Semi-automated integration of domain ontologies to DSL meta-models

Hele-Mai Haav* and Andres Ojamaa

Institute of Cybernetics at Tallinn University of Technology,
Laboratory of Software Science,
Akadeemia tee 21/1, 12618 Tallinn, Estonia
Email: helemai@cs.ioc.ee
Email: andres.ojamaa@cs.ioc.ee
*Corresponding author

Abstract: This paper investigates the use of ontologies for the development of Domain-Specific Languages (DSL). In Model-based Software Development (MBSD), ontologies as formal models are mostly used in the phase of domain analysis as formal representations of domain models and requirements of a system. In this paper, we extend this view by introducing a concept of a DSL meta-model ontology that is defined as consisting of system ontology of a MSDB tool linked to one or more domain ontologies and external software artefacts. According to the solution presented in this paper, the given domain ontologies are semi-automatically integrated into a DSL meta-model using a set of transformations between constructs of ontology modelling language OWL and a modelling language used for representing DSL meta-models. This approach enables a dynamic semantic composition of a DSL meta-model and is prototypically implemented in Java as an extension to the DSL development tool CoCoViLa.

Keywords: DSL; domain specific language; meta-models; domain ontologies; semantic integration; DSL meta-models; DSL meta-model ontology; OWL; ontology web language; DSL development; ontology-based software engineering; model transformations; Java.


Biographical notes: Hele-Mai Haav is a Senior Researcher at the Institute of Cybernetics at Tallinn University of Technology, Estonia. She received her PhD in Computer Science from the Institute of Cybernetics of the Estonian Academy of Sciences, Estonia. Her current research is focused on ontology engineering and ontology integration to model-based software development.

Andres Ojamaa is a researcher at the Institute of Cybernetics at Tallinn University of Technology, Estonia. He received his MSc in Computer Science from Tallinn University of Technology. His current research interests are in model-based software development and cyber security.

This paper is a revised and expanded version of a paper entitled ‘Semi-automated generation of DSL meta models from formal domain ontologies’ presented at 5th International Conference on Model and Data Engineering (MEDI), Rhodes Island, Greece, 26–28 September 2015.
1 Introduction

Model-based software development (MBSD) is a software development method that uses software models in order to improve productivity and quality of the creation of a software system. MBSD tries to change the role of software models used traditionally as a part of a project documentation to become essential artefacts in the process of software development. As the consequence, this enhances requirements on software models to be formalised, complete, and precise. This in turn is related to a new trend in MBSD that concerns integration of ontologies as formal domain models to the MBSD process. For example, the EU project MOST (http://www.most-project.eu/) and its activities were devoted to promote the idea of integration of the Ecore meta-meta-model and Ontology Web Language (OWL) (Motik, 2012) meta-model in order to provide meta-meta-model for modelling DSL languages. The Ecore represents the meta-meta-model of the eclipse modelling framework (EMF) (Gronback, 2009).

In MBSD, ontologies as formal models are mostly used in the phase of domain analysis as formal representations of domain models and requirements of a system. In this paper, we extend this view by using a concept of a DSL meta-model ontology that is defined as consisting of system ontology of a MBSD tool linked to one or more domain ontologies and external software artefacts. OWL is used as a common language for representation of semantics of software artefacts.

The main contribution of this paper is an approach that focuses to a partial automation of the design and implementation phases of the DSL development process by introduction of formal domain ontologies into this process. The approach facilitates the semantic integration of distributed software artefacts into a coherent DSL meta-model as well as partial automation of the development lifecycle and evolution of a DSL.

Comparing to our previous work (Ojamaa et al., 2015; Haav et al., 2015), this paper combines the main results of the both works into one framework that extends previous results with providing precise implementations for execution of mapping rules from OWL to the CoCoViLa modelling language and for semi-automated generation of a DSL meta-model. In addition, this paper presents a new methodology for building DSL meta-models within the implemented framework.

Novelty of our approach comparing to other ontology-driven software development methodologies lies in using formal domain ontologies as a basis for automated generation of concept specification templates of a DSL meta-model that are consistent with the given domain ontology. This makes concepts of a DSL closely related to the domain concepts. The consistency of the DSL with domain knowledge is ensured by predefined mappings between constructs of the ontology modelling language OWL and a modelling language for DSL meta-models (i.e. the CoCoViLa modelling language in our case).

