DOI: 10.2298/CSIS101231019C

Ontology Driven Development of Domain-Specific Languages

Ines Čeh, Matej Črepinšek, Tomaž Kosar, and 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 (DSLs) are computer (programming, modeling, specification) languages devoted to solving problems in a specific domain. The development of a DSL includes the following phases: decision, analysis, design, implementation, testing, deployment, and maintenance. The least-known and least examined are analysis and design. Although various formal methodologies exist, 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 another form. A particularly suitable form is the use of ontologies. This paper focuses on ontology-based domain analysis and how it can be incorporated into the DSL design phase. We will present the preliminary results of the Ontology2DSL framework, which can be used to help transform ontology to a DSL grammar incorporating concepts from a domain.

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 (GPLs) and domain-specific languages (DSLs) [1], [2], [3], [4]. GPLs, 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 becomes 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 using a DSL are as follows: easier programming, re-use of

semantics, and the easier verification and programmability for end-users [1], [2]. However, using a DSL also has its disadvantages, such as high development costs [1], [5]. 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 a problem that belongs to a problem family and when 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, testing, deployment and maintenance [1], [6], which are discussed in greater detail in Section 2. While the implementation phase has attracted a lot of researchers [5], some of the DSL development phases are less known and are not as closely examined (e.g. analysis, design).

The knowledge of 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) [7], FODA (Feature-Oriented Domain Analysis) [8], and ODM (Organization Domain Modeling) [9]. 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 are not any clear guidelines on 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 be reflected in the DSL constructs, and commonalities could be incorporated into the DSL execution 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 the aforementioned problems, we propose that domain analysis (hereinafter referred to as classic domain analysis (CDA)) be performed with the use of existing techniques from other fields of computer science. A particularly suitable one is the use of ontologies [10], [11], [12]. An ontology provides the vocabulary of a specialized domain. This vocabulary represents the domain objects, concepts and other entities. Some types of domain knowledge can be obtained from the relationships of the entities, as presented by the vocabulary. Ontologies in the CDA have already been used in [13]. Whereas Tairas et al. apply ontologies in the early stages of domain analysis to identify domain concepts; we propose that an ontology replace the CDA. They also investigated how ontologies contribute to the design of the language [13]. Ontologies in connection with DSL are also used by other authors. Miksa et al. applied ontology-enabled software engineering in the area of DSL engineering [14]. Guizzardi et al. proposed the use of an upper ontology (top-level ontology) [15] to design and evaluate domain concepts [16]. Walter et al. applied ontologies to describe DSL [17]. Bräuer and

Lochmann proposed an upper ontology to describe interoperability among DSLs [18].

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 a DSL?« This paper focuses on ontology-based domain analysis (OBDA) and how it can be incorporated into the DSL design phase. We will present the preliminary results of the Ontology2DSL framework, which can be used to help transform an ontology to a DSL grammar.

The organization of this paper is as follows. Section 2 presents the background information required for the understanding of this paper. Section 3 is intended to demonstrate the similarities and variabilities between the CDA and OBDA. Section 4 presents the transformation rules used for the development of a DSL from an ontology, as well as the example of an ontology to a DSL transformation. Section 5 presents the framework Ontology2DSL and its architecture. The conclusion and future work are summarized in Section 6.

2. Background

2.1. DSL development phases

In [1], the authors have identified the following DSL development phases: decision, analysis (CDA), design, implementation, and deployment. The additional phases are testing and maintenance. The maintenance phase was introduced in [6]. Fig. 1 presents these phases along with the input and output of every phase and examples of patterns for the individual phases. The decision phase provides the answer to the question of when to develop a DSL. Other phases focus on the question of how to develop it. DSL and GPL development processes have a few differences with respect to the phases of development, since the phases are identical. The differences are in the activities, approaches and techniques used in the individual phases. The difference is expressed in the greater diversity of the activities, approaches and techniques in DSL development. It should be taken into consideration that DSL development is not a simple sequential process. Often, the phases overlap one another. For instance, the design of the DSL is influenced by the decision on the implementation approach. In the following section, the DSL development phases are briefly discussed.

Decision. It is often far from evident that a DSL might be useful or that developing a new one might be worthwhile. The concepts underlying a suitable DSL may emerge only after a lot of GPL programming has been

done. Decision patterns [1] describe situations (e.g., task automation, domain-specific Analysis, Verification, Optimization, Parallelization, and Transformation (AVOPT)) for which, in the past, developing a new DSL was fruitful.

Domain analysis (CDA). The precondition of the design and implementation of a DSL is a detailed domain analysis. 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 [19]. 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. Besides the development of the domain model, CDA also includes domain planning, identification and scoping. The inputs to the domain analysis are different sources of implicit and explicit domain knowledge. The information sources for the analysis are: technical literature, existing implementations, customer demands, expert advice, and current and future requirements [4]. An important note is the fact that the domain analysis process not only collects existing information. The systematic and organized collection of existing information enables and encourages the extension of information with new knowledge. In some cases, CDA can be informal, while in others it incorporates 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. The domain analysis can result in different DSLs. However, they all share essential information acquired in the domain analysis phase.

Design. Language design includes the definition of constructs and language semantics. The semantics formalize the meaning of every construct in the language and the behavior not specified in the program. The approaches to the design of a DSL can be classified into two orthogonal dimensions: the relation between DSL and a computer language and the formality degree of the DSL description [1]. The first dimension refers to the exploitation of an existing language (GPL or DSL) or the invention of a new language. The most basic method for DSL construction is if the DSL is based on an existing language. The existing language can be: partially reused (piggyback pattern), limited (language specialization pattern) or extended (language extension pattern) [1]. The advantages of building a DSL on an existing language are: easier implementation and the familiarity of the development environment to users who are experienced with the existing language. If the connection between the DSL and an existing language does not exist, a new language must be developed from the beginning. The second dimension refers to the informal and formal design of the language. With informal design, the specification is usually in the natural language with optional program examples. When the design is formal, the specification is usually in the form of a well-known formal definition method (BNF for syntax

specification, attribute grammars, denotational semantics, or algebraic specifications for semantic specification).

