Tool-Supported Building of DSLs from OWL Ontologies

Ines Čeh[1], Matej Črepinšek[1], Tomaž Kosar[1], Marjan Mernik[1], Pedro Rangel Henriques[2], Maria João Veranda Pereira[3], Daniela da Cruz[2], Nuno Oliveira[2]

[1] University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova 17, 2000 Maribor, Slovenia
{ines.ceh, matej.crepinsek, tomaz.kosar, marjan.mernik}@uni-mb.si

[2] University of Minho – Department of Computer Science, Campus de Gualtar, 4715-057, Braga, Portugal
{prh, danieladacruz, nunooliveira}@di.uminho.pt

[3] Polytechnic Institute of Bragança, Campus de Sta. Apolónia, Apartado 134 - 5301-857, Bragança, Portugal
mjoao@ipb.pt

Abstract. Domain-specific languages (DSLs) are computer languages intended for problem solving in a specific domain. Ontology is a formal representation of a set of concepts from a particular domain and the relations between them. An ontology may be used to describe a domain and to reason about the entities within the domain. This paper presents an Ontology2DSL framework to build DSLs from OWL ontologies. Ontology2DSL enables the semi-automated construction of a formal grammar and programs from an OWL ontology. The design approach, the functionalities of the framework, and a case study are also addressed in this paper. Special attention is paid to the architecture that encompasses the following components: the transformation pattern builder, the OWL parser, the rule reader, the rule execution component and the transaction logger.

Keywords: Domain-Specific Language, Ontology, OWL.

1 Introduction

Domain-specific languages (DSLs), such as HTML and SQL, are computer languages devoted to solving problems in a specific domain [1], [2]. DSL development is comprised of the following phases: decision, analysis, design, implementation, testing, deployment, and maintenance [1], [3], [4].

When compared to the efforts aimed at the implementation phase [5] relatively little attention is being paid to the analysis and design phases. There are some methodologies for domain analysis. Some examples of such methodologies include:

DSSA, FAST, FODA etc. Many of them have proven to be too complex and subsequently the domain analysis is often performed informally. Even when formal methodologies are used, questions on how to proceed with the development of DSL are still raised, mainly because no clear guidelines exist on how to use the information gathered in the analysis phase of the design phase.

Instead of developing a less complex domain analysis methodology, we propose that domain analysis be performed with the reuse of existing approaches in the computer science field. A fitting approach involves ontologies [6], [7], [8], [9]. We therefore propose replacing the domain analysis with an ontology domain analysis. This means that a DSL would be developed on the basis of domain information from an existing ontology. A preexisting ontology, which contains all the domain information required for DSL development, renders the phase of domain analysis unnecessary. This results in major savings in both time and money in the development process of a DSL. These savings are seen as the greatest advantage of the approach we will be presenting. The disadvantage lies in the fact that, in the majority of cases, existing ontologies do not fully match and describe every concept.

A comparison between domain analysis and an ontology domain analysis as well as the reasons for why the ontology domain analysis is a fitting approach and what the rules for executing the transformation of an ontology to the language grammar, are shown in [4]. This paper focuses on the presentation of the Ontology2DSL framework that constructs the language grammar [10] and programs, from an OWL ontology [8], [9].

DSL development with the presented framework is easier and thus cheaper because the framework is able to execute a large part of the transformation independently of the DSL engineer. The involvement of the engineer is required in the steps where the framework has to "understand" the meanings of the concepts for which the work is being done. Another advantage of the framework is the ability to quickly test and verify different solutions; developing different grammars. Of course, the framework does require some knowledge before it can be effectively used. Familiarization with ontologies ensures a much easier understanding of the framework.

The developed framework is appropriate for use in education as well as industry. Students will find it particularly useful when they study the construction of grammars, as the framework autonomously accomplishes several steps and leads them to the correct path. Industrial use would be the primary goal, as it would be leveraged to speed up the process of DSL development as well as lowering the costs.

Ontologies have been used by other authors in the DSL field [11], [12], [13], [14], [15], [16]. A survey of literature has not given any reference where research was aimed at the development of DSLs from ontologies. Also, our framework cannot be compared to various tools for DSL creation (e.g., EMFText, Xtext, MetaEdit+, GME), where DSL is created from a language model (meta-model). All these tools require that domain concepts and relationships among them are already known. Hence, these tools do not support a domain analysis phase, which is usually done *ad hoc*.

The organization of the paper is as follows: Section 2 presents the ontology web language OWL. Section 3 introduces the framework, while Section 4 examines the architecture in greater detail. Section 5 shows a case study. The conclusion and ideas for future work are summarized in Section 6.

2 The web ontology language OWL

In existing literature, there are many definitions of ontology. One of the most widely used definitions of ontology is by Studer et al. In 1998, they defined ontology as follows: "An ontology is a formal, explicit specification of a shared conceptualization." [6]. The meaning of the Studer et al. definition is detailed in [7].

To work with an ontology we need ontology languages. Ontology languages allow for the acquirement of knowledge about specific domains and often include rules that allow the processing of knowledge in existing ontologies.

The web-based ontology language, OWL, was created on the basis of RDFS [8], [9]. The officially recommended exchange syntax for OWL is the RDF/XML exchange syntax, which provides an XML representation of an RDF graph [17]. This syntax is very verbose and can be difficult to read. Consequently, it is also non-trivial to parse OWL ontologies represented in RDF.

The three basic components of OWL, which will henceforth simply be referred to as OWL, are: classes, properties, and individuals [8], [9]. OWL classes, which define basic concepts, may be organized into a hierarchy. OWL defines two kinds of classes: simple named and predefined (the "Thing" and "Nothing"). Simple named classes are classes that are defined by users. Predefined classes are classes that are provided by OWL. The second component, the properties, is a binary relation, which associates values with individuals. The two main kinds of properties in OWL are object properties and datatype properties. Object properties relate objects to other objects. Datatype properties relate objects to datatype values. The third component, the individuals, are members of the user-defined ontology classes.

3 Ontology2DSL framework

The Ontology2DSL framework enables the semi-automated construction of a formal grammar and programs from an OWL ontology.

Ontology2DSL accepts an OWL document as its input. The document is parsed and the resulting information is used to fill the internal data structure. The internal data structure is called an ontology data structure (ODS). A transformation pattern is then applied over the ODS. The transformation pattern is a sequence of transformation rules. An individual transformation pattern contains rules from the framework's set of supported rules. The resulting output of the Ontology2DSL includes the language grammar and programs. The workflow diagram of the Ontology2DSL framework is presented in Fig. 1.

Before further use, such as DSL development, the resulting grammar is inspected by a DSL engineer in order to verify it and find any irregularities. All the irregularities that are discovered are resolved with regard to their type. The engineer can either correct the constructed language grammar or change the transformation pattern or the source ontology. If the change was done on the ontology or transformation pattern, then a new transformation run is required. The framework can then use the old transformation pattern on the new ontology, the new transformation pattern on the old ontology, or the new transformation pattern on the new ontology.

Whereas the changes to the transformation pattern are supported within the framework, the DSL engineer can edit the source ontology with the use of a preexisting tool, such as Protégé [18].

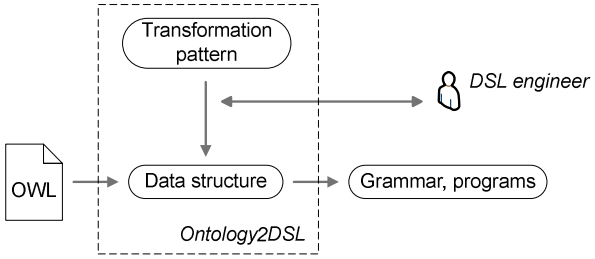

Fig. 1. Ontology2DSL framework workflow diagram.

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 [20]). The development of the DSL tools process is demonstrated in the workflow shown in Fig. 2.

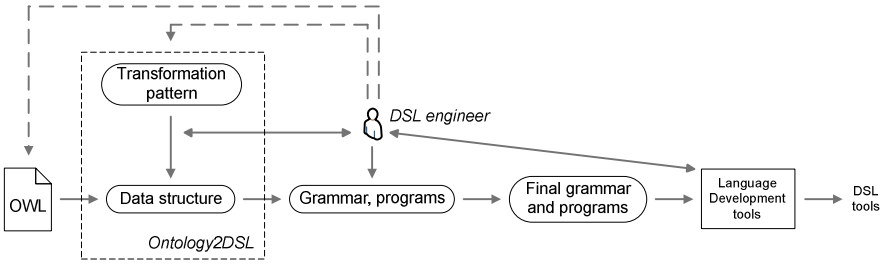

Fig. 2. The DSL tools development process.

4 Architecture of the framework

The cornerstone of the Ontology2DSL's framework is the transformation engine (TE). TE sequentially reads the rules from the transformation pattern and executes them. The individual executions form parts of the grammar fragments. The TE logs the changes and the system state after every rule execution. Rule execution and logging is performed through the following components of the TE: the rule reader, the rule execution component, and the transaction logger. The prerequisite for the proper working of the transformation engine are these components: the transformation pattern builder and the OWL parser. All of these components are shown in Fig. 3 and discussed in greater detail in subsections 4.1-4.5.

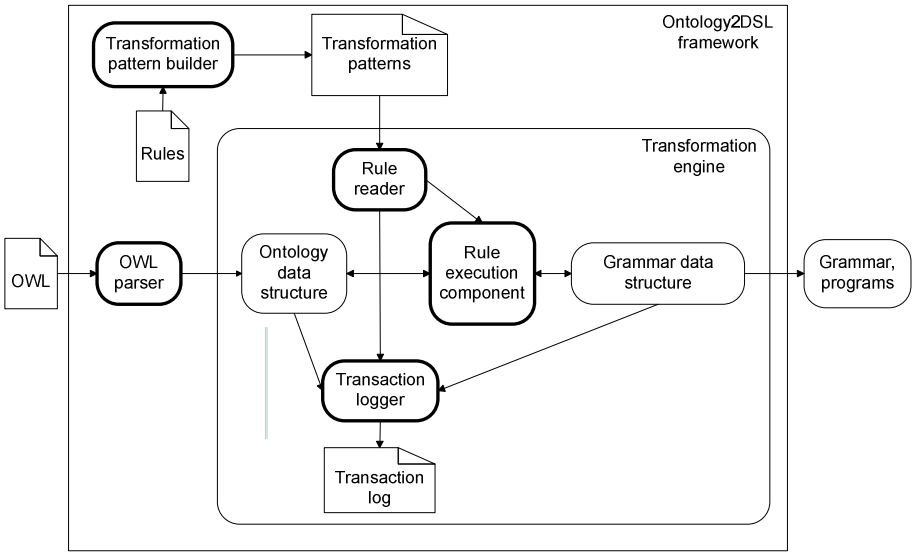

Fig. 3. Architecture of the Ontology2DSL framework.

4.1 The transformation pattern builder

The transformation pattern builder (TPB) is the component that the DSL engineer uses to create transformation patterns. The transformation pattern (TP) is a sequence of transformation rules. Every rule used in the process of constructing grammar and programs from ontology, regardless of whether the rule actually executes a transformation or not, is called a transformation rule.

The following transformation rules have been proposed [4]: the rules for the search for grammar start symbols, the rules for the transformation of classes to terminals or non-terminals, the rules for the creation of productions, the rules for syntax enrichment, etc. Transformation rules and use cases are presented in more detail in [4]. The rules can be divided with regard to the scope of their effect: some rules perform a particular transformation while others perform a particular transformation type on the entire ontology. An example of the former type is the "Possible start symbol rule," which checks if a class is a possible grammar start symbol. An example of the latter type is the "Find start symbols rule," which is used to check if every class in the ontology is a start symbol. The individual check of every class is performed by the "Possible start symbol rule." Presently the framework supports a set of rules which are common enough that they can be used for the majority of transformations.

