Using ontology in the development of domain-specific languages

Ines Čeh, Matej Črepinšek, Tomaž Kosar, Marjan Mernik

Faculty of electrical engineering and computer science, Smetanova 17,
2000 Maribor, Slovenia
{Ines.Ceh, Matej.Crepinsek, Tomaz.Kosar, Marjan.Mernik}@uni-mb.si

Abstract. Domain-specific languages (DSL) are programming languages devoted to solve problems in a specific domain. Development of a DSL includes the following phases: decision, analysis, design, implementation and deployment. The least known and examined are analysis and design. Although various formal methodologies exist, the domain analysis is still done informally, most of the time. A common reason why formal methodologies are not used as often as they could be is that they are very demanding. Instead of developing a new, less complex methodology, we propose that domain analysis could be replaced with a previously existing analysis in some other form. A particularly suitable form for such is ontology. This paper focuses on ontology based domain analysis and how it can be incorporated into the DSL design phase. We present preliminary results of the Ontology2DSL framework, which can be used to help transform ontology to DSL grammar.

Keywords: domain-specific language, domain analysis, ontology

1 Introduction

Programming languages are used for human-computer interaction. Depending on the purpose of their use, programming language can be divided into general-purpose languages (GPL) and DSL [1]. GPL, such as Java, C and C#, are designed to solve problems from any problem area. In contrast to GPLs, DSLs, such as Latex, SQL and BNF, are tailored to a specific application domain.

When developing new software a decision must be made as to which type of programming language will be used; GPL or DSL. The issue is further complicated if an appropriate DSL does not exist. Then, the decision is whether to start to develop with a GPL language or to start with the development of the required DSL and then develop the software system with it. Reasons for the use of DSL are as follows: easier programming, re-use of semantics, the easier verification and programmability for the end-users. However, DSL also have their disadvantages, for example high development costs. The key is to answer the question: “When to develop a DSL?” The simplest answer to this question is: a DSL should be developed whenever it is necessary to solve the problem, which belongs to a problem family and we expect that in the future more problems from the same problem family will appear. A more detailed response can be found in [1].

DSL development consists of the following phases: decision, analysis, design, implementation and deployment [1]. DSL development phases are not equally researched. The least known and examined phases are the analysis and design.

The knowledge on the problem domain and its definition is achieved at the domain analysis phase. Various methodologies for domain analysis have been developed. Examples of such methodologies include: DSSA (Domain Specific Software Architectures) [2], FODA (Feature-Oriented Domain Analysis) [3], and ODM (Organization Domain Modeling) [4]. Often, formal methodologies are not used due to complexity and the domain analysis is done informally. This has the consequence of complicating future DSL development. Even if the domain analysis is done with a formal methodology, there aren't any clear guidelines how the output from domain analysis can be used in a language design process. The outputs of domain analysis consist of domain-specific terminology, concepts, commonalities and variabilities. Variabilities would have been entries in the design of DSL, while terminology and concepts should reflect in the DSL constructs, and commonalities could be incorporated into the executing DSL environment. Although it is known where the outputs of the domain analysis should be used, there is a need for clear instructions on how to make good use of the information, which are retrieved during the analysis phase, in the design stage of the DSL.

To partially solve aforementioned problems, we propose that domain analysis (hereinafter referred to as classic domain analysis (CDA)) can be performed with the use of existing techniques from other fields of computer science. A particularly suitable is ontology [5]. Ontology provides a vocabulary of a specialized domain. This vocabulary represents the domain objects, concepts and other entities. Some type of domain knowledge can be obtained from the relationships of the entities, presented by the vocabulary. Ontologies in the CDA have already been used in [6]. Whereas Tairas et al. apply ontology in the early stages of domain analysis to identify domain concepts; we propose that ontology replaces the CDA. They [6] also investigated how ontologies contribute to the design of the language. Ontologies in connection with DSL are also used by other authors. Guizzardi et al. [7] propose the usage of an upper ontology (top-level ontology) [8] to design and evaluate domain concepts. Walter et al. [9] apply ontologies to describe DSL. Bräuer and Lochmann [10] propose an upper ontology to describe interoperability among DSLs.