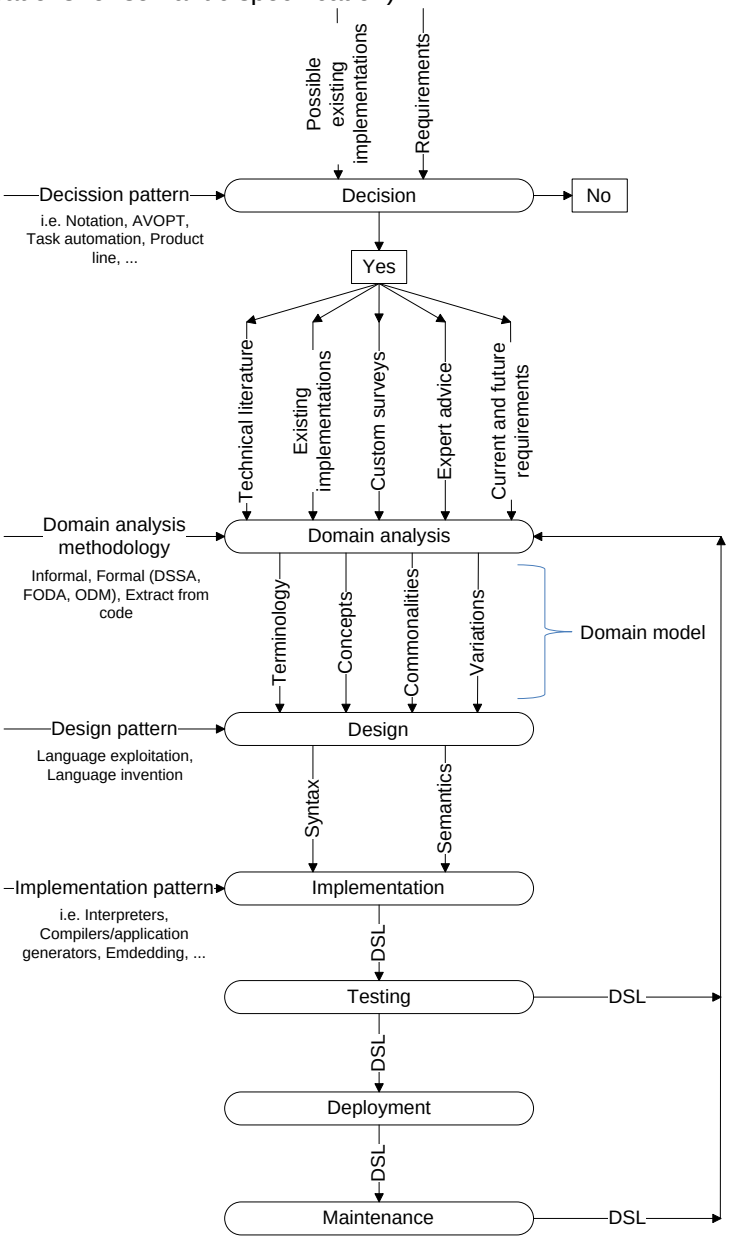

Fig. 1. DSL development phases

Implementation. Different approaches for DSL development can be used, such as: interpreter, compiler/application generator, embedding,

preprocessing, extensible compiler/interpreter, Commercial Off-The-Shelf (COTS), and hybrid. The approaches are presented in greater detail in [1]. Cleary we want to select the approach that requires the least effort during implementation and offers the greatest efficacy to the end user. The link between the implementation approaches, the effort of implementation and the efficacy to the end user is presented by the authors of [5].

Testing. In this phase, a DSL evaluation is performed. As shown in the study [20], this phase is often skipped or relaxed by language developers. The skipping or relaxing of this phase is not desirable because it may lead to the development of inadequate languages.

Deployment. In this phase, DSLs and applications constructed with them are used.

Maintenance. In this phase, the DSL is updated to reflect new requirements.

The number of phases and their individual complexity result in the discovery of high costs when developing a new DSL. DSL development requires domain knowledge and language expertise [1], [5]. These are the main reasons why DSLs are not so often used for solving software engineering problems. The development cost is seen as the greatest disadvantage of DSLs [5].

The main goal of the presented research is to investigate if the classic domain analysis (CDA) phase can be adequately replaced with an already existing domain knowledge and representation (e.g., ontology). In this manner the DSL development cost could be minimized.

2.2. Ontology

There are many definitions of ontologies in existing literature and one of the most commonly used definitions is that of Studer et al. They defined an ontology as follows: »An ontology is a formal, explicit specification of a shared conceptualization.« [12]. The meaning of the Studer et al. definition is detailed in [21]. 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, the 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 allow for the acquisition of knowledge about specific domains and often include rules that allow the processing of knowledge in existing ontologies. Ontology languages can be divided into two major groups: traditional (i.e. Flogic, Ontolingua) and web-based languages (i.e. RDF(S), OWL) [22]. Recently, a new group of languages, rule-based (i.e. RuleML, SWRL) [23], has emerged. These languages differ in their purpose and in their expressive power. The main requirements for an ontology language are:

a well-defined syntax, well-defined semantics, efficient reasoning support, sufficient expressive power and convenience of expression [21].

3. Comparison of CDA and OBDA

Subsection 3.1 presents the FODA methodology with which the domain analysis is performed. Subsection 3.2 introduces the ontology language OWL. The examples in both subsections are for the case of a home robot [24]. Subsection 3.3. compares the information obtained through FODA and through ontology domain analysis.

3.1. FODA

FODA is a CDA method that was developed by the Software Engineering Institute [19]. It is known for its models and feature modeling. In FODA, a 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. The 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 the links between them. The result of information analysis is the information model. The result of feature analysis is a feature model, which is presented below. An 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 the following:

- The Feature diagram (FD) represents a hierarchical decomposition of features and their kinds (mandatory, alternative, and optional feature). Mandatory features are those that each system must have in the domain. Alternative features are features that a system can only possess one at a time. Optional features are features that a 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 into 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. 2.
- Feature definitions describe all features (semantics).

- Composition rules for features describe which combinations are valid or invalid.
- Rationale for features represents the reasons for choosing a feature.

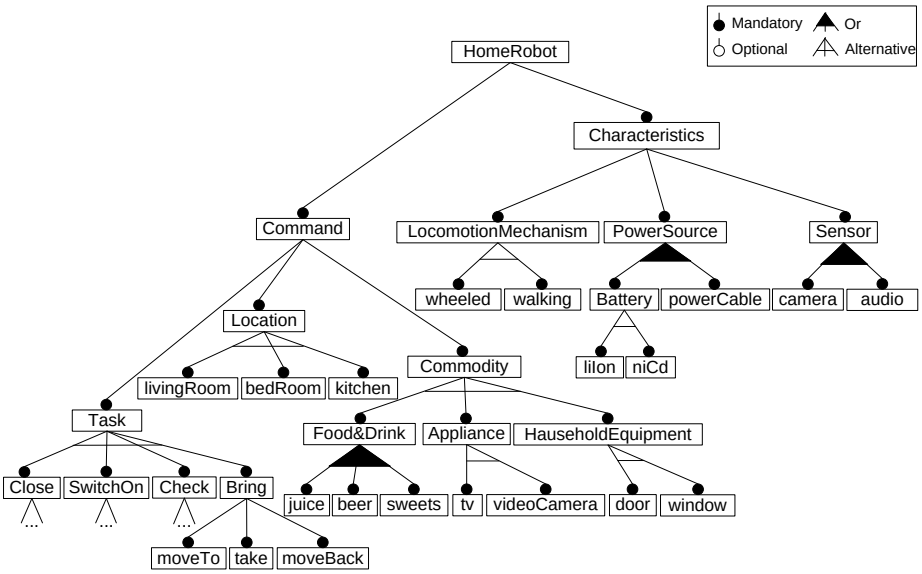

Fig. 2. Feature Diagram for a concept of a HomeRobot

Fig. 2 represents a simple FD of a HomeRobot. The root node of the diagram, HomeRobot, 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. Every house robot has individual characteristics and executes some commands. Every house robot has to have a PowerSource, uses some sensors for its locomotion and moves in a particular manner. Every robot can have multiple power sources but only one mechanism of motion. Every robot performs tasks, which are comprised of subtasks. The robot executes one order per location, every task is focused on an item, while some tasks focus on multiple items at the same time.

Feature models are not only represented in the visual form of FDs but also in textual form. Van Deursen and Klint have proposed the feature description language (FDL) for the textual representation. The FDL definition constitutes the feature definitions followed by a colon (":") and the features expression. Possible feature expression forms are presented in [25]. FDL exceeds the graphic feature diagram in terms of expressive power and is appropriate for automatic processing. The FD for a home robot in FDL is shown in Fig. 3.

```
HomeRobot: all ( Command, Characteristics )
Command: all ( Task, Location, Commodity )
Characteristics : all ( LocomotionMechanism,
PowerSource, Sensor )
Task: one-of ( Close, SwitchOn, Check, Bring )
Bring: all ( moveTo, take, moveBack )
Location: one-of ( livingRoom, bedRoom, kitchen )
Commodity: one-of ( Food&Drink, Appliance,
                    HauseholdEquipment )
Food&Drink: more-of ( juice, beer, sweets )
Appliance: one-of ( tv, videoCamera )
HauseholdEquipment : one-of ( door, window )
LocomotionMechanism: one-of ( wheeled, walking )
PowerSource: more-of ( Battery, powerCable )
Sensor : more-of ( camera, audio )
Battery: one-of ( liIon, niCd )
```

Fig. 3. FD for the home robot in FDL

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, as presented in [25].

Constraints, which are intended for variability reduction, are an optional component of the FDs. The constraints are enforced with satisfaction rules [25]. The constraints are of two types [25]: diagram constraints and user constraints. The former include the “A1 requires A2” (if the 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.

3.2. OWL

OWL is the most commonly used ontology language. It was created on the basis of RDFS [10], [11]. It has three sublanguages; OWL Full, OWL DL, and OWL Lite [10], [11], [21]. 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 the ontology. Class Nothing is empty and it is the subclass of every included class. The class hierarchy of the Home robot ontology (HRO) is represented by Fig. 4. HRO is based on [24]. The main functionalities of the robot are comprised of common household tasks, such as checking if the window is closed. HRO formalizes terms for three classes: locations, items and tasks. The locations are physical places where tasks are performed. Items are part of the tasks in the manner that the same action is performed on them or with them. The tasks are the actions being performed. Each task is comprised of subtasks. For the HRO annotation we used the OWL-DL, a sublanguage of the Ontology Web Language (OWL). The tool used for the creation of the ontology was Protégé [26], [27].

The second component, the properties, is a binary relation. OWL defines two main kinds of properties: object properties and datatype properties. 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 an 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).

3.3. Comparison

Both analysis incorporate a concept vocabulary, enable the display of property and class hierarchies, and provide a constraint mechanism (see Table 1). 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.

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	Yes	Yes (i.e. Protege)
Semantics	Yes	Yes
Query support	No	Yes (DL Query)

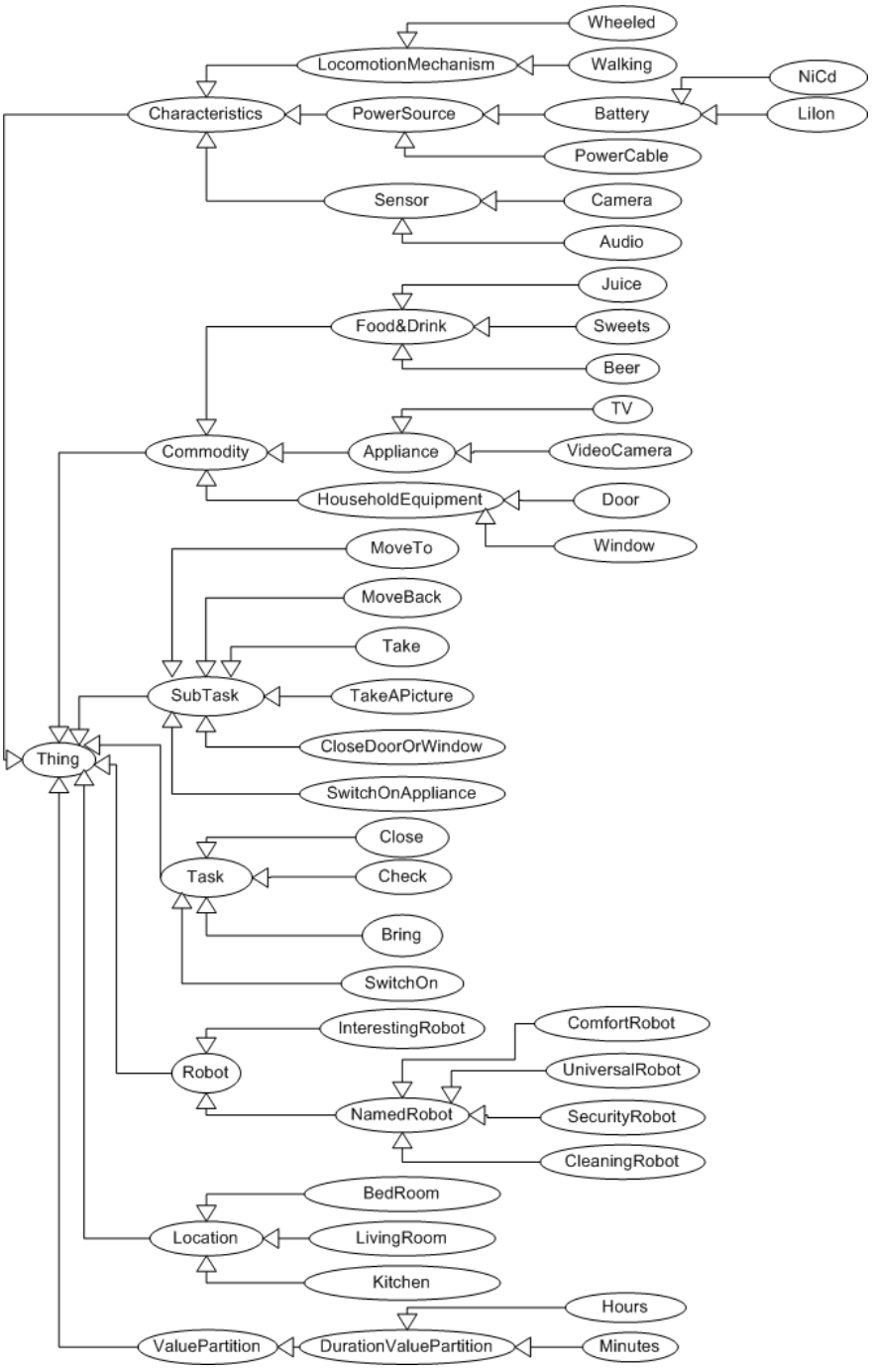

Fig. 4. Class hierarchy of a Home robot ontology

The CDA differs from OBDA in its capability to record the reasons for the use of a 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 of doing. The advantages of an ontology are reasoning and querying, because they enable the validation of an ontology. A valid ontology significantly reduces or prevents errors in DSL development. Semantics, which are inherently defined with the ontology, are 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 areas of research. This provides the method for elimination of the domain analysis phase in DSL development and might significantly reduce the time needed for language development.

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 also adds new capabilities.

4. Designing the DSL grammar

While the previous chapter shows that OBDA is appropriate for DSL development, this chapter demonstrates the process of grammar [28] construction from an OWL ontology. Section 4.1 introduces the rules used in the transformation from an ontology to its corresponding grammar. Section 4.2, however, is the application example which demonstrates the usage of the rules presented in 4.1.

4.1. Transformation rules

The input in the transformation is a data structure named the ontology data structure (ODS), which carries the data extracted from the OWL document. The result of the transformation is again a custom data structure, in this case the grammar data structure (GDS). GDS is a formal annotation of the resulting grammar.

With regard to the effect the transformation has on the ODS, the rules can be divided into the following two groups: (1) rules that do not affect the ODS and (2) rules that do affect and alter it. The results of the former can only be observed on the GSD, while the results of the latter are identifiable on both ODS and GDS. The rules that affect ODS can be used to alter the ontology.

The rules that do not alter the primary data structure:

1. $R_1(C) \rightarrow N_1 \in N$ (R - rule, C - class, N_1 - nonterminal, N - set of nonterminals). Rule R_1 is used to convert class C into nonterminal N_1. An application example for rule R_1 can be seen in chapter 3.2, step 1.

2. $R_2(C) \rightarrow T_1 \in T$ (T_1 - terminal, T - set of terminals). Rule R_2 is used to convert class C into terminal T_1. The user can, if necessary, change the name of terminal T_1. Name changes must be recorded in the dictionary.

3. $R_3(C) \rightarrow P \cup \{C ::= "C_N"\}$ (C_N - class name, P - set of productions, G - grammar ($G = < T, N, S, P >$)). Rule R_3 is used to add production into the grammar G. An application example for the rule R_3 can be seen in chapter 3.2, step 2.

4. $R_4(C_1, C_2) \rightarrow P \cup \{C_1 ::= C_2\}$. Rule R_4 is used to add production into the grammar G.

5. $R_5(C, CL_D, type) \rightarrow P \cup \{C ::= C_1 | C_2 | ... | C_n | type\}, C_1 ... C_n \in CL_D$ $if\ C_D \in L(CH) \rightarrow C_D \in T$ (L - list, CH - class hierarchy, CL_D - list of disjoint classes [17], C_D - disjoint class, $type$ - string, integer, …). The rule R_5 accepts the following inputs: class C, list of disjoint classes CL_D and $type$. The inputs CL_D and $type$ are optional. In any case, at least one of them must be present. If the inputs C and CL_D exist we are talking about the rule R_{5a}. If inputs C and $type$ exist we are talking about the rule R_{5b}. If all three inputs exist we are talking about the rule R_{5c}. Rule R_{5a} is used to add a production in the form $C ::= C_1 | C_2 | ... | C_n$ into grammar G. Rule R_{5c} is an extension of rule R_{5a} and is used to add productions in the form $C ::= C_1 | C_2 | ... | C_n | type$ into grammar G. A precondition for the successful transformation is that the children are disjointed; otherwise the resulting grammar is not a context free grammar. Classes from CL_D that

are also leafs of the class diagram, are transformed, with the use of rule R_3 to the set of terminals T. Rule R_{5b} is used to add productions in the form $C ::= type$ into grammar G. The rule enables grammar generalization, as described by class C, and the associated part of ontology. Each rule is used according to the bottom up principle; first on the lower of the class hierarchy levels, followed by the higher classes. An application example for the rules R_{5a} and R_{5c} can be seen in chapter 3.2, step 2.

6. $R_6(C, PL) \rightarrow \text{if } PL = \{\} | (\forall C \in PL) \cap LAC \rightarrow P \cup \{S ::= C\}$
(PL - parent list, S - start symbol, LAC - list of anonymous classes). Rule R_6 is used to define grammar start symbols. Class C is a possible grammar start symbol in the case that its parent list PL is empty or if all classes from PL are anonymous classes. Grammars can have more than one start symbol. An application example for the rule R_6 can be seen in chapter 3.2, step 5.

The rules that affect and alter the primary data structure:

7. $R_7(C, O, NC) \rightarrow P \cup \{C ::= typeTrans(NC)\}, O \in \{*, +, ?\}$ (O - operator). Rule R_7 is used to formalize the number of repetitions of some classes. The rule accepts the following inputs: class C, operator O, which defines the number of repetitions of some class and the new class NC. The rule is carried out in three steps. In the first step, the children of class C are assigned to class NC ($Child(NC) = Child(C)$). In the second step, the children of class C are removed ($RemoveChilds(C)$). In the third step the production is formalized. An application example for rule R_7 can be seen in chapter 3.2, step 6.

8. $R_8(C, T_L, T_R, NC) \rightarrow P \cup \{C ::= T_L\ Nc\ T_R\}$ (T_L - left terminal, T_D - right terminal). Rule R_8 is used to enrich the syntax. Either the left or the right terminal can be omitted. The rule is carried out in three steps. In the first step, the children of class C are assigned to the class NC ($Child(NC) = Child(C)$). In the second step, the children of class C

are removed ($RemoveChilds(C)$). In the third step, the production is formalized. An application example for rule R_8 can be seen in chapter 3.2, step 7.

9. $R_{9some}(C_1, C_2, T_M, NC) \rightarrow P \cup \{C_1 ::= NC\ T_M\ C_2\}$ (T_M - middle terminal). Rule R_{9some} is used to formalize productions which describe restrictions (some). The rule accepts the following inputs: class C_1 to which the restriction refers, class C_2 which determines the possible values of class C_1, the middle terminal T_M and the new class NC. The rule is carried out in three steps. In the first step, the children of class C_1 are assigned to class NC ($Child(NC) = Child(C)$). In the second step the children of class C_1 are removed ($RemoveChilds(C_1)$). In the third step the production is formalized.

10. $R_{10}(C, LofCs) \rightarrow P \cup \{C ::= C_1\ C_2 ... C_n\}, C_1\ C_2 ... C_n \in LofCs;$
$if\ C_i \in L(LofCs) \rightarrow C_i \in T$ ($LofCs$ - list of classes)
The rule R_{10} accepts the following inputs: class C and a list of classes $LofCs$. Rule R_{10} is used to add a production in the form $C ::= C_1\ C_2\ ...\ C_n$ into the grammar G. Classes from $LofCs$ that are also leafs of the class diagram, are transformed, with the use of rule R_3 to the set of terminals T. The rule is used on the first level. The class Thing is ignored in the transformation. An application example for rule R_{10} can be seen in chapter 3.2, step 4.

This chapter lists some of the rules necessary for the transformation of an onotology into a DSL grammar.

4.2. Ontology to DSL transformation: Home robot example

The prerequisite of the Ontology to DSL transformation (Ontology2DSL) is a proper understanding of the target ontology. The language designer must understand what the ontology describes and why it was designed. Moreover, the language designer needs to know what the DSL requirements are, and what the purpose of the DSL is. In most cases, the DSL requirements and the ontology do not overlap in all concepts. A single ontology, for instance the

HRO, can be used to develop many different DSLs. We continue with the examination of the DSL used for the home robot. The robot is tasked with performing various chores on different locations in the household.

The transformation requires a list of ontology classes and a collection of individually disjoint classes. All the required data was obtained from the OWL document. During the transformation, we also relied on the class hierarchy presented in Fig. 4.

Classes. `Commodity, Food&Drink, Juice, Sweets, Beer, Appliance, TV, VideoCamera, HauseholdEquipment, Door, Window, Task, Close, Bring, Check, SwitchOn, Robot, Location, BedRoom, LivingRoom, Kitchen.`
(Class Thing and other classes from Fig. 4, which are not mentioned in the above list, are ignored in the transformation.)
Disjoint classes.

- `Food&Drink, Appliance and HauseholdEquipment`
- `Juice, Sweets and Beer`
- `TV and VideoCamera`
- `Door and Window`
- `Close, Check, Bring and SwitchOn`
- `Robot, Commodity, Location and Task`
- `BedRoom, LivingRoom and Kitchen`

Step 1. In the first step, all classes are converted into nonterminals.

```
R1(Robot)
N = {Robot}
```

Rule R_1 in this step is used on all the classes and results in the following set of nonterminals N.

```
N = {Commodity, Food&Drink, Juice, Sweets, Beer,
     Appliance, TV, VideoCamera, HauseholdEquipment,
     Door, Window, Task, Close, Bring, Check, SwitchOn,
     Robot, Location, BedRoom, LivingRoom, Kitchen}
```

Step 2. The transformation is continued on the lowest, third, level. It is performed with the rules R_3, R_{5a}, and R_{5c}.

```
R5c(Food&Drink, {Juice, Sweets, Beer}, string)
R3(Juice)
R3(Sweets)
R3(Beer)
R5a(Appliance {TV, VideoCamera})
R3(TV)
R3(VideoCamera)
R5a(HauseholdEquipment {Door, Window})
R3(Door)
R3(Window)
```

```
T = {»juice«, »sweets«, »beer«, »TV«, »videoCamera«,
     »door«, »window«}
P = { Food&Drink ::= Juice | Sweets | Beer | string
      Juice ::= »juice«
      Sweets ::= »sweets«
      Beer ::= »beer«
      Appliance ::= TV | VideoCamera
      TV ::= »TV«
      VideoCamera ::= »videoCamera«
      HauseholdEquipment ::= Door | Window
      Door ::= »door«
      Window ::= »window«}
```

Step 3. The transformation is continued on the second level. The rules used are R_3 and R_{5a}.

```
R5a(Commodity, {Food&Drink, Appliance,
    HouseholdEquipment})
R3(Food&Drink)
R3(Appliance)
R3(HouseholdEquipment)
R5a(Location, {BedRoom, LivingRoom, Kitchen})
R3(BedRoom)
R3(LivingRoom)
R3(Kitchen)
R5a(Task, {Close, Check, Bring, SwitchOn})
R3(Close)
R3(Check)
R3(Bring)
R3(SwitchOn)
```

```
T = {…, »food&Drink«, »appliance«, »householdEquipment«,
     »bedRoom«, »livingRoom«, »kitchen«, »close«,
     »check«, »bring«, »switchOn«}
P = { …, Commodity ::= Food&Drink | Appliance |
                       HouseholdEquipment
      Food&Drink ::= »food&Drink«
      Appliance ::= »appliance«
      HouseholdEquipment ::= »householdEquipment«
      Location ::= BedRoom | LivingRoom | Kitchen
      BedRoom ::= »bedRoom«
      LivingRoom ::= »livingRoom«
      Kitchen ::= »kitchen«
      Task ::= Close | Bring | Check | SwitchOn
      Close ::= »close«
      Check ::= »check«
      Bring ::= »bring«
      SwitchOn ::= »switchOn«}
```

Step 4. The first level is transformed with the R_{10} rule.

```
R10(Robot, {Task, Commodity, Location})
T = {…, »Task«, »Commodity«, »Location«}
P = {…, Robot ::= Task Commodity Location}
```

Step 5. In the next step, all possible grammar start symbols are extracted.

```
R6(Commodity, {})
R6(Task, {})
R6(Robot, {})
R6(Location, {})
R5a(S, {Commodity, Task, Robot, Location})
S = {Commodity, Task, Robot, Location }
P = {…, S ::= Commodity | Task | Robot | Location}
```

Step 6. In the next step, rule R_7 is used. Strikethrough production is eliminated from the set of production.

```
R7(Commodity, +, Commodities)
P = { …, ~~Commodity :: = Food&Drink | Appliance |~~
                          ~~HauseholdEquipment~~
      Commodity::= Commodities+
      Commodities ::= Food&Drink | Appliance |
      HouseholdEquipment}
```

Step 7. In the last step the syntax is enriched. Strikethrough productions are eliminated from the set of production.

```
R8(Location, {»from«| »in«}, {}, LocationE)
LocationE:: = {»from«| »in«} Location
P = { …, ~~S ::= Robot | Commodity | Location | Task~~
      ~~Robot ::= Task Commodity Location~~
      S ::= Robot | Commodity | LocationE | Task
      Robot ::= Task Commodity LocationE}
```

Obtained grammar:

```
P = { Robot ::= Task Commodity LocationE

      Task ::= Close | Check | Bring | SwitchOn
      Close ::= »close«
      Check ::= »check«
      Bring ::= »bring«
      SwitchOn ::= »switchOn«

      Commodity::= Commodities+
      Commodities ::= Food&Drink | Appliance |
                      HouseholdEquipment
```

```
Food&Drink ::= Juice | Sweets | Beer | string
Food&Drink ::= »food&Drink«
Juice ::= »juice«
Sweets ::= »sweets«
Beer ::= »beer«
Appliance ::= TV | VideoCamera
Appliance ::= »appliance«
TV ::= »TV«
VideoCamera ::= »videoCamera«
HouseholdEquipment ::= Door| Window
HouseholdEquipment ::= »householdEquipment«
Door ::= »door«
Window ::= »window«

Location ::= BedRoom | LivingRoom | Kitchen
BedRoom ::= »bedRoom«
LivingRoom ::= »livingRoom«
Kitchen ::= »kitchen«
LocationE :: = {»from«| »in«} Location}
```

Program examples:

```
close door in bedRoom
check window in kitchen
switchOn TV in livingRoom
bring beer chips from kitchen
```

5. Ontology2DSL

The Ontology2DSL framework enables automated grammar construction as well as one or more programs from a target ontology. The framework accepts an OWL document as an input, parses it and uses the information retrieved to create and fill internal data structures. Then a transformation pattern, annotated with the proper rule execution order, is applied over the data structures and the corresponding grammar and programs are constructed. The resulting grammar, acquired fully automatically, is then inspected by a DSL engineer in order to verify it and find any irregularities. If any irregularities are found, they are tasked with their resolution with regard to the source and type. The engineer can either correct the constructed grammar, programs or the transformation pattern (i.e. change the order in which the rules are applied or construct new rules and include them in the pattern). The framework then rebuilds the grammar and programs as required. The rebuild process can utilize a new transformation pattern on an old ontology, an old pattern on a new ontology, or a new pattern on a new ontology. The process is repeated until the DSL engineer can no longer find any irregularities. The framework also has the option of constructing in sequential steps instead of the fully automated method. In that case, the engineer can execute each rule

individually and can, at any time, return to a previous step if the result proves to be unsatisfactory. This method allows for complete control over the grammar and the resulting program's construction process. The final (correct) grammar can later be used by the DSL engineer for the development of DSL tools. The latter are developed with the use of language development tools, such as LISA [29] or VisualLISA [30]. The development of DSL tools from an ontology is a process demonstrated in the workflow of Fig. 5.

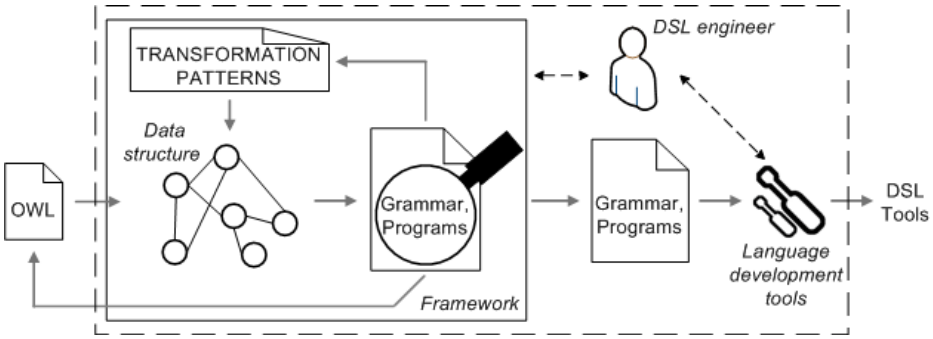

Fig. 5. Ontology2DSL workflow

5.1. Architecture of the framework

The architecture of the Ontology2DSL framework, shown in Fig. 6 is comprised of the following:

- **OWL parser.** The parser is tasked with the parsing of OWL documents and the filling of the data structure with the retrieved data. The data structure is composed of the following individual data structure types: a class tree, an object properties tree, a datatype properties tree, a list of anonymous classes, a list of disjoint classes, a list of instances and a list of ID's of all the ontology building blocks in the aforementioned lists. Part of the data structure for the HRO is presented in Fig. 7. The building of hierarchy objects (trees) and lists is done with a sequential scan of the OWL document. Each retrieved element is added to an appropriate list and is assigned all the necessary information. A check is also performed to determine if the new element possesses any new information that should be assigned to other elements. In instances where the new element has some information that is important for the elements that have not yet been added to any of the trees or lists, that information is cached until the required elements are not added to the data structure.

- **Rule reader.** The reader is tasked with the sequential read operations on the rules list. The reader forwards each rule to the rule execution and transaction logger components.

- **Rule execution component (REC)** is used for the execution of individual rules. The necessary data for the execution is retrieved automatically by REC from the data structure. After a rule is executed, REC refreshes the data structure if the rule execution result requires it. Also, the set of grammar elements are refreshed and parts of the code are written out. The element set of the grammar in the final result becomes the final grammar and the code parts become the programs that represent the final output of the Ontology2DSL framework.

- **Transaction logger.** After the execution of every rule, the system's current state is logged by the transaction logger. The logger stores the entire content of the data structure, the last executed rule, the output of the rule execution component and the current grammar and program parts.

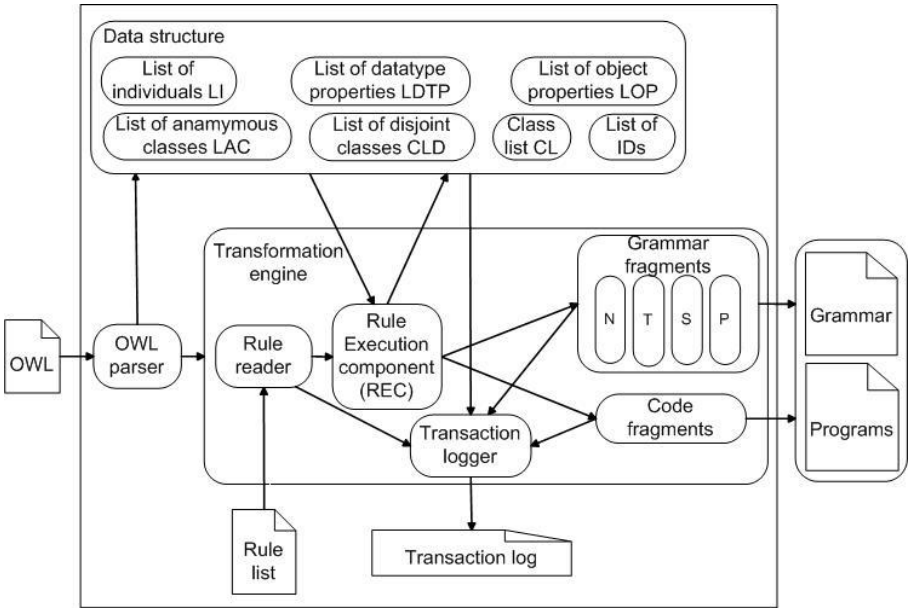

Fig. 6. Architecture of the framework

```
Class tree
Class: Close
Equivalent classes: /
Superclasses: Task, hasDuration some Minutes (AnonymousClass1)
Infered anonymous classes: /
Members: /
Disjoint classes: Bring, Check, SwitchOn, Wash

Object properties tree
Object property: hasDuration
Characteristics: FunctionalProperty
Domains: /
Ranges: DurationValuePartition
Equivalent object properties: /
Super properties: /
Inverse properties: /
Disjoint properties: /
Properties chain: /

List of anonymous class
Anonymous class: AnonymousClass1
Class: Close
Property: hasDuration
ValueType: someValueFrom
    Value: Minutes

List of disjoint classes
Disjoint class collection: Collection1
Superclass: Location
Disjoint classes: BedRoom, Kitchen, LivingRoom

List of ID's
Close (Class), hasDuration (Object property),
AnonymousClass1 (AnonymousClass), Collection1 (Collection) ...
```