The approach is prototypically implemented as an extension to the current version of the DSL development tool CoCoViLa (http://cocovila.github.io/). Novel extensions provide developers with domain ontology-driven DSL modelling facilities. Our approach is general enough to be applied for the development of different kinds of DSLs (e.g. DSLs for physical systems and simulations) for various application domains by using the corresponding domain ontologies, the CoCoViLa system ontology and modelling language for the creation of DSL meta-models with the CoCoViLa tool. Concerning the usage of principles of our approach outside of the CoCoViLa system and its extensions requires development of system ontology for the particular DSL modelling tool, the
creation of the DSL meta-model ontology and using this for building a DSL meta-model within the framework of a particular tool.

The rest of the paper is structured as follows. Section 2 is devoted to the related work and Section 3 provides background knowledge about the CoCoViLa modelling language and describes system extensions. Section 4 is devoted to a description of ontologies used for the DSL development process. In Section 5, a set of transformations from OWL ontologies to a set of constructs of the CoCoViLa DSL modelling language is provided. Section 6 gives an overview of the process of building DSL meta-models using the approach presented in this paper. Section 7 provides an evaluation of the approach. Section 8 concludes the paper.

2 Related work

Software development may benefit from the usage of ontologies in many ways. For example, ontologies enable to dynamically adapt the system to new semantic constraints. On the other hand, ontologies may be used in domain engineering during the domain analysis as a systematic and formal way of capturing domain knowledge (Aßmann and Zschaler, 2006; Čeh et al., 2011; Guizzardi, 2013; Walter et al., 2014). In these works, domain models are considered as descriptive models consisting of a set of domain instances [ABox in description logics (DL)] (Baader et al., 2003) and a set of classes for classifying these instances (TBox in DL). Comparing to traditional analysis models formally described ontologies have additional useful features like reasoning based on DL (Baader et al., 2003).

We may consider ontology driven software development as a kind of MBSD, where models of different levels are based on ontologies. Some approaches (Staab et al., 2010; Walter et al., 2014) integrate ontologies to the OMG meta-pyramid of MDA (OMG MDA) and to the Ecore meta-meta-model of the EMF (Gronback, 2009) in order to provide meta-meta-model for modelling DSL languages. In contrast to the above mentioned approaches, we do not consider the general OMG MDA and the Ecore meta-meta-model as targets for ontology integration. However, similarly to these works, we also introduce ontologies to the meta-meta-model level but we focus to the particular DSL modelling language and the tool.

In the existing ontology driven software engineering methodologies, ontologies are mostly used for the consistency checking of software models (Staab et al., 2010) and as tools for representing model transformations (Roser and Bauer, 2008). With relation to model checking, we use a DSL meta-model ontology for semi-automatically providing DSL meta-models that are consistent with the given domain ontology that is a part of the DSL meta-model ontology. Comparing to ontology supported model transformation in Roser and Bauer (2008) we do not use reference ontology in order to annotate meta-models and to provide ontology-based model transformations. However, we use predefined transformation rules for generating concept specification templates of the modelling language from domain ontologies described in OWL.

Proposals that are tightly related to the work provided in this paper can be divided according to the modelling and technical viewpoints. From the modelling point of view, semantic search and composition of models of software components (Katasanov, 2012) are important topics. Their work is based on the usage of several types of ontologies and provides ontology-based services for semi-automated composition of software models. Its
similarity to our approach consists of utilising a software system ontology that principally can be compared to the system ontology component of our DSL meta-model ontology. In contrast to their model composition services we do not consider this process on the model level but instead of that the CoCoViLa system uses its internal computational model in order to automatically compose a model of an executable program. Another difference is that our approach is implemented as a part of the CoCoViLa DSL development tool but their framework is not implemented as a part of a tool.

From the technical point of view, works devoted to integration of OWL and Java are essential. For example, a hybrid modelling approach that enables software models partially developed in Java and in OWL is given in Puleston et al. (2008). They use a notion of concept referencing class that is a Java class including references to the relevant domain ontology classes. Corresponding mappings are provided via a configuration file. Comparing to their solution, we semi-automatically create Java classes from concept specification templates that are automatically generated from the given DSL meta-model ontology by the usage of Java and Apache Jena (https://jena.apache.org/).

3 Background knowledge of the CoCoViLa modelling language and extensions to the system

3.1 An overview of the modelling language

The CoCoViLa modelling language allows to specify DSLs and the corresponding meta-models as well as application oriented models in the domains where mathematical computations play an important role (e.g. engineering, manufacturing, simulation, etc.) (Kotkas et al., 2011).

The most important features of the CoCoViLa modelling language are as follows:

- declarative and reusable specifications
- tight integration with Java
- capacity to support automated code generation from specifications of models.

The CoCoViLa modelling language consists of diagrammatic and textual declarative languages. Intended users of the CoCoViLa modelling language are DSL designers (together with domain experts), who create meta-models and a DSL for a particular domain. DSL designers can create textual and/or graphical DSLs.

3.2 Main concepts of the CoCoViLa textual language

The most important construct of the textual declarative modelling language is a concept specification that represents a collection of instances (see Figure 1). Concept specifications can be arranged into taxonomy.

Concept specifications include descriptions of structural components of a concept as declarations of variables (see Figure 1). In addition, they may include relations that are specifications showing how to derive values of some variables from the values of other variables.
Relations are divided into equations and axioms. Equations define dependencies between variables bound by the equation. Axioms describe functional dependencies between variables and differ from equations in that they have realisations. Realisations of axioms are procedural descriptions of algorithms that specify how to carry out computations from input to output. They are implemented by Java methods. However, other programming languages can also be used for implementation of algorithms.

Let us consider as an example the concept specification of a circle from the geometry domain. In the following specification of the concept Circle, all possible calculations are included as equations (see Figure 2).

The concept specification of the concept Circle starts with the keyword specification followed by the name of the concept i.e. Circle.

The following statement is the declaration of variables with type double:

```java
public class Circle {
    /*@ specification Circle */
    double Radius, Diameter, Area, Perimeter;
    Area = 3.14 * Radius ^ 2;
    Perimeter = 6.28 * Radius;
    Diameter = 2 * Radius;
    }
    @*/
```

The concept specification is included between Java comments (/* and */) and wrapped into the corresponding Java class that has the same name as the concept specification. Concept specifications are ignored during the compilation of Java source code. However, they are used in the process of automatic program construction.

We may also specify a square using the same type of language constructs (see Figure 3).
If we are interested in an application that needs to calculate the area that is difference of areas of a circle and a square where diagonal of a square is equal to diameter of a circle, then we need to create a new concept specification (e.g. `SquareWithinCircle`) called also application specification, where we reuse previously specified concept specifications to specify a particular task to be solved. The application specification is depicted in Figure 4.

In this specification we see that the specifications of concepts `Square` and `Circle` are used as types for variables `S` and `C`. Dot notation is used to refer to inner concept specifications in the following equations:

\[
C.\text{Diameter} = S.\text{Diagonal}; \\
\text{Area}\_\text{difference} = C.\text{Area} - S.\text{Area}; \\
S.\text{Height} = 10; \\
S.\text{Height} \rightarrow \text{Area\_difference};
\]

In order to perform calculations (e.g. compute value of the variable `Area\_difference` by given value of variable `S.\text{Height}`) on the basis of given specifications and automatically generate the corresponding program we need to provide input values of variables (e.g. `S.\text{height}`) and specify the task to compute the value of the variable `Area\_difference` from given value of the variable `S.\text{Height}`. The program that computes this value is automatically generated using a method of automatic program synthesis (Mints and Tyugu, 1982).
The computability information can be represented by simple equations as in Figures 2 to 4, or it can be explicitly represented by axioms specifying the applicability of methods of a Java class. In the current version of the CoCoViLa system equations are considered as a set of axioms derivable from them.

For example, instead of the following equation in Figure 4

\[ \text{Area}\_\text{difference} = \text{C.Area} - \text{S.Area}; \]

we may write or the system can generate the following axioms

\[ \text{C.Area, S.Area} \rightarrow \text{Area}\_\text{difference} \quad \text{(<JavaMethod1>)} \]
\[ \text{C.Area, Area}\_\text{difference} \rightarrow \text{S.Area} \quad \text{(<JavaMethod2>)} \]
\[ \text{Area}\_\text{difference, S.Area} \rightarrow \text{C.Area} \quad \text{(<JavaMethod3>)} \]

In these statements, Java methods are declared as realisations of the given axioms that represent the computability of the values of the corresponding variables from the values of the other variables. In CoCoViLa, realisations (i.e. Java methods) of the axioms derived from the equations are automatically generated on the basis of the given equations. Equations are solved analytically and it is not allowed to have the same variable on both sides of an equation.

CoCoViLa uses this computability information for automatically synthesising, if possible, a Java program that performs a task given by a task specification.

A task specification specifies a task of computing values of output variables listed in the task specification from given values of input variables listed in the task specification. Input list of the task specification can be empty. The method of program synthesis used in CoCoViLa is based on formal logic and it is called the Structural Synthesis of Programs (SSP) (Mints and Tyugu, 1982; 1990).

Using the application specification SquareWithinCircle (see Figure 4) we can specify different task specifications, which specify computational problems that need to be solved.

For example, we can use the task specification statement

\[ \rightarrow \text{Area}\_\text{difference}; \]

or

\[ \text{S.Height} \rightarrow \text{Area}\_\text{difference}; \]

In the case of the first statement we require to compute a value of area difference of a circle and a square under conditions given in the specification. Values of input variables are assumed to be given or computable on the basis of the specification. Thus, we additionally need to give an input value as follows (see Figure 4):

\[ \text{S.Height}=10; \]

If output variables of a task are computable from (possibly empty) list of inputs, the task is solved and the Java program is automatically generated. The syntax of the full CoCoViLa modelling language is presented and explained in Kotkas et al. (2011).
3.3 The CoCoViLa system and its extension with ontology-based DSL development facilities

3.3.1 The CoCoViLa system

CoCoViLa is a system for DSL development and application. The CoCoViLa modelling language is used for expression of DSL meta-models. A DSL meta-model created by using CoCoViLa consists of the following three parts: a textual specification, an XML description of graphical part of the model and Java classes together with methods that represent axioms and other implementation details of a DSL. Application specific models built by using a particular DSL are instances of the DSL meta-model and they need to include a task specification.

For developers, the CoCoViLa system consists of two parts that can be executed separately: the Class Editor and the Diagram Editor. The Class Editor is used for modelling and implementing DSLs for various domains. This can be done by diagrammatically defining different elements of a DSL and their interactive aspects as well as by using the textual specification language and Java. The Diagram Editor is a tool for application modelling by using a DSL. Diagram Editor provides tools for drawing diagrams using a diagrammatic language of a DSL, compiling and running programs defined by an application diagram and/or a textual specification and a task specification.

The current DSL development process requires a tight co-operation between a domain expert and a programmer in order to transfer the domain knowledge into Java classes and annotate these classes with concept specifications. However, steps related to the DSL application for solving a particular problem are mostly done automatically by the tool as depicted in Figure 5.

Figure 5 Components of the CoCoViLa tool (see online version for colours)

Application developers build the problem specification using the DSL and translate it into the computational model with the help of the CoCoViLa tool. The following components are included to the automated process: a computational model, the planner, an algorithm and the generated Java code. For more details we refer to Grigorenko et al. (2005).
Computational model is an internal structure for representing the computational problem and concept specifications. The planner (a theorem prover) takes as input the computational model and considers it as a logical theorem with axioms derived from the functional dependencies defined in the specification. The theorem prover is based on intuitionistic logic that makes it possible to extract an algorithm of solving the specified computational problem from the constructive proof. The algorithm is basis for generation of the Java source code. CoCoViLa compiles and executes the source code at runtime and provides the results to the user. The generated code can be later (re)used.

3.3.2 The CoCoViLa extension

Figure 6 depicts the architecture of the CoCoViLa extension (see also Haav et al., 2015) that is mainly related to the improvement of a DSL development while components of the previous system are used for a DSL application.

The CoCoViLa extension provides facilities for DSL designers to carry out the ontology-based DSL development process.

3.3.3 The CoCoViLa extension

Figure 6 The CoCoViLa extension (see online version for colours)

Most important components that constitute the extension of the CoCoViLa architecture are a DSL meta-model ontology and concept specification templates. The templates are automatically generated from a DSL meta-model ontology, particularly, from domain ontologies. After a manual enhancement, the templates become ordinary concept specifications of the CoCoViLa modelling language. Those constitute the most important part of a DSL meta-model. The process of automated generation of a DSL meta-model requires as input also instances of the corresponding DSL meta-model ontology.

The components that constitute the CoCoViLa extension are in more detail described and explained in the following sections of this paper.

4 Ontologies used for DSL development

We represent ontologies in OWL that is widely utilised in semantic web technological space and supported by well-known standards by W3C as well as description logics (DL).
reasoning facilities (Baader et al., 2003). OWL ontologies are serialised in the resource description framework (RDF, https://www.w3.org/RDF/).

Ontologies play twofold role in our approach as follows:

- ontologies as software artefacts (i.e. RDF documents) that semantically describe other software artefacts (e.g. the CoCoViLa system) or domain(s) are used in the DSL development and implementation processes.
- ontologies (e.g. domain and system ontology) as web resources (i.e. RDF documents) are accessed using SPARQL (https://www.w3.org/TR/rdf-sparql-query/) queries during the process of automated generation of a DSL meta-model.

We basically consider the following types of ontologies: the MBSD system ontology that serves as a conceptual model of the MBSD tool used for a DSL development, domain ontologies that provide domain-specific knowledge and the DSL meta-model ontology that integrates the system and domain ontologies.

### 4.1 The concept of DSL meta-model ontology

The notion of the DSL meta-model ontology plays central role in our approach. It was introduced for the first time in Ojamaa et al. (2015). We define the DSL meta-model ontology as a formally described ontology that links together the system ontology and one or more domain ontologies as well as may include links to external software artefacts on the Web (see Figure 7). Software artefacts are needed for the DSL development and application processes. These are for example several types of components of a DSL meta-model like CoCoViLa specifications, Java classes from Java libraries, diagrams and their elements, application packages, the Java source code, etc.

![Figure 7](image.png) The concept of DSL meta-model ontology
The system ontology formally describes concepts of a particular modelling language and the corresponding software system as well as relationships among them. For example, the CoCoViLa system ontology includes concepts like JavaClass, MetaClass, ConceptSpecification, etc.

The domain ontology provides a specification of a conceptualisation of a domain. For example, the geometry domain ontology may contain concepts like TwoDimShape, Rectangle, Square, etc.

The DSL meta-model ontology facilitates semantic integration of software artefacts for DSL development and implementation by semantically describing these artefacts or providing links to external web resources. As shown in Figure 7, concepts of the system ontology may be related to the concepts of domain ontology via the implementation relationship or they may provide links to external web resources used for DSL development. In addition, a part of system ontology concepts may refer to components of models created by the other DSL development tools. This case requires availability of the system ontology of these external tools. In order to ensure the consistency of a DSL meta-model ontology, DL reasoning services are used.

4.2 The CoCoViLa system ontology

The CoCoViLa system ontology formally describes the CoCoViLa modelling language and system concepts. In Figure 8, a fragment of this ontology is given in OWL functional style syntax (http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/), where the prefix ‘sys’ denotes the CoCoViLa system ontology elements. The current version of this ontology includes OWL descriptions of 40 classes, 21 object properties and 16 data properties.

The ConceptSpecification class represents a collection of concept specifications that are textual specifications in the CoCoViLa modelling language. Concept specifications are used to facilitate automatic composition of an application. Individuals of the ConceptSpecification class are related to the automatically generated ConceptSpecificationTemplate class instances via the hasTemplate object property.

Concept specification templates are restricted forms of concept specifications that do not include specifications of dynamic parts of a DSL meta-model like relations (except equality relation). These templates are automatically generated from class descriptions of domain ontology using transformation rules provided in Chapter 5 of this paper. The corresponding relationship is expressed in ontology by the isGeneratedFrom object property that provides links to domain concepts used in a DSL meta-model. Concept specification templates can be later manually extended with the CoCoViLa language statements that could not be covered by transformations (e.g. equations and relations).

Individuals of the MetaClass class are implemented on the basis of individuals of the ConceptSpecification class and Java. The corresponding relationship is represented by the implements object property whose domain is the class MetaClass that is a subclass of the class JavaClass collecting instances that are Java classes. The MetaClass class collects individuals that are Java classes and may contain Java methods that are realisations of relations described by the ConceptSpecification class individuals.
Figure 8  A fragment of the CoCoViLa system ontology

```
SubClass axioms

SubClassOf( sys:VisualClass sys:MetaClass )
SubClassOf( sys:MetaClass sys:JavaClass )
SubClassOf( sys:JavaClass sys:Thing )
SubClassOf( sys:DiagramElement sys:Thing )
SubClassOf( sys:Field sys:Thing )
SubClassOf( sys:Port sys:Thing )
SubClassOf( sys:ConceptSpecificationTemplate sys:Thing )
SubClassOf( sys:ConceptSpecification sys:Thing )

DataProperty axioms

DataPropertyDomain( sys:hasIcon sys:VisualClass )
DataPropertyRange( sys:hasIcon xsd:anyