RHS to contain a value for each element on the right side of the production) evaluated according to the semantic rule below

   `A -> B...C ===> A.struct.b = B.syn;...; A.struct.c = C.syn;`

Revisiting The Book Index *example.*

To illustrate the semantic rules above, Listing 5 shows a fragment of the attribute grammar generated for the *Book Index* example discussed in the previous section. Notice that the package'bookindex.*' imported in the *header* contains the definitions of the java classes corresponding to each non-terminal symbol. Those classes will be used to type the attribute struct synthesized by the corresponding non-terminal.

```
grammar bookindex;

@option{

                                                    }

@header{
language=Java;
import java.util.ArrayList;
import bookindex.*;
}
@parser::members{
 public ArrayList<Instances> instances=new ArrayList();
}
thing returns[Thing struct, ArrayList<Instances> insta]
  @init{
   $thing.struct=new Thing();
   instances=new ArrayList();
   $thing.insta=new ArrayList();
  }
  @after{ $thing.insta.addAll(instances); }
    : 'Thing['
    ( p00=specialterm {$thing.struct.specialterms.add($p00.struct);}
        | p01=page {$thing.struct.pages.add($p01.struct);}
        | p02=book {$thing.struct.books.add($p02.struct);}
        | p03=title {$thing.struct.titles.add($p03.struct);}
    )+ ']'
;
book returns[Book struct]
  @init{
   $book.struct=new Book();
   Instances i=new Instances();i.
   type="Book";
  }
  @after{ instances.add(i); }
    : 'Book{'
    bookID { $book.struct.bookID=$bookID.value; i.name=$bookID.value.replaceAll(" ","_");
          }
        ( ',' '['
     (page1=book_has_page { $book.struct.page_has.add($page1.value);
           i.objectproperties.add(new Objectproperties($page1.value.replaceAll(" ","_")
                ,"has")); } )*
         (title2=book_has_title { $book.struct.title_has.add($title2.value);
            i.objectproperties.add(new Objectproperties($title2.value.replaceAll(" ","_
                ),"has")); } )+ ']
    )?'}'
;

bookID returns[String value]
    : STRING { $bookID.value=$STRING.text; }
;
book_has_page returns[String value]
    : '{' 'has' 'page' pageID { $book_has_page.value=$pageID.value; } '}'
;
book_has_title returns[String value]
    : '{' 'has' 'title' titleID { $book_has_title.value=$titleID.value; } '}'
;
page returns[Page struct]
  @init{
   $page.struct=new Page();
   Instances i=new Instances();i.
   type="Page";
  }
  @after{ instances.add(i); }
    : 'Page{'
    pageID { $page.struct.pageID=$pageID.value; i.name=$pageID.value.replaceAll(" ","_");
          }
        ('text' text { $page.struct.text=$text.value; i.dataproperties.add(new
            Dataproperties($text.value,"text")); })?
        ('number' number { $page.struct.number=$number.value; i.dataproperties.add(new
            Dataproperties($number.value,"number")); })?
        (',' '[' (specialterm1=page_contains_specialterm { $page.struct.
            specialterm_contains.add($specialterm1.value);
          i.objectproperties.add(new Objectproperties($specialterm1.value.replaceAll("
              ","_"),"contains"));})* ']
    )? '}'
;
pageID returns[String value]
    : STRING { $pageID.value=$STRING.text; }
;
page_contains_specialterm returns[String value]
    : '{' 'contains' 'specialterm' specialtermID { $page_contains_specialterm.value=$
        specialtermID.value; } '}'
;
```

Listing 5. : Example of a Generated Attribute Grammar: BookIndex

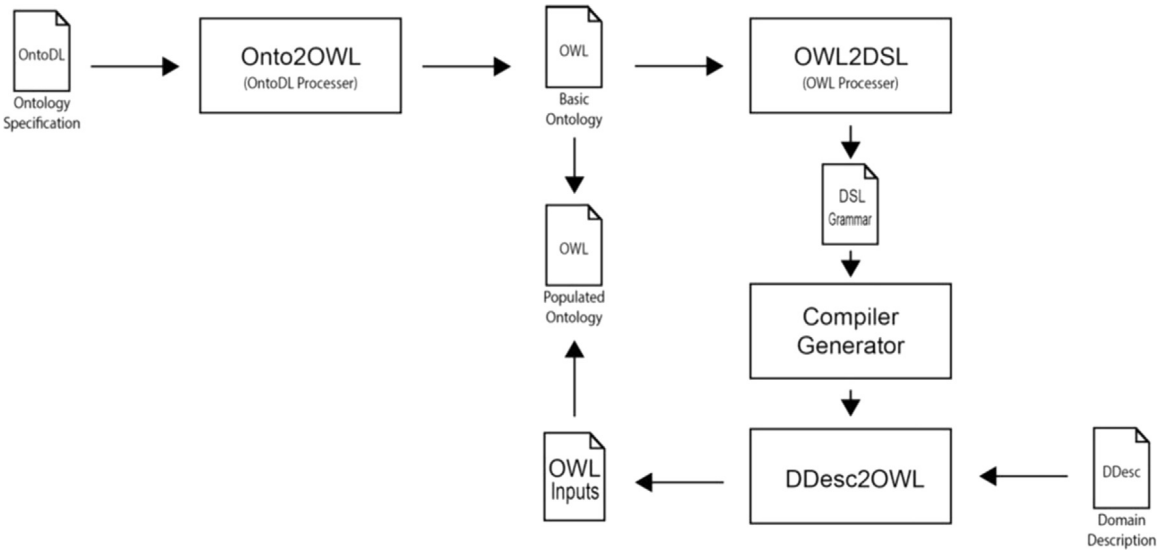

Fig. 1. Onto2Gra architecture.

5. Onto2Gra architecture

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

Fig. 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.

The original ontology to be processed by Onto2Gra can be provided in OWL, or it can be written in OntoDL, *Ontology light-weight Description Language*, a DSL specially tailored for that purpose following the definition in Section 3.1, designed with a natural language flavor. We sketch in Listing 6 the OntoDL program template, enhancing the main building blocks.

```
Ontology{ Concepts [List of concepts(name,desc,atts)]
          (,Hierarchies [List of pairs(concl,conc2)])?
          (,Relations [List of relations])?
          (,Links [Listof links(subj,pred,obj)])? }
```

Listing 6. : Template for a OntoDL File

As said previously, to describe the domain objects the ontology uses Concepts, or Classes. A Concept has a name, and optionally a textual 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, as can be seen in Listing 6, it is possible to define the Hierarchy between one or more pairs of concepts, the first concept is the super-class, and the second concept is the sub-class. If one of the concepts is not previously specified, the processor Onto2OWL ignores the Hierarchy that is being specified and issues a warning message. After defining the hierarchical relations holding among concepts, it is necessary to list the non-hierarchical Relations that will be used to connect concepts. At last, we need to define the Links to identify the ontology triples (the *subject* and *object* concepts and the *predicate* that connects them).

So, the first block in Onto2Gra architecture, Onto2OWL, is a tool to convert an OntoDL ontology into an acceptable OWL format that can be uploaded to Protégé System. After loaded into Prot´eg´e the generated OWL file can be explored or edited (more information can be added to the domain description without creating the specification from the scratch).

The aim of Onto2OWL 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, it is the core of Onto2Gra and it will be described in detail in the next section. It is composed of a tool, OWL2DSL that makes the conversion of an OWL file into a grammar. This grammar is created systematically using the set of rules explained in the previous section. From a given OWL ontology, OWL2DSL is able to infer: the non-terminal and terminal symbols; the grammar production rules; the synthesized attributes and the evaluation rules. These new attributes are implemented by Java classes defined according to the structure of the production rules (as seen above, a systematic derivation approach is applied) and they will be used to store the concrete data that instantiate the right hand side symbols. The execution of those evaluation rules will create an Internal Representation of the concrete ontology extracted from each DSL program.

The Attribute Grammar generated by OWL2DSL is written in such a format that can be compiled by a Compiler Generator (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 (a translator for Domain Descriptions into OWL) and it is precisely the engine in the center of the third block. As can be seen in Fig. 1, DDesc2OWL will read an input file, with a concrete description of the Domain specified by the initial ontology, and it will generate an OWL file that, when merged into the original OWL file, will populate the ontology.

Fig. 2 shows the interface of the third and last block.

To run the third module, it is necessary to specify three parameters, that are three files to be processed in different moments. The first parameter is the pathname for the original OWL ontology (that was used for the creation of the grammar by the OWL2DSL Module). This file will be processed again to rebuild the ontology in order to create the populated version.

The second parameter is the pathname for the AG file generated by OWL2DSL. This attribute grammar, complemented with the Java classes also generated at this stage, will be used to build DDEsc2OWL compiler.

The last parameter is the pathname for the input DDesc text file. This file, written by the end-user in the DSL notation according to the rules defined by the generated AG, describes instances that will create the individuals of the ontology.

6. OWL2DSL

This section introduces the module OWL2DSL and explains how it is possible to generate a grammar from a formal description of a domain. A diagram to sketch the architecture of this module is presented in Fig. 3. As can be seen in that figure, this processor is composed of two components: a Parser to transform the input into an internal representation; and a Generator to output the desired Grammar from the internal representation.

The first module is the Ontology Parser. As the research done showed, there are several standard formats to specify an ontology; therefore it is desirable that the front-end module supports most of them (at least the more common). For this

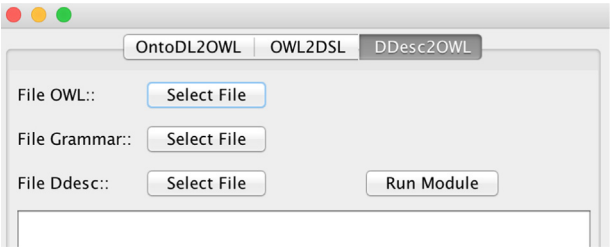

Fig. 2. DDesc2OWL module interface.

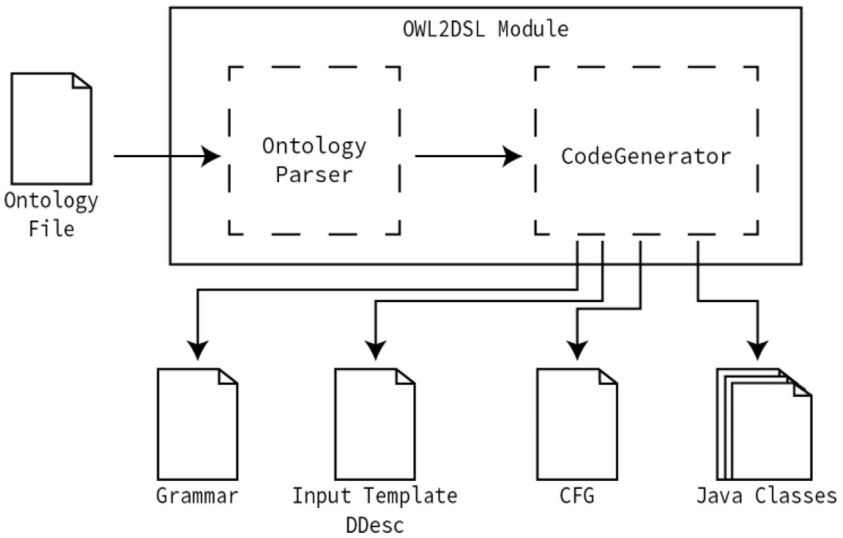

Fig. 3. OWL2DSL architecture.

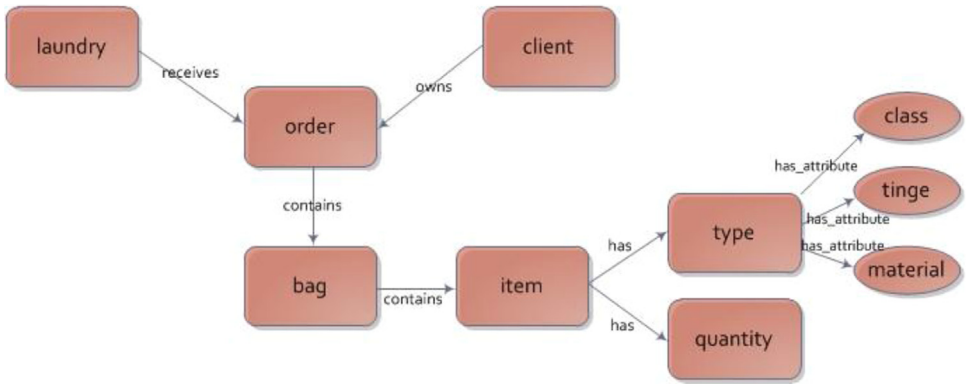

Fig. 4. Ontology graph.

purpose we use the OWL API; so it can process various ontology formats, namely OWL/XML and RDF/XML. This module analyzes the input and decides its type in order to use the appropriate parser without the user interaction.

No matter the parser activated, the final result that outcomes from this component is an *Ontology Object (OO)*, that contains all the information about the concepts (classes) and relations (hierarchies and properties) extracted from the ontology description contained in the input file. This information, gathered in the *OO*, is crucial because it will enable the creation of the desired grammar.

The second module *CodeGenerator*, is a recursive function that traverses the OO internal structure and visits all the *Concepts* and *Relations* (both *hierarchical* and *non-hierarchical*) to generate several files: the desired *attribute grammar* (AG); a simplified version containing only the underlying *context free grammar* (CFG); several *Java Classes* to be used by the dynamic semantics specified in the AG part; and a *DDesc Template* useful for the end-user write sentences in the newly created DSL.

The first output is the AG (in AnTLR notation) that specifies the aimed DSL, enriched with Java code for further processing; the second is a CFG (without attributes and without the Java code for the semantic rules), to allow a better understanding of the generated DSL, and to enable an easier modification (if needed).

This module also generates a *DDesc template*. This functionality allows end-users not familiar with grammar notation to know how to write correct sentences in the new DSL.[4]

Finally, the *CodeGenerator* produces a set of Java classes that are used by the next module of Onto2Gra framework, DDesc2OWL, to properly process the sentences written according to the template above referred. The generated AG imports these classes in order to store internally the information extracted from the source DDesc texts for further process.

Next sections will explain in detail the grammar generation process and the associated rules, the Java classes generation and the DDesc Template.

6.1. A running example

For a better explanation of the generation process we introduce now a more complex running example—a *Laundry company*. Fig. 4 represents an ontology that describes the concepts involved in a laundry business.

Consider a laundry company and the workflow of a collecting point: its front office is responsible for receiving laundry bags from its clients, sending them to the back office (where clothes and linen are really processed) in a daily basis. The bags are sent with an ordering note that identifies the collecting point, and the content of a set of bags. Our aim in this problem is precisely to create a language to describe an ordering note. Going deeply, each bag is identified by a unique identification number, and the name of the client owning it. The content of each bag is separated in one or more items. Each item is a quantified set of laundry of the same type (articles of clothing, linens, etc.), that is, with the same basic characteristics, for an easier distribution at washing time. The collecting point workers should always typify the laundry according to a class, a kind of tinge and a raw-material. The class is either body cloth or household linen; the tinge is either white or colored; and finally, the raw-material is one of cotton, wool or fiber.

Listing 7 shows a textual description of this ontology written in OntoDL notation.

[4] That is, users that do not read easily an AG.

```
Ontology {
  Concepts [
    {Laundry}, {Order}, {Client}, {Bag}, {Item},
    {Type,
            Attributes[ {classes string}, {tinge string}, {material string} ]},
    {Quantity} ]
  Hierarchies[ ]
  Relations[
    {has}, {owns}, {receives}, {contains} ]
  Links [
    {Laundry receives Order},
    {Client owns Order},
    {Order contains Bag},
    {Bag contains Item},
    {Item has max 1 Type},
    {Item has max 1 Quantity} ]
}
```

Listing 7. : Laundry OntoDL

6.2. Grammar generation

The generation of the grammar is systematic and obeys to the set of transformation of rules presented in Section 4. In this section we illustrate this process using the running example previously introduced.

The grammar Axiom, or start symbol, corresponds to the super concept

Thing and is named *thing*.

The first production, with the grammar axiom *thing* in its LHS, has in its RHS all the other *Concepts* that are hierarchically connected to the class Thing (linked to it by the relation is a) as alternatives. These alternatives are followed by one iterator to allow one or more occurrences.

Listing 8 shows one of the outcomes produced by OWL2DSL after processing the ontology in Listing 7.

As can be seen in the example below the seven concepts that are subclasses

of Thing correspond to the seven alternatives in the RHS of the first production. The iterator at the end of this production means that at least one concept shall be derived, but more than one can be described.

```
grammar Laundry;

thing: 'Thing[' (type|laundry|bag|client|item|order|quantity)+ ']'
      ;
type  : 'Type{' typeID ('tinge' tinge)?('material' material)?('classes' classes)? '}'
      ;
typeID : STRING
      ;
tinge: STRING
      ;
material: STRING
      ;
classes: STRING
      ;
laundry: 'Laundry{' laundryID (',' '[' (laundry_receives_order)* ']' )? '}'
      ;
laundry_ID : STRING
      ;
laundry_receives_order: '{' 'receives' 'order' orderID '}'
      ;
bag  : 'Bag{' bagID (',' '[' (bag_contains_item)* ']' )? '}'
      ;
bagID : STRING
      ;
bag_contains_item: '{' 'contains' 'item' itemID '}'
      ;
client:      'Client{' clientID (',' '[' (client_owns_order)* ']' )? '}'
      ;
clientID: STRING
      ;
client_owns_order: '{' 'owns' 'order' orderID '}'
      ;
item: 'Item{' itemID (',' '[' (item_has_quantity)?(item_has_type)? ']' )? '}'
      ;
itemID : STRING
      ;
item_has_quantity: '{' 'has' 'quantity' quantityID '}'
      ;
item_has_type: '{' 'has' 'type' typeID '}'
      ;
order  : 'Order{' orderID (',' '[' (order_contains_bag)* ']' )? '}'
      ;
order_ID : STRING
      ;
order_contains_bag: '{' 'contains' 'bag' bagID '}'
      ;
quantity :'Quantity{' quantity_ID '}'
      ;
quantity_ID: STRING
      ;
```

Listing 8. : Laundry generated Grammar

In the listing above it is possible to see clearly, in the production for the non-terminal 'type', how the attributes in OntoDL (called Data Properties in Protégé) are translated to the grammar formalism.

As previously said, each of the seven non-terminals in the first production, will appear in the LHS of a new production; the RHS will start by a keyword with the name of the respective concept followed by an *IDentifier* that specifies the name of the instance of that concept. After that we need to address the attributes of that Concept, as emphasized above. These attributes are optional, this is, it is not mandatory to define these values in the *DDesc* file. The attribute is represented by its the name and its value, that can be a string, an integer, a float or a boolean.

The next big part of the grammar generation process is concerned with the hierarchical connections between other intermediate concepts. The rule used is the same one applied for the main production'thing'. This means that a concept that is a super-class will have each sub-class as a symbol on a RHS alternative. Alternatives are grouped and an iterator operator is added to allow zero or more occurrences of the alternatives.

Applying recursively this small set of simple rules, all the concepts are represented by grammar symbols and all the triples defined by the relations are translated into grammar productions. The grammar generated can be analyzed by a Language Engineer and modified or adapted, or can be used directly by the last module of *Onto2Gra*.

To complement this generated grammar and in order to be directly processed by the last stage of *Onto2Gra* system, a set of Java classes is also generated as described in the next section. These classes store the information extracted from the DDesc input files. The semantic actions that will be added to the grammar can be very useful to process the collected information in further applications.

6.3. Java class set generation

The Java classes enables the storage of the information that is gather for the DDesc input file processing. This information is equivalent to the Individuals in the OWL terminology. This concrete data will populate that domain with Individuals (or Instances) of the abstract ontology.

The main production of the grammar must save all the information associated with all the classes hierarchically connected to Thing. This fact defines the semantic action that should be added in the AG to this special production.

For each concept (a non-terminal symbol in the AG) a Java Class will be created and the respective code generated. These classes are created based on the *Concept* attributes (data properties) and relations (object properties); the generation process is the same but the code generated is different for each *Concept*. The *Concept* attributes are represented using simple variables. The Hierarchical connections are represented with an ArrayList, as happen for the main class Thing. This allows the storage of all the hierarchical instances specified on the input. The other non-hierarchical relations have the same representation with an ArrayList, but instead of saving Concept object it saves the reference for that object.

The generation of these classes complements the grammar as it was explained in Section 6.2 in order to be able to process the DDesc inputs files. There is also another important class generated, the Main. This executable class helps the implementation of the DDesc Module.

The next section will explain what is a DDesc text file and how the template is generated.

6.4. Template generation

For some future users, it can be difficult to create an input file reading the generated AG. With that in mind, the *CodeGenerator* module of OWL2DSL engine, while processing the ontology to generate the grammar file and the auxiliary Java classes code, will also produce a DDesc input file template. The template is aimed at aiding the end-user to write its sentences in the new DDesc language without the need of reading the AG—instead, he just needs to customize the template.

This template is created with all the alternatives deriving from the grammar. Of course the user does not need to use all the *Concepts* but if needed he can look at the template to understand how it should be specified. This template allows also to understand the cardinality of the relations between *Concepts* and how they should be correctly specified.

The generation of this template adopts a recursive procedure so obvious, following an algorithm similar to the one used for the AG or the Java classes production, that we will not go into details about the process. We conclude the section with an example of its use.

Listing 9 illustrates the template usage, showing a possible input file that was constructed fulfilling the template text generated for the running example.

```
    Thing [
Client{
  " Client 1",
          [
    {owns order " order 2" }
    {owns order " order 4" }
          ]
        }
        Client{
          " Client_2",
          [
            {owns order " order_1" }
            {owns order " order_3" }
          ]
        }
        Item{
          " item_1",
          [
            { has type " type 1" }
            { has quantity "8" }
          ]
        }
        Type{
          " type_1"
          tinge " colour_full"
          classes" colour_cloth"
          material " not fibers"
        }
        Laundry {
          " Laundry_name",
          [
            { receives order " order 1" }
            { receives order " order 2" }
            { receives order " order 3" }
            { receives order " order 4" }
          ]
        }
        Bag{
          " bag_1",
          [
            { contains item " item 1" }
            { contains item " item 2" }
          ]
        }
        Quantity {
          " quantity_private"
        }
        Order{
          " order_1",
          [
              { contains bag " bag 1" }
          ]
        }
       Order{
          " order_2",
          [
              { contains bag " bag 2" }
          ]
       }
       Order{
          " order_3",
          [
              { contains bag " bag 1" }
          ]
       }
       Order{
          " order_4",
          [
              { contains bag " bag 2" }
          ]
       }
    ]
```

Listing 9. : Laundry Process DDesc input

This input DDesc text, a sentence of the Laundry DSL generated, describes the laundry work along one day corresponding to 2 clients with 2 orders each (a total of 4 orders with 1 bag each). One of the items in bag number 1 is composed of colored clothes made out of a raw material not fiber.

7. Project evaluation

To evaluate our project—the approach (the general principals that guided our work), the rules defined, and the implementation (the tool)—we have to consider different perspectives:

- quality of the system outcome, i.e., the DSL's quality;
- coverage of the system, i.e., range of ontologies that can be transformed;
- usability of the approach and tool, i.e., how comfortable a domain expert or a grammar engineer feels using the system; and how easy is to use the system interface.

As language engineers and as authors of a proposal aimed at generating a new language, it is obvious that our main concern is to assess the quality of the produced DSL. This topic is deeply discussed in the next section.

Regarding the coverage, we did not find—neither from a conceptual point of view, nor from a practical usage in different test cases—any restrictions to the domain nor to the ontology to be processed. As already said, we tested the tool with different size ontologies, written in OntoDL or in OWL, from different application domains. Information about the three main case studies were given along the paper: the first example"The Book Index" was described in page 10 using the Listings 2, 3 and 4; a second example "A Laundry Company" in page 14 with Fig. 3 and Listings 5, 6 and 7; and a third case study "Language Processing Domain" was referred in Figs. 4 and 5 in pages 24 and 25. Other smaller examples were tested just with the purpose of corroborating partially the evidences here discussed. However we are aware that we need, in the future, to experiment the tool with bigger ontologies to collect evidence about the system scalability.

Concerning the third perspective above, we plan, as future work, to collect ideas from the usability tests defined by the HCI community (Human–Computer-Interface researchers) and then to draw effective experiments with end-users. However even without special usability tests, we can affirm that our proposal is easy to use for grammar experts because the system accepts ontologies in different formats (our own lightweight language, or the standard OWL)—no need at all to rewrite a previous ontology to be transformed by Onto2DSL—and after selecting the input file, all the transformation process is automatic and the interface just provides the necessary buttons to set up from one phase to the next phase. The generated grammar can be opened and manipulated with any text editor.

The main role of the domain experts will be to collaborate with language engineers in the ontology definition (discussing the concepts and relations involved in each domain). Moreover, the use of OntoDL (our DSL to specify ontologies that is much more user-friendly than other more complete and elaborated notations) can help in this task. We completely agree that a visual editor for OntoDL could be very important to aid domain experts in this task. If a standard XML notation should be used by the project team, the visual editor provided by Protégé is recommend.

7.1. Measuring the quality of the generated DSL

Concerning the assessment of a DSL quality we found in the literature different approaches according to the perspective to be measured, as will be reviewed in the next paragraphs. Before evaluating the DSL produced by our tool, we will look for the possible criteria to be adopt. Some of them address the quality of the interface (usability), others the quality of the underlying grammar, others the quality of the processing task, and others present guidelines for DSL design.

7.1.1. Language usability

In [1], the authors propose a systematic approach based on User Interfaces experimental validation techniques to assess the impact of the introduction of DSLs on the productivity of domain experts.

This perspective is not completely inline with the objectives of our work. We intend to analyze and evaluate the grammar derived from the ontology in terms of its efectiveness to implement the DSL, not from the point of view of usability in the modeling activity.

7.1.2. Grammar metrics

The authors in [24] present a relation between the meta-models used to create DSLs and the appropriateness of the DSLs created. For that, they define two metrics: DS magnitude and DS depth. The first metric is oriented towards the assessment of the metamodel in order to enable the reasoning about the derived DSLs' appropriateness. The second metric considers the domain to evaluate the appropriateness of the DSL. It is important to measure the complexity of the metamodel (let's say, the grammar) but we think it is not trivial to infer from it the complexity of the language. For instance, in our case we can make some shortcuts in grammar productions and improve the grammar readability, however this does not imply any improvement in the language readability. The authors state that the appropriateness of a language depends not only on its syntax and semantics but also on the user ability to express solutions. The appropriateness is defined as being able to *cover everything that must be covered but nothing more*. For domain experts with little programming experience a rich set of domain specific commands can be added to the language. So, they will prefer bigger value of DS magnitude and small value of DS depth.

Also in the area of grammar/language engineering, authors as [23] evaluate the language quality based on grammar metrics: number of productions, number of terminals and non-terminals, McCabe Cyclomatic complexity (greater number of alternatives for each non-terminal, greater parsing conflicts), Halstead Effort (HAL-effort to understand the grammar), average of right hand side size (the greater AVS is, the greater will be the stack size). Once more these metrics are concerned with grammar quality and the inherent parser efficiency. Like in the previous case, grammar metrics are relevant to assess the grammar generated by our OWL2DSL tool, however it remains difficult to come up with conclusions about the language

quality or complexity. The same occurs in [31] where the authors propose metrics based on LR table metrics. So, the question is, how to relate the quality of the grammar (grammar metrics) with the quality of the language.

7.1.3. Language design

In [28] the authors define a very interesting set of techniques to improve the language. The main idea is called *Corpus Evaluation*; it is done analysing a large set of programs written in that language. Over those programs four types of analyses can be performed: *clone analysis* to detect and analyse clones in order to identify the need for new metamodel elements; *cluster analysis* to identify strongly associated elements (sub-languages); *semantic-based analysis* to detect semantically similar usages to determine the most optimal version; and *usage analysis* to identify highly utilized metamodel elements to direct focus on improvement evaluations. The authors also propose formal techniques to confirm the ability to create valid instances, use the user feedback to analyse user-based experience and opinion, and perform runtime-based evaluation to identify common user errors in runtime logs.

In [12] the authors present a very complete set of guidelines to construct new language. Starting at the early stages of defining the language purpose, they refer issues related with the language design (choosing the type of the new language, reusing or not existing languages, etc.); then they discuss the content and syntax of the language, coming up with the following ideas:

- Concerning the *language content* the main principal is *to keep it as simple as possible*, and for that they suggest:

1. reflect only the necessary domain concepts;
2. avoid unnecessary generality (use lots of domain concepts);
3. limit the number of language elements;
4. avoid conceptual redundancy.

- Concerning concrete syntax:

1. adopt existing domain expert notations;
2. use descriptive notations (use keywords with a widely-accepted meaning);
3. define keywords easily identifiable;
4. restrict number of keywords to memorize easily the syntax of the language);
5. make elements distinguishable;
6. use syntactic sugar appropriately (the syntactic sugar can improve the readability but can also improve the error proponess);
7. permit comments;
8. allow for organizational structure of the code;
9. assure a balanced compactness and comprehensibility (use short terms to frequently used elements and long terms to rarely used ones);
10. use the same style everywhere;
11. identify usage conventions.

We think that these are important guidelines relevant for the assessment of the DSLs derived from our grammars; in the next section, we will analyse each one and verify if our generated language and grammar should be improved.

7.1.4. Language metrics

Henriques in [4,9] identify three main *factors* that shall be considered to define and assess the quality of a language in what concerns its *usability*[5]: ease of *learning*; ease of *writing* (productivity); and ease of *comprehending*[6] (maintainability). The author defines a set of eight *characteristics* that can be appraised in order to evaluate a language[7]: expressiveness, documentation, uniqueness, consistency, scalability, extendability, reliability, and modularity.

Expressiveness is concerned with the power of the language to express all the concepts and actions of the domain. If we have terminals for all concepts and relations it is possible to express everything related with this domain. The way these terms are organized (relations between concepts) may influence the complexity of the language. *Documentation* refers to the availability of mechanisms to enrich a sentence with extra text with complementary information about the message itself. *Uniqueness* is related with the number of alternative paths that can be followed to transmit a message, i.e., to write sentences with the same meaning. *Consistency* is concerned with the way identical ideas are expressed in a language. The language is consistent if similar operations are writing in a similar way. Notice that the clone analysis (used to detect parts of the program that are repeated), and the cluster analysis (used to find parts of the program that appear always together) can

[5] The language Recognition/Processing efficiency should also be considered, but it also depends on other issues that are not directly related with the language design. So, it will not be discussed in the present context.

[6] How easy is to understand the language sentences.

[7] Please see [4] for an English, although very summarized, version of the basilar document referred.

be an useful instrument to study the language consistency. *Scalability* is related with the capability to keep the language quality (without degradation) when the size of the sentences increase significantly; this is, when the problem grows in size or complexity, the sentences should be kept simple and easy to write and to read/understand. *Extendability* reflects the capacity of the language to be extended with new symbols, new syntactic constructors or new semantics. *Reliability* means that the interpretation of the language sentences actually corresponds to what it was expected to be; a semantics rigorously specified contributes to increase our confidence. In case of programming languages, it also comprises the error handling mechanism and the way the system copes with exceptions during execution. *Modularity* is related with the mechanisms provided to organize the code in modules, and how do they can be combined/composed.

The presence of each one of the characteristics above raises up the language usability, and so increases its quality. Some of them impact on ease of learning it, but most of them affect the writing (productivity), and the understanding—lets say, in the user satisfaction.

Other parameters like the *verbosity*, or wordiness, also influence the language usability. The size of keywords and signals, and number of terminals used in a correct sentence increase the language readability (making it easier to comprehend), may be make easier learning it, but decreases the productivity and augments the error proneness. It also has a negative impact into the scalability.

7.2. Assessing the DSL

Now that the main approaches to the appraisal of grammar/language quality have been reviewed, it is possible to comment on the quality of the grammars generate by Onto2Gra system and the derived languages.

Regarding the suggestion for language content proposed in [12], we can say that our ontological based approach clearly helps to keep the language simple, as it allows to use exactly the terms of the domain. The language symbols are limited to the domain concepts and their relations, because all the grammar terminal and non-terminal symbols are derived from the given ontology, and nothing more than that is included. The ontology will impose the appropriate level of generality and limit the number of language elements, according to the consensus of the domain experts. However, we recognize that some redundancy or unused elements can appear as the result of the automatic transformation process, because the ontology may contain islands (isolated concepts) or more concepts than those that are actually involved in a concrete situation. This is a drawback that can be tackled a-posteriori, cleaning the generated grammar. Actually, Onto2DSL generator will produce grammar symbols (and respective derivation rules) for all the concepts present in the given domain (because all of them are present in the input ontology); so, if the DSL is intended to be used in a more restricted sub-domain, the grammar engineer can prune from the generated grammar all the symbols and rules that do not pertain to this sub-domain.

In what concerns the syntax recommendations recalled above, in our approach, the language syntactic sugar is added into the grammar systematically, according to a set of pre-defined translation rules. Obviously, this syntactic sugar is generated in the same way and style for all the output grammars following our sensibility and taste. So, it can be more or less heavy, but at any moment it can be generally changed and adapted to other circunstancies. It is only a matter of modifying once the referred translation rules.

Grammar terminals are derived from the domain concepts, inheriting their names. Some work can be done after the generation of the grammar in order to minimize the size of the keywords, and consequently control/tune the verbosity of the language.

Since the grammar is generated from the same templates, obviously the style of the produced language is always the same. This assures that the new language will be *consistent* and *uniform*.

Due to the methodological approach followed and regarding the usability of the language generated from the ontology, we would say that: it is *expressive*, as it allows to discourse appropriately about the domain implicit in the given ontology; it allows just one way to express the same relations (*uniqueness*); it is *consistent*, as already argued; it is *semantically well-defined* and so static checks can be done to make it *reliable*.

Finally we recognize that the language: does not include (by the moment being) *comments* to insert documentation alongside the sentences—however, it is just a lexical detail easily outcome; its scalability (at present stage) can be slightly compromised due to its verbosity; and no constructions for modularity are provided, nor extension mechanisms.

8. Conclusion

Aware of the importance of Domain Specific Languages (DSLs), and of the need to support their development, we have research the possibility of derive systematically a grammar from a domain specification. To accomplish this aim we propose to consider an ontology as a formal description of knowledge domain suitable to enable the derivation of a language specific for that domain. After a careful analysis of both formalisms, we have settled down some rules to obtain grammar symbols from the ontology concepts, and grammar derivation rules from the ontology relations. The mapping so far defined allows to generate a Context Free Grammar, CFG, from a given ontology. Moreover we add other translation rules to create attributes and semantic rules (that compose an Attribute Grammar, AG) that allow to process the sentences of the new language defined by the referred CFG.

The proposal schematized above was implemented and a prototype, called OWL2Gra, was created as a proof of concept. The system was tested and the grammars generated from different ontologies were checked for correctness. We also assessed the languages produced by our engine considering the criteria defined by different authors, as discussed in the last section—we came out with a positive balance. Actually from an OWL ontology we can derive a language that is expressive, offers uniqueness, is consistent, and is semantically well defined. Syntactic sugar and other language details can be easily tuned handcrafting the grammar outputted allowing its improvement. In the paper, two additional modules included in the Onto2Gra system were briefly presented: Onto2OWL that translates an ontology written in a light notation for ontology definition, OntoDL, into OWL notation; and DDesc2OWL that processes the sentences written in the new DSL in order to populate, or instantiate, the original ontology.

As future work, the main research objective is to consider the axioms of the ontology and to understand how can they be transformed into semantic rules of the generated AG. In fact, in the present version, we do not process all the ontology axioms (if they exist in the input ontology): we just consider the axioms concerned with cardinality constraints which are reflected in the language syntax; Considering other axioms we will be able to increment the generate AG with contextual conditions to check static semantics. Also some deduction mechanisms can also be derived from the ontology inference rules, in the case such a final behavior is expected. However, the generation of other translation rules to implement the specific DSL dynamic semantics is not possible as this component is not specified in the ontology.

Acknowledgments

This work has been supported by FCT Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2013.

References

[1] Barišić A, Amaral V, Goulão M, Barroca B. How to reach a usable DSL? Moving toward a systematic evaluation Recent advances in multi-paradigm modeling in electronic communications of the EASST 2011;vol. 50.
[2] Čeh I, Črepinšek M, Kosar T, Mernik M. Ontology driven development of domain-specific languages. Comput Sci Inf Syst 2011;8(2):317–42.
[3] Chomsky N. Context-free grammars and pushdown storage. MIT; 1962.
[4] Cruz J, Henriques PR, da Cruz D. Assessing attribute grammars' quality: metrics and a tool. In: Sierra-Rodriguez J-L, Leal JP, Simões A, editors. Proceedings of the 2015 symposium on languages, applications and technologies, SLATE'15. Fundacion General UCM; 2015. p. 219–24.
[5] da Silva AR. Model-driven engineering: a survey supported by the unified conceptual model. Comput Lang Syst Struct 2015;43:139–55.
[6] Dean M, Schreijber G. OWL Web Ontology Language Reference. Recommendation 2004.
[7] Deransart P, Jourdan M, Lorho B. Attribute grammars: main results, existing systems and bibliography. Springer-Verlag; 1988.
[8] Gruber TR. Toward principles for the design of ontologies used for knowledge sharing. Kluwer Academic Publishers; 907–28.
[9] Henriques PR. Brincando às Linguagens com Rigor: Engenharia Gramatical. Habilitation in Computer Science (Technical Report), Dep. de Informática, Escola de Engenharia da Universidade do Minho, habilitation monography presented and discussed in a public session held in April 2012 at UM/Braga; 2011.
[10] Hopcroft JE, Motwani R, Ullman J. Introduction to automata theory, languages, and computation. 3rd ed.Addison-Wesley; 2006 Ch. 5 – Context-Free Grammars and Languages.
[11] Horridge M, Knublauch H, Rector A, Stevens R, Wroe C. A practical guide to building OWL ontologies using the protege-OWL Plugin and CO-ODE Tools Edition 1.0; 2004.
[12] Karsai G, Krahn H, Pinkernell C, Rumpe B, Schindler M, Völkel S. Design Guidelines for Domain Specific Languages. CoRR from Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM' 09) abs/1409.2378; 2014.
[13] Kleppe A. In: Software language engineering: creating domain-specific languages using metamodels. 1st ed.Addison-Wesley Professional; 2008.
[14] Knuth DE. Semantics of context-free languages. Math Syst Theory 1968;2(2):127–45.
[15] Kosar T, López PE, Barrientos PA, Mernik M. A preliminary study on various implementation approaches of domain-specific language. Inf Softw Technol 2008;50(5):390–405.
[16] Kosar T, Oliveira N, Mernik M, Pereira MJV, Črepinšek M, da Cruz D, Henriques PR. Comparing general-purpose and domain-specific languages: an empirical study. Comput Sci Inf Syst 2010;7(2):247–64.
[17] McGuinness DL, Harmelen FV. Owl web ontology language overview; 2004.
[18] Mernik M, Heering J, Sloane AM. When and how to develop domain-specific languages. ACM Comput Surv 2005;37(4):316–44.
[19] Mernik M, Lenič, M, Avdičauševič, E, Žumer V. Compiler/in-terpreter generator system LISA. In: IEEE Proceedings of 33rd Hawaii international conference on system sciences; 2000.
[20] Mernik M, Žumer V. Domain-specific languages for software engineering. Proceedings of the Hawaii international conference on system sciences 2001;9:4002.
[21] Oliveira N, Pereira MJV, Henriques PR, da Cruz D, Cramer B. Visuallisa: A visual environment to develop attribute grammars. ComSIS – Comput Sci Inf Syst J Spec Issue Adv Lang Relat Technol Appl 2010;7(2):266–89.
[22] Parr T. The definitive ANTLR reference: building domain-specific languages.Raleigh: The Pragmatic Bookshelf; 2007.
[23] Power JF, Malloy BA. A metrics suite for grammar-based software: research articles. J Softw Maintenance Evol 2004;16(6):405–26.
[24] Rožanc I, Slivnik B. On the appropriateness of domain-specific languages derived from different metamodels. IEEE; 190–5.
[25] Serra I, Girardi R. A process for extracting non-taxonomic relationships of ontologies from text. Intell Inf Manag 2011;3:119.
[26] Studer R, Benjamins VR, Fensel D. Knowledge engineering: principles and methods. Data Knowl Eng 1998:161–97.
[27] Sun Y, Demirezen Z, Mernik M, Gray J, Bryant B. Is my DSL a modeling or programming language? In: Proceedings of 2nd international workshop on Domain-Specific Program Development (DSPD). Nashville, Tennessee; 2008.
[28] Tairas R, Cabot J. Corpus-based analysis of domain-specific languages. Softw Syst Model 2013:1–16.
[29] Tairas R, Mernik M, Gray J. Models in software engineering. Using Ontologies in the Domain Analysis of Domain-Specific Languages 2009:332–42.
[30] Uschold M, Gruninger M. Ontologies: principles, methods and applications. Knowl Eng Rev 1996:93–136.
[31] Črepinšek M, Kosar T, Mernik M, Cervelle J, Forax R, Roussel G. On automata and language based grammar metrics. ComSIS – Comput Sci Inf Syst J Spec Issue Compilers Relat Technol Appl 2010;7(2):309–29.

[32] Visser E. WebDSL: A case study in domain-specific language engineering. In: Lammel R, Saraiva J, Visser J, editors. Generative and Transformational Techniques in Software Engineering (GTTSE 2007). Lecture Notes in Computer Science. Springer; 2008.
[33] Walter T, Parreiras FS, Staab S. OntoDSL: an ontology-based framework for domain-specific languages. In: Schurr A, Selic B, editors. Model driven engineering languages and systems. Vol. 5795 of Lecture Notes in Computer Science; 2009. p. 408–22.
[34] Walter T, Parreiras FS, Staab S. An ontology-based frame-work for domain-specific modeling. Softw Syst Model 2014;13(1):83–108.