Fig. 7. An excerpt of the data structure for HRO

6. Conclusion and future work

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

The results of the comparison between both analyses 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. The current version is composed of all of the basic components: an OWL parser, a rule reader, REC and a transaction logger. As

opposed to other components that are fully developed, REC is not fully developed, as it does not yet construct code fragments. The framework in the current development phase can only be used to construct grammar. Additionally, in the current version, a DSL engineer cannot add custom rules and create custom transformation patterns. In the future, we intend to fully develop the Ontology2DSL framework. We will also focus on validating the developed grammar and the use of previously unused information (i.e. for semantics development) that was acquired with an OBDA. The results of our research work will also include the transformation of the developed DSL to a form that is compatible with compiler generators, such as LISA [29] or VisualLISA [30]. Our future work also encompasses empirical studies to evaluate the success of our methodology and to compare it with the existing methodologies. One of our future activities, to complete the methodology Ontology2DSL, will be an evaluation of DSLs. As shown in study [20], this activity is often underestimated by language developers. There is a plan to support this activity with a tool based on a questionnaire similar to [31] which 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, Vol. 37, No. 4, 316-344. (2005)
2. Kosar, T., Oliveira, N., Mernik, M., Veranda Pereira, M. J., Črepinšek, M., da Cruz, D., Henriques, P. R.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
3. Thibault, S., Marlet, R., Consel, C.: Domain-Specific Languages: From Design to Omplementation Application to Video Device Drivers Generation. Conception, Implementation and Application. IEEE Transactions on Software Engineering, Vol. 25, No. 3, 363-377. (1999)
4. Thibault, S.: Domain-Specific Languages: Conception, Implementation and Application. Phd thesis. Université de Rennes, France. (1998)
5. Kosar T., Martínez López P.E., Barrientos P.A., Mernik M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology, Vol. 50, No. 5, 390-405. (2008)
6. Mernik, M., Hrnčič, D., Bryant, B. R., Javed, F.: Applications of grammatical inference in software engineering : domain specific language development. In: Martin-Vide, C. (ed.): Scientific applications of language methods, Vol. 2. Imperial College Press, London, 421-457. (2011)
7. Taylor, R. N., Tracz, W., Coglianese, L.: Software development using domain-specific software architectures. ACM SIGSOFT Software Engineering Notes, Vol. 20, No. 5, 27-38. (1995)
8. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA). Technical report. (1990)
9. Simons, 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. IEEE Computer Society, Victoria, BC, Canada, 94-102. (1998)

10. Lacy, L.: Representing Information Using the Web Ontology Language. Trafford Publishing. (2005)
11. Hebeler, J., Fisher, M., Blace, R., Perez-Lopez, A.: A Semantic Web Programming. Wiley Publishing. (2009)
12. Studer, R., Benjamins, R., Fensel, D.: Knowledge engineering: Principles and methods. Data & Knowledge engineering, Vol. 25, No. 1-2, 161-198. (1998)
13. Tairas, R., Mernik, M., Gray, J.: Using Ontologies in the Domain Analysis of Domain-Specific Languages. In: Chaudron, M. R. V (ed.): Models in Software Engineering. Lecture Notes in Computer Science, Vol. 5421. Springer-Verlag, Berlin Heidelberg New York, 332-342. (2009)
14. Miksa. K., Sabina, P., Kasztelnik, M.: Combining Ontologies with Domain Specific Languages: A Case Study from Network Configuration Software. In: Assmann, U., Bartho, A., Wende, C. (eds.): Reasoning Web. Semantics technologies for software engineering, Vol. 6325. Springer-Verlag, Berlin Heidelberg New York, 99-118. (2010)
15. Guarino, N.: Semantic Matching: Formal ontological distinctions for information organization, extraction, and integration. In: Pazienza, M. T.: Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology. Lecture Notes in Computer Science, Vol. 1299. Springer-Verlag, Berlin Heidelberg New York, 139-170. (1997)
16. Ontology-Based Evaluation and design of domain-specific visual modeling languages, http://www.loa-cnr.it/Guizzardi/ISD2005.pdf.
17. Walter, T., Parreiras, F. S., Staab, S.: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. In: Schürr, A., Selic, B. (eds.): Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, Vol. 5795. Springer-Verlag, Berlin Heidelberg New York, 408-422. (2009)
18. Bräuer, M., Lochmann, H.: An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In: Beckhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.): The Semantic Web: Research and Applications. Lecture Notes in Computer Science, Vol. 5021. Springer-Verlag, Berlin Heidelberg New York, 34-48. (2008)
19. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. ACM Press/Addison-Wesley Publishing Co. (2000)
20. 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). Cuenca, Ecuador, 149-162. (2010)
21. Stabb, S., Studer, R., editors. Handbook on Ontologies. Springer Verlag Berlin Heidelberg. (2009)
22. Corcho, Ó., Gómez-Pérez, A.: A Roadmap to Ontology Specification Languages. In: Dieng, R., Corby, O.: Knowledge Engineering and Knowledge Management. Lecture Notes in Computer Science, Vol. 1937. Springer-Verlag, Berlin Heidelberg New York, 80-96. (2000)
23. Milanović, M., Gašević, D., Giurca, A., Wagner, G., Lukichev, S., Devedžić, V.: Model Transformations to Bridge Concrete and Abstract Syntax of Web Rule Languages. Computer Science and Information Systems, Vol. 6, No. 2, 47-85. (2009)
24. Cho, K., Kawamura, T.: Blogalpha: Home automation robot using ontology in home environment. In Proceedings of the 25th International Multi-Conference Artificial Intelligence and Applications. ACTA Press Anaheim, CA, USA, 197-203. (2007)

25. Van Deursen, A., Klint, P.: Domain-specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology, Vol. 10, No. 1, 1-17. (2002)
26. Welcome to Protégé. [Online]. Available: http://protege.stanford.edu/ (current April 2011)
27. Jung, H., Park, S.: A Grammar-based Model for the Semantic Web. Computer Science and Information Systems, Vol. 8, No. 1, 73-100. (2011)
28. Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, USA. (2007)
29. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Horspool, R. N. (ed.): Compiler Construction. Lecture Notes in Computer Science, Vol. 2304. Springer-Verlag, Berlin Heidelberg New York, 1-4. (2002)
30. Oliveira, N., Veranda Pereira, M. J., Henriques, P. R., da Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
31. 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)

Ines Čeh received the B.Sc. degree in computer science at the University of Maribor, Slovenia in 2008. Her research interests include domain-specific languages and ontologies. She is currently a Ph.D student, employed as a researcher at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Matej Črepinšek received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research interests include grammatical inference, evolutionary computations, object-oriented programming, compilers, grammar-based systems and Android application development. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific modelling languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998, respectively. He is currently Professor of Computer Science at the University of Maribor. He is also Visiting Professor of Computer and Information Sciences at the University of Alabama at Birmingham, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modeling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Received: December 31, 2010; Accepted: May 13, 2011.