The proposed solution of the first problem, the use of ontologies, has a significant effect on the second problem related to CDA. It translates the problem »How to make good use of the information, retrieved during the analysis phase, in the design stage of the DSL?« into the problem »How to make good use of the information contained in an ontology in the design stage of DSL?« This paper focuses on ontology based domain analysis (OBDA) and how it can be incorporated into the DSL design phase. We present preliminary results of the Ontology2DSL framework, which can be used to help transform ontology to DSL grammar.

The organization of this paper is as follows. Section 2 is intended to represent the similarities and variabilities between the CDA and OBDA. Section 3 presents the development of grammar from ontology as well as the framework Ontology2DSL. The conclusion and future work are summarized in Section 4.

2 Domain analysis

2.1 Classic Domain Analysis (CDA)

The goal of CDA is to select and define the domain of focus and collect appropriate domain information and integrate them into a coherent domain model; the result of CDA [11]. A representation of the domain system properties and their dependencies is the domain model. The properties are either common or variable which is represented in the model along with the dependencies between the variable ones [11]. Beside the development of the domain model, CDA also includes domain planning, identification and scoping [11].

CDA can incorporate different methodologies. Methodologies differ based on the degree of formality, information extraction techniques or their products. We have listed the most known methodologies in the introduction. FODA has been proven as the most commonly used formal methodology in DSL development.

FODA is a CDA method that was developed by the Software Engineering Institute. It is known for its models and feature modeling. In FODA, feature is an end-user characteristic of a system. A FODA process consists of two phases: context analysis and domain modeling. The goal of context analysis is to determine the boundaries (scope) of the analyzed domain. The purpose of domain modeling is to develop a domain model. FODA domain modeling phase is comprised of the following steps: information analysis, features analysis and operational analysis. The main goal of information analysis is to capture domain knowledge in the form of domain entities and links between them. The result of information analysis is the information model. The result of feature analysis is feature model, which is presented below. Operational analysis results in the operational model. It represents how the application works and covers the links between objects in the informational model and the features in the feature model. An important product from the phase of domain modeling is the domain dictionary. It defines the terminology used in the domain and it also includes textual definitions of domain concepts and features.

A feature model consists of:

- *Feature diagram* (FD) represents a hierarchical decomposition of features and their kinds (mandatory, alternative, and optional feature). Mandatory features are features that each system must have in the domain. Alternative features are features of which a system can possess only one at a time. Optional features are features that system may or may not have. A system can also have more than one feature at a time. These features are called or-features. Features are also classified as atomic or composite. Whereas atomic features cannot be further subdivided in other features, composite features are defined in terms of other features. The root node of the diagram represents a concept and the remaining nodes represent features. An example of a feature diagram is shown in Fig. 1.
- *Feature definitions* describe all features (semantics).
- *Composition rules for features* describe which combinations are valid or invalid.

- *Rationale* for features represents reasons for choosing a feature.

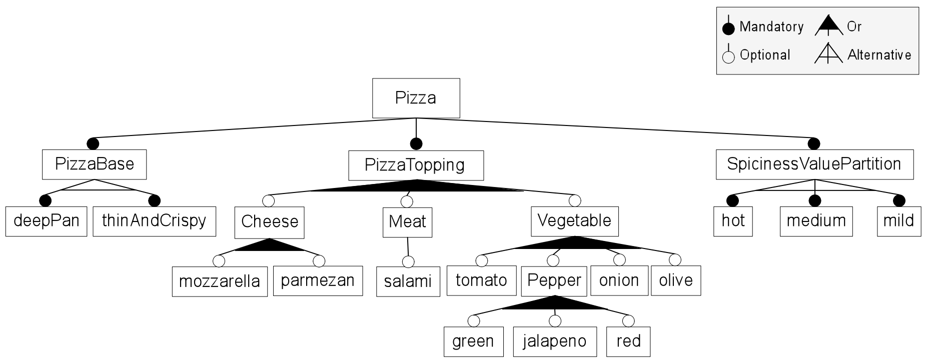

Fig. 1. Feature Diagram for a concept of a pizza.

Fig. 1 represents a simple FD of a pizza. The root node of the diagram, Pizza, represents a concept; the remaining nodes represent its features. Whereas mandatory features are indicated by a filled circle, optional features are indicated by an empty circle. Alternative and or-features are both indicated by a triangle, the former with an empty one and the latter with a filled triangle. The names of atomic features are written in lower-case while the composite features are written with their first letter in upper-case. Each pizza is composed of the pizza-base and at least one topping. The pizza-base is either "DeepPan" or "ThinAndCrispy", never both in the same pizza. The toppings are one or more of the following: "Cheese", "Meat" or "Vegetable". One pizza can have multiple toppings as well as multiple toppings of the same type (cheese topping of both "mozzarella" and "parmezan"). The pizza can be hot, medium or mild according to its spiciness (only one at the time).

Feature models are not only represented in the visual form of FDs but also in the textual form. Van Deursen and Klint [12] have proposed the feature description language (FDL) for the textual representation. The FDL definition constitutes of the feature definitions followed by a colon (":") and the features expression. Possible feature expression forms are presented in [12]. FDL exceeds the graphic feature diagram in the terms of expressive power and is appropriate for automatic processing. FD for pizza in FDL is listed below:

```
Pizza: all ( PizzaBase, PizzaTopping,
SpicinessValuePartition )
PizzaBase: one-of ( deepPan, thinAndCrispy )
PizzaTopping: more-of ( Cheese, Meat, Vegetable )
Cheese: more-of ( mozzarella, parmezan )
Meat: all ( salami? )
Vegetable: more-of ( tomato, Pepper, onion, olive )
Pepper: more-of ( green, jalapeno, red )
SpicinessValuePartition: one-of ( hot, medium, mild )
```

An important role of the FDs is to describe the variability of the programming system. The number of all possible configurations per system can be calculated with the use of variability rules, presented in [12].

Constraints, which are intended for variability reduction, are an optional component of the FDs. The constraints are enforced with the satisfaction rules [12]. The constraints are of two types [12]: diagram constraints and user constraints. The former include the "A1 requires A2" (if feature A1 is presented, then feature A2 should also be presented) and "A1 excludes A2" (if feature A1 is presented, then feature A2 should not be presented) constraints, while the latter include the "include A" (feature A should be present) and "exclude A" (feature A should not be present) constraints.

2.2 Ontology based Domain Analysis (OBDA)

There are many definitions of ontology in the literature and one of the most commonly used definitions is that of Gruber. He defined ontology as a "formal, explicit specification of shared conceptualization" [5]. Formal refers to the fact that it is machine readable. The specification is explicit because it summarizes the concepts, properties and relations between concepts. Furthermore, shared conceptualization contains knowledge that a group of experts has agreed upon. Conceptualization refers to the fact that it incorporates the target domain completely.

Ontologies are commonly encoded using ontology languages. Ontology languages can be divided in two major groups: traditional (i.e. Flogic, Ontolingua) and web-based languages (i.e. RDF(S), OWL, OWL 2) [13]. Recently, a new group of languages, rule-based (i.e. RuleML, SWRL), has emerged. These languages differ in their purpose and in their expressive power. The main requirements for an ontology language are: well defined syntax, well defined semantics, efficient reasoning support, sufficient expressive power and convenience of expression [14].

OWL is the most commonly used ontology language. It has three sublanguages; OWL Full, OWL DL and OWL Lite [14], [15]. These sublanguages have different levels of expressiveness. Whereas OWL Full is the most expressive, OWL Lite is the least expressive. Only OWL-DL allows automated reasoning.

The three components of OWL are: classes, properties, and individuals. Classes are interpreted as sets that contain individuals. Classes may be organized into a hierarchy. This means that a class can subsume other classes or it can be subsumed by other classes. The consequence of the subsumption relation is inheritance. Inheritance refers to the inheritance of properties which the children inherit from their parents. Whereas some ontologies only allow single inheritance, most ontologies, like OWL, allow multiple inheritance. OWL defines two special classes called „Thing“ and „Nothing“. Class Thing is the most general class and it is the superclass of every class that is included in ontology. Class Nothing is the empty and it is subclass of every included class. The class hierarchy for the truncated version of the previously existing Pizza ontology (PO) is shown in Fig. 2. The PO is used as an example in a practical guide to building OWL ontologies using Protégé [15]. PO has been choosen as an example because it includes the majority of the OWL features. The PO, written in OWL-DL, describes various pizzas based on their toppings.

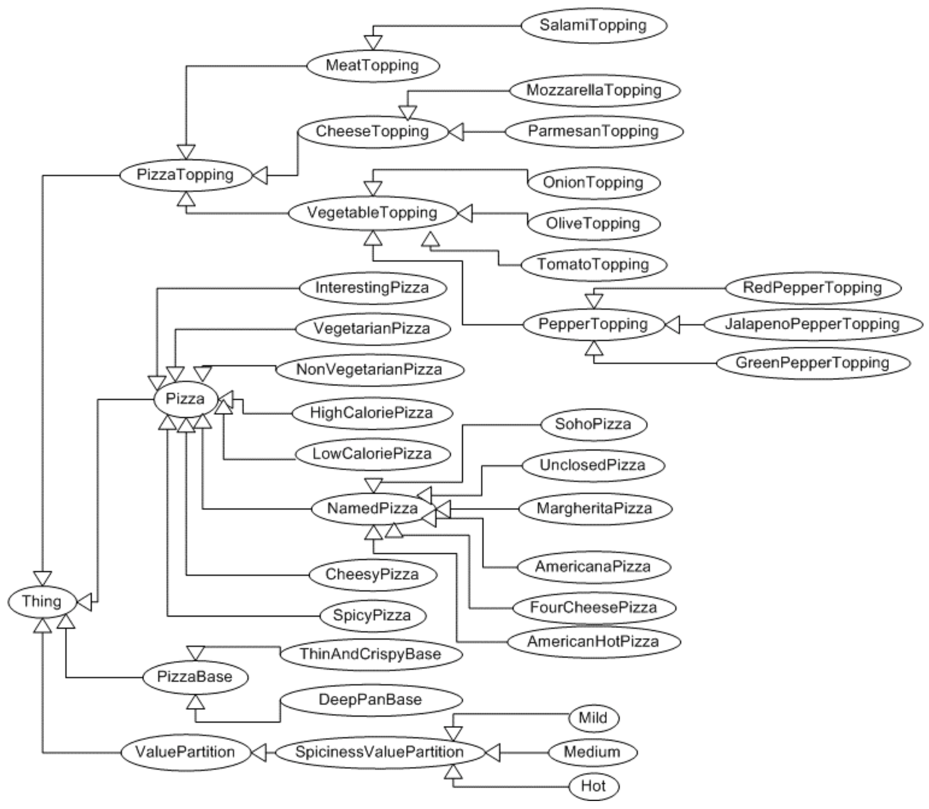

Fig. 2. Class hierarchy of Pizza Ontology.

Fig. 2 shows the class hierarchy of the PO used in this paper. The classes are represented with ellipses. All the classes are subclasses of the Thing class.

The second component, the properties, is a binary relation. OWL defines two main kinds of properties; object properties (i.e. hasTopping) and datatype properties (i.e. hasCaloricContentValue). Whereas object properties relate objects to other objects, datatype properties relate an object to datatype values. OWL supports XML schema primitive datatypes. The third component, the individuals, is the basic component of ontology. They represent objects in the domain of discourse. They can be concrete individuals (i.e. animals, airplanes, and people) as well as abstract individuals (i.e. words and numbers).

The relationships between classes are the means of the class definition in OWL. Such classes can be defined with the use of restrictions. Three main categories of restrictions that exist in OWL: quantifier restrictions (existential and universal), cardinality restrictions and „hasValue“ restrictions [15].

2.2 Comparison of CDA and OBDA

Both analysis incorporate a concept vocabulary, enable the display of property and class hierarchies, and provide a constraint mechanism. The CDA uses this mechanism for variability reduction while the OBDA uses it for the description of class properties. Both types of analysis describe semantics and are machine readable. The CDA differs from OBDA in its capability to record the reasons for the use of particular property (rationale) and the calculation of all possibilities. OBDA, on the other hand, provides the existence of objects, reasoning and querying. Numerous tools are available for it and ontologies are created across diverse research areas and are therefore available for use. The comparison shows that OBDA is capable of most of what the CDA is capable. The advantages of ontology are reasoning and querying, because they enable the validation of ontology. Valid ontology significantly reduces or prevents errors in DSL development. Semantics, that are inherently defined with the ontology, is also of great use when developing language semantics. Existing tools provide easy access to the ontology and enable efficient information extraction procedures. It is also a very important fact that ontologies are present in different research areas. That provides the method for elimination of domain analysis phase in DSL development and might significantly reduce the time needed for the language development.

Table 1. Comparison of CDA and OBDA

Property	FD + FDL	OWL ontology
Concept vocabulary	Features names	Name Class or property
Hierarchy	Feature diagram	Class hierarchy
Constraints	FDL constraints	Restrictions
Rationale	FD rationale properties	No
Objects	No	Individuals
Possible combinations	Variability rules (FDL)	No
Reasoning support	No	Reasoners (i.e. FaCT++)
Machine readable	Yes	Yes
Tools	In its infancy	Yes (i.e. Protege)
Semantics	Yes	Sets of relations
Domain analysis in use	No	Existing ontologies
Query support	No	Yes (DL Query)

The comparison leads to the conclusion that the CDA can indeed be replaced with OBDA, primarily because the ODBA provides everything needed for DSL development and adds new capabilities.

3 Language design

The grammar design is a feature of the Ontology2DSL framework. The framework enables the transformation of the OWL document to an appropriate internal data structure. The data structure is then transformed with the use of transformation patters. The resulting output is in the form of grammar and one or more programs. A DSL engineer that uses various tools at his/her disposal reviews them. If irregularities are found, they are resolved in accordance to their type in either the ontology or the transformational patterns. With regard to the type of the fix applied, the tool then uses new patterns on the old ontology, old patterns on the new ontology or new patterns on the new ontology. The process is repeated until the engineer finds no more irregularities, which finally results in the language grammar definition and one or more programs. The framework, besides the grammar development, can be supported with the development of DSL tools. They can be developed by the DSL engineer with the language development tools such as LISA [16]. Ontology2DSL framework is presented as a workflow diagram on Fig. 3.

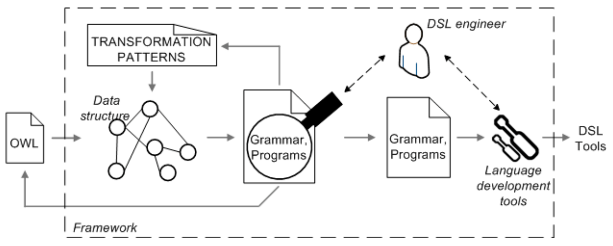

Fig. 3. Ontology2DSL framework.

3.1 Designing DSL grammar

Before starting with an explanation of basic steps of Ontology to DLS transformation (O2DSL), target ontology needs to be well understood. Language designer must understand what ontology describes and why it was designed. Moreover, language designer needs to know what are DSL requirements and what is the purpose of DSL. In most cases the DSL requirements and the ontology, do not overlap in all concepts.

In this paper, methodology O2DSL is demonstrated on a PO example. Single ontology, as well as PO example, can be used to develop many different DSLs. The purpose of the one presented here is a DSL to describe pizzas that can be queried by pizzas characteristics. Concepts of the simple pizza query language are not described in the given PO, therefore are omitted from this example transformation. The DSL is named as Pizza Language (PL) and a result of O2DSL is the obtained language grammar [17]. As a side effect of transformation, domain data that is described in DSL programs is extracted.

Basic concept transformation. Ontology classes are interpreted as abstract groups and represent production rules in grammar non-terminals. In transformation classes, names have been used for grammar non-terminal names. Sequences and alternatives that describe production rules are obtained through class dependences and class hierarchy. For example, class hierarchy for `PizzaTopping` (Fig. 2) describes the group of toppings `CheeseTopping`, `MeatTopping` and `VegetableTopping`. Subgroup `CheeseTopping` includes subgroup of classes `ParmesanTopping` and `MozzarellaTopping`. In O2DSL transformation class hierarchy is transformed into production alternatives. Grammar example:

```
PizzaTopping ::=  CheeseTopping | MeatTopping |
                  VegetableTopping
CheeseTopping ::= ParmesanTopping | MozzarellaTopping
```

Generalization extraction. This pattern extracts abstract group from ontology and presents it as a configurable external element (program or constant attribute). In most cases this pattern is used for leaf classes in class hierarchy. For example in `PizzaTopping` (Fig. 2) leafs can be in general named `Topping`, that represents terminal in DSL grammar. Information of `ParmesanTopping` is then moved in DSL program as `Topping` instance (`parmesan`). For ontology class `CheeseTopping`, the following production is derived:
`CheeseTopping ::= Topping` and `Topping ::= #string`
Program fragment examples: `'parmesan'` and `'mozzarella'`.

Often, desired feature of DSL is its scalability. Therefore, the ability to define more instances of class `CheeseToping` is added to our DSL. The right side of `CheeseTopping` production is defined as a set of toppings:
`CheeseTopping ::= {Topping}`

Multi class generalization. It is common in the ontology that more than one class represents a similar domain concept, usually the only difference between these classes is in their derivation hierarchy. For example, in PO classes `ParmesanTopping`, `SalamiTopping`, `OnionTopping`, etc. they describe the same concept of topping, but they derive from different classes. In DSL grammar, this can be described with the same non-terminal. Grammar example:

```
CheeseTopping ::= Topping
MeatTopping ::= Topping
VegetableTopping ::= Topping
Topping ::= #string
```

Program fragment examples: `'parmesan'`, `'salami'` and `'onion'`.

Because of generalization we can get ambiguous grammar (one topping belongs to different abstract groups). To solve that problem, an enriching syntax can be used.

Enriching the syntax. One of the goals of DSL is to have a clear and easy to understand syntax with intuitive semantic. To achieve that, different patterns for enumerations, concepts that are seperated with brackets, adding reserved words, etc. have been used in O2DSL. For example in the `PizzaTopping` class reserved word `topping` is used. After all patterns are applyed, the following grammar is obtained:

```
PizzaTopping ::= topping
(CheeseTopping|MeatTopping|VegetableTopping)
CheeseTopping ::= cheese ToppingList
MeatTopping ::= meat ToppingList
VegetableTopping ::= vegetable ToppingList
ToppingList ::= '(' Topping {',' Topping} ')'
Topping ::= #string
```

Program fragment examples:

```
topping cheese ( 'parmesan' , 'mozzarella' )
topping meat ('salami')
```

In some cases, multi class generalization has an additional level in class hierarchy (additional abstract group between two classes). For example, `VegetableTopping` has an additional subclass `PepperTopping` that has subclasses `RedPepperTopping`, `GreenPepperTopping` and `JalapenoPepperTopping` (Fig. 2). One way to express an additional abstract group is to add a new alternative on `VegetableTopping` level with associated productions, other is by skipping this level and expect additional information in derivation of non-terminal `Topping`. Program fragment example:

```
topping vegetable ('red pepper', 'green pepper',
'jalapeno pepper')
```

Object properties and restriction transformations. Additional information about class relations can be obtained from ontology's object and class properties. For example `RedPepperTopping` has relations with `SpicinessValue-Partition` that has subclasses: `Hot`, `Mild` and `Medium`. In PO example, the relation is described by property `hasSpiciness Hot`. Class `PepperTopping` owns the property, therefore non-terminal `Topping` gets additional information. This property is not set for all `PizzaTopping` derivations and therefore it is optional. Following production is obtained:

```
Topping ::= #string [is SpicinessValuePartition]
```

Program fragment examples:

```
topping vegetable ('red pepper' is hot, 'green pepper' is
mild, 'jalapeno pepper' is medium, 'onion')
```

All class restrictions can be transformed in DSL by the similar transformation. In case that some restrictions are in addition defined by logical expression, support for logical expressions can also be added as part of DSL.

Obtained grammar. The appropriate order (sequence) of domain main concepts is defined from class hierarchy and class restrictions. For example, in case of PL, different instances of `PizzaTopping` must be defined before the definition of `NamedPizza`. Part of final PL grammar:

```
PL ::= {PizzaTopping} {Pizza} {Individual} {Query}
PizzaTopping ::= topping
(CheeseTopping|MeatTopping|VegetableTopping)
CheeseTopping ::= cheese ToppingList
MeatTopping ::= meat ToppingList
VegetableTopping ::= vegetable ToppingList
ToppingList ::= '(' Topping {',' Topping} ')'
Topping ::= #string [is SpicinessValuePartition]
SpicinessValuePartition ::= mild | hot | medium
Pizza ::= pizza (Interesting | Vegetarian | NonVegetarian
     | HighCalorie | LowCalorie | Named | Cheesy | Spicy)
…
```

Obtained DSL syntax is easy to understand and gives all flexibility and usability of DSLs. Obtained grammar and program fragments are used as a base for language development tool frameworks.

4 Conclusion and future work

In this paper, we have focused on the presentation of a new design methodology that enables the development of the language grammar, based on the OBDA. The limitations of the CDA have been examined and the replacement in the form of OBDA has been proposed. Both analysis have been presented and compared for similarities and differences. Grammar development, based on the OBDA, and the Ontology2DSL has also been briefly presented.

The results of the comparison between both analysis show that the OBDA is comparable to the CDA and also provides some additional information that can be used to specify language behavior. As such, it is also suitable as an alternative to CDA for grammar development. The framework Ontology2DSL is still under development. Currently, the framework supports the import of OWL ontology to an internal data structure and the transformation rules have been defined. The continuing development of the framework is a part of our future work. More specifically, we will focus on validation of the developed grammar and the use of previously unused information (i.e. for semantics development) that has been acquired with OBDA. The results of our research work will also be the transformation of the developed DSL to a form that is compatible with the compiler generators, such as LISA [16]. One of the future activities, to complete the methodology O2DSL, is evaluation of DSLs. As shown in the study [18], this activity is often underestimated by language developers. There is a plan to support this activity with tool based on questionnaire similar to [19] that will further improve the language.

References

1. Mernik, M., Heering, J., Sloane, A. M.: When and how to develop domain-specific languages. ACM Computing Surveys (CSUR) 37, 316--344 (2005)
2. Taylor, R. N., Tracz, W., Coglianese, L.: Software development using domain-specific software architectures. ACM SIGSOFT Software Engineering Notes 20, 27--38 (1995)
3. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA). Technical report, (1990)
4. Simos, M., Anthony, J.: Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domain Engineering. In: Proceedings of the 5th International Conference on Software Reuse, pp. 94--102. IEEE Computer Society, (1998)
5. Gruber, T.R.: A translation approach to portable ontology specification. Knowledge Acquisition 5, 199--220 (0993)
6. Tairas, R., Mernik, M., Gray, J.: Using Ontologies in the Domain Analysis of Domain-Specific Languages. In: Models in Software Engineering. LNCS, vol. 5421, pp. 332--342. Springer, (2009)
7. Ontology-Based Evaluation and design of domain-specific visual modeling languages, http://www.loa-cnr.it/Guizzardi/ISD2005.pdf
8. Guarino, N.: Semantic Matching: Formal ontological distinctions for information organization, extraction, and integration. In: Information Extraction A Multidisciplinary Approach to an Emerging Information Technology. LNCS, vol. 1299, pp. 139--170. Springer, (1997)
9. Walter, T., Parreiras, F. S., Staab, S.: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. In: Model Driven Engineering Languages and Systems. LNCS, vol. 5795, pp. 408--422. Springer, (2009)
10. Bräuer, M., Lochmann, H.: An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In: The Semantic Web: Research and Applications. LNCS, vol. 5021, pp. 34--48. Springer, (2008)
11. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. ACM Press/Addison-Wesley Publishing Co., (2000)
12. Van Deursen, A., Klint, P.: Domain-specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology 10, 1--17 (2002)
13. Corcho, Ó., Gómez-Pérez, A.: *A Roadmap to Ontology Specification Languages. In:* Knowledge Engineering and Knowledge Management Methods, Models, and Tools. LNCS, vol. 1937, pp. 80--96. Springer, (2000)
14. Antoniou, G., van Harmelen, F.: Handbook on Ontologies. Springer, Heidelberg (2009)
15. A Practical Guide to Building OWL Ontologies Using Protégé 4 and CO-ODE Tools, http://owl.cs.manchester.ac.uk/tutorials/protegeowltutorial/resources/ProtegeOWLTutorialP4_v1_2.pdf
16. Mernik, M., Lenič, M., Avdičauševič, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Horspool, N. (ed.) Compiler Construction. LNCS, vol. 2304, pp. 1-4. Springer, (2002)
17. Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, (2007)
18. Gabriel, P., Goulão, M., Amaral, V.: Do Software Languages Engineers Evaluate their Languages? In: Proceedings of the XIII Congreso Iberoamericano en "Software Engineering" (CIbSE'2010), pp. 149--162. CIbSE2010 (*Ecuador*), (2010)
19. Haugen, O., Mohagheghi, P.: A Multi-dimensional Framework for Characterizing Domain Specific Languages. In: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), Montréal, Canada, (2007)