In addition to the creation of new patterns, TPB also provides the capability to edit and remove existing patterns. Individual patterns contain rules supported by the framework. All supported rules are stored in an XML file. Each rule is associated with its name and the name of the executing function. An excerpt from the XML file with the rules is presented in Fig. 4a. As with the rules, the patterns are also stored in

an XML file. An example of the “Create start production pattern” pattern, which enables the construction of the target DSL’s start production, is presented in Fig. 4b. A pattern contains rules in the correct sequence of an application. When creating or editing a pattern, the rule priority must be observed. For the successful completion of the transformation some rules must be applied before others. For example, the pattern “Create start production pattern” should have the rule “Find all start symbols” before the rule “Create start production” since the start production is created from start symbols. The verification of the dependency between rules and the sequence is done automatically by the TPB, and the DSL engineer is notified if any irregularities are found.

```
<?xml version="1.0" encoding="us-ascii"?>
<Rules>
    <Rule name="Find all start symbols" correspondingFunction="Rule1"/>
    <Rule name="Create start production" correspondingFunction="Rule2"/>
</Rules>
```

a)

```
<?xml version="1.0" encoding="us-ascii"?>
<Patterns>
  <Pattern name="Create start production pattern">
    <Rule name="Convert all classes into nonterminals"/>
    <Rule name="Find all start symbols"/>
    <Rule name="Create start production" />
  </Pattern>
</<Patterns>
```

b)

Fig. 4. Excerpts from the XML files.

4.2 The OWL parser

The OWL parser (OWLP) extracts information used to fill the ODS. OWLP takes an OWL ontology as its input. The ontology is written in RDF/XML syntax. As previously mentioned, the RDF/XML syntax represents an XML representation for an RDF graph. RDF graphs can be manipulated in different ways [21]. The most suitable way for our framework is with the use of programs.

ODS contains data extracted from the OWL file. ODS is a data structure comprised of the following native structures: a class tree, an object properties tree, a datatype properties tree, and an individual tree. The class tree for the truncated version of the Home robot ontology is presented in Fig. 5. The entire Home robot ontology is represented in [4].

The nodes in the trees are objects that store information needed at each node. For instance, a node in the class tree stores an object that contains a name of the class, the lists of the equivalent classes, superclasses, members, disjoint classes, comments and labels.

The OWLP component uses the C#LibraryForOWL class library to acquire the necessary data from an OWL ontology. The library is our product and it is implemented in the programming language C#. The library is accessible within the framework.

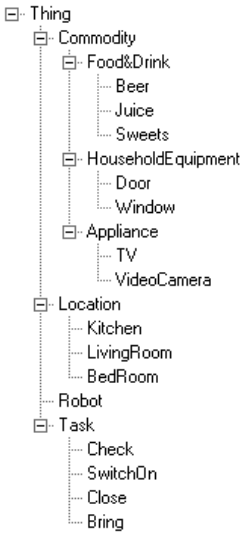

Fig. 5. Class tree.

4.3 Rule reader

The rule reader (RR) subsequently reads the rules written in the TP. RR transfers each rule to the rule execution component and to the transaction logger.

4.4 Rule execution component

The rule execution component (REC) is the main component of the transformation engine. Its purpose is the execution of rules. Each rule is executed separately and in the proper order.

REC works with two data structures: ODS and grammar data structure (GDS). REC acquires the necessary data from ODS and GDS. The results from every transformation step are written to GDS. GDS is a set of XML documents. The first document contains the set of classes acquired from the ontology, the rest contain individual sets from the quadruple G= (N,T,P,S) [10]. All five XML documents that form GDS are represented in Fig. 6a - 6e.

Fig. 6. presents the state of GDS after the application of the "Create start production pattern" pattern. The application of the pattern did not result in the creation of terminals; therefore, Fig. 6c presents only the structure of the document that contains the terminal set.

The most important of the aforementioned documents is the document presented in Fig. 6e, which contains the production set. Each production is associated with the name, which is also the content of the production and the name of the rule that was

used to create the production. Each production also stores its parts. Each part is associated with its type and the position on which it is located within the production.

Documents that contain sets from the quadruple G=(N,T,P,S) represent the final grammar after the transformation is complete. The final grammar is the result of the Ontology2DSL framework.

Fig. 6. Grammar data structure (GDS).

4.5 Transaction logger

The transaction logger (TL) logs the system state after every step (executed rule), thereby enabling the engineer to return to any step if necessary. The stored data is: the name of the executed rule and the entire contents of the ODS and GDS.

5 Case study

This chapter presents an example of the transformation of the Home robot ontology (HRO), which is presented as a class hierarchy in Fig. 5, into the HomeRobot DSL.

Before the transformation occurs, it is necessary to understand the target ontology and the purpose of its creation. Also, the requirements and the goal of the language being acquired must be known. Additionally, it should be noted that in most cases the DSL requirements do not match the ontology in every concept. It must also be understood that a single ontology can be used to develop many different DSLs.

HRO ontology contains terms that refer to the tasks of a home robot, the items that are subjects of the tasks and locations on which the tasks are performed. The purpose of the HomeRobot DSL language that is being developed is the execution of commands by the home robot.

For the purpose of the transformation we have created a transformation pattern named the "HomeRobot pattern," which is presented in Fig. 7. A pattern is created by selecting the required rules from the list of supported rules. The selected rules are then ordered into the correct sequence. The HomeRobot pattern incorporates 7 rules that directly correspond to the number of steps in the transformation.

Before the transformation, the OWLP component was used to fill the ODS. This resulted in the filled tree, presented in Fig. 5. The transformation was then carried out over these classes (concepts). The transformation resulted in the model (grammar) of the DSL.

In the first step of the transformation the rule "Convert all classes into nonterminals" was used. The rule transforms all the classes from the ontology to nonterminals, which are written to the corresponding document within the GDS.

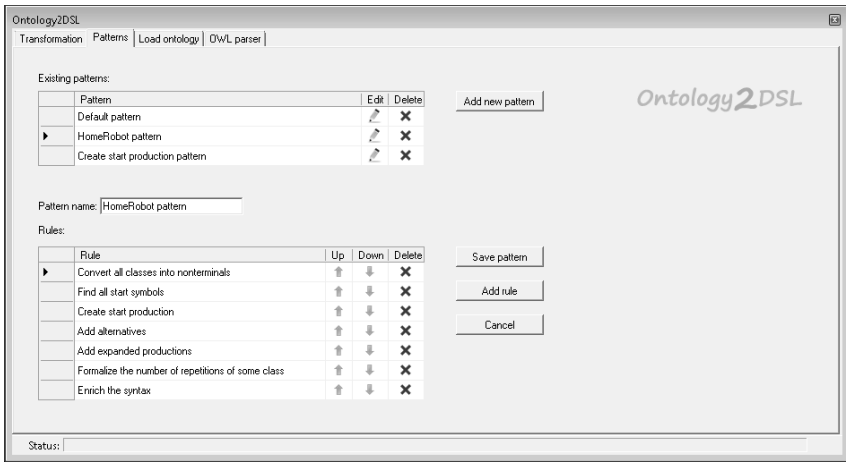

Fig. 7. The HomeRobot pattern.

In the second step, the rule "Find all start symbols" was applied and resulted in the acquirement of all possible grammar start symbols. In the third step the start production was created with the application of the "Create start production" rule. This step requires some involvement from a DSL engineer. The engineer builds the start production by selecting the start symbol from the application interface and sorts the remaining start symbols into the correct order. The interface that enables the creation of the start production is presented in Fig. 8.

In step four, the rule "Add alternatives" was applied. This rule resulted in all possible productions, which were added to the XML document containing the production set in the GDS. Step five expanded the user-selected production with the application of the "Add expanded productions" rule. This step also required some involvement from the DSL engineer. In the penultimate step, the number of repetitions of a particular class was determined with the rule "Formalize the number

of repetitions of some class." The last step used the "Enrich the syntax" rule to enrich the syntax. The last two steps required involvement from the engineer. The results of the transformation by individual steps, and the final grammar acquired by the transformation, are presented in Fig. 9.

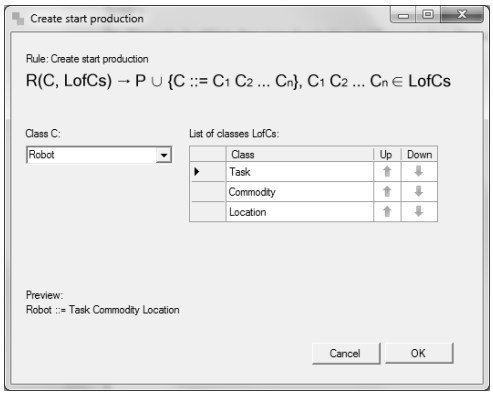

Fig. 8. The "create start production interface."

The output of the framework, besides the formal grammar, are also programs. Programs are automatically generated from the final formal grammar. Presently the framework generates basic programs. An example would be the `Close door kitchen` program, where the logic and sequence of closing a door at the specified location is required.

The Ontology2DSL framework produced optimal grammar. Optimal grammar was the result of using an appropriate pattern in the transformation process and the decisions that were made on the start production, syntax enrichment and elsewhere. It is important to note that all patterns are not appropriate for all transformations.

6 Conclusion and future work

In this paper, we focused on the presentation of the Ontology2DSL framework, which can be used to build DSLs from OWL ontologies. The functionalities of the framework, the design approach and the case study have been described. Special attention has been paid to its architecture.

The Ontology2DSL framework is still being actively developed. The current version enables working with patterns, the grammar and the construction of basic programs. The construction of more complex programs is intended for our future work. In the future, we also plan to develop the framework to its full potential. In addition, our future work will focus on: the validation of the resulting grammar, the use of currently ignored additional information in the ontology, the transformation of the developed DSL to a form that is compatible with compiler generators, and the evaluation of DSLs.

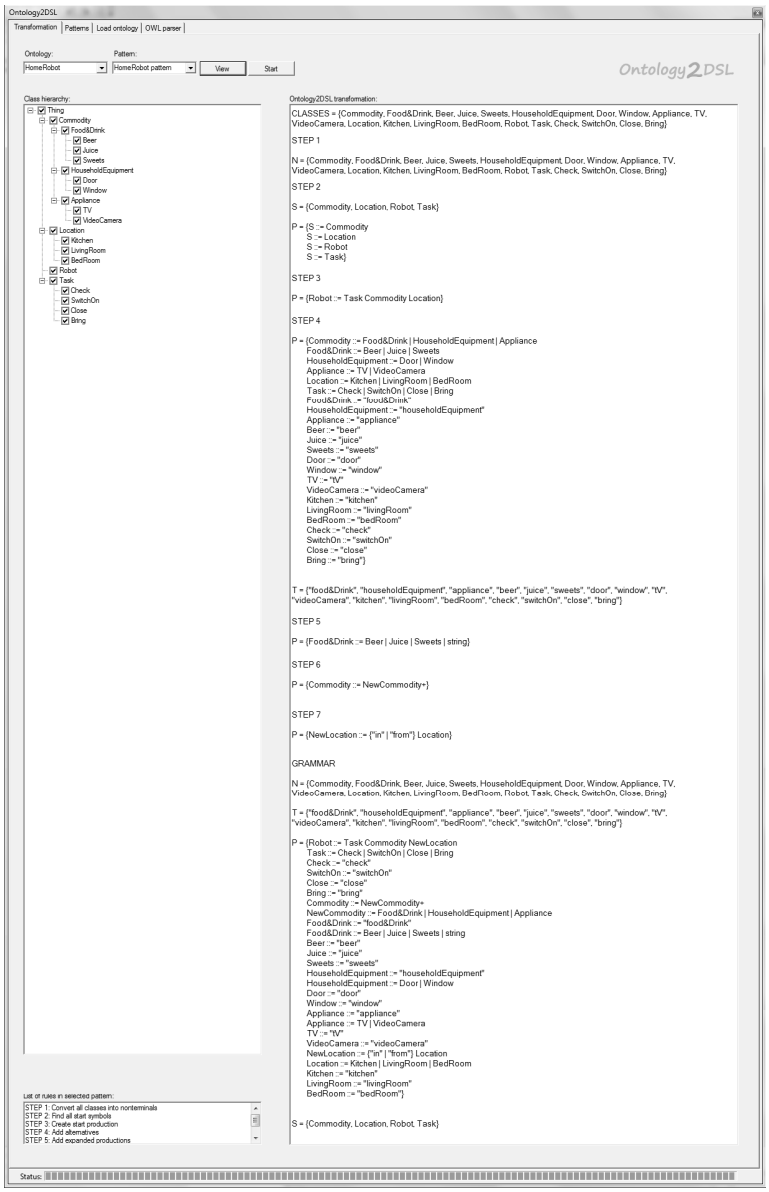

Fig. 9. Transformation steps and the final grammar of the HomeRobot language.

References

1. Mernik, M., Heering, J., Sloane, A. M.: When and how to develop domain-specific languages. ACM Computing Surveys 37, 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 7(2), 247-264 (2010)
3. Mernik, M., Hrncic, 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 2, pp. 421-457. Imperial College Press, London, (2011)
4. Čeh, I., Črepinšek, M., Kosar, T., Mernik, M.: Ontology Driven Development of Domain-Specific Languages. Computer Science and Information Systems 8(2), 317-342 (2011)
5. Kosar, T., Martinez Lopez, P. E., Barrientos, P. A, Mernik, M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology 50(5), 390-405 (2008)
6. Studer, R., Benjamins, R., Fensel, D.: Knowledge engineering: Principles and methods. Data & Knowledge engineering 25(1-2), 161-198 (1998)
7. Staab, S., Studer, R. (editors): Handbook on Ontologies. Springer Verlag (2009)
8. Lacy, L. W.: OWL: Representing Information Using the Web Ontology Language. Trafford Publishing (2005)
9. Hebeler, J., Fisher, M., Blace, R., Perez-Lopez, A.: Semantic Web Programming. Wiley Publishing (2009)
10. Aho, A. I., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison Wesley (2007)
11. 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)
12. 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, LNCS, vol. 6325, pp. 99-118. Springer-Verlag (2010)
13. 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)
14. Ontology-Based Evaluation and design of domain-specific visual modeling languages, http://www.loa-cnr.it/Guizzardi/ISD2005.pdf
15. 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)
16. Bräuer, M., Lochmann, H.: An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In: Sure, Y., Domingue, J. (eds.): The Semantic Web: Research and Applications. LNCS, vol. 5021, pp. 34-48. Springer, (2008)
17. RDF/XML syntax specification, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/
18. Protégé, http://protege.stanford.edu/
19. Mernik, M., Lenič, M., Avdičauševič, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Horspool, R. N. (ed.): Compiler Construction. LNCS, vol. 2304, pp. 1-4. Springer-Verlag (2002)
20. 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, 7(2), 291-307. (2010)
21. Grobe, M.: RDF, JENA, SparQL and the "Semantic Web". In: Proceedings of the 37th annual ACM SIGUCCS fall conference, pp. 131-138. (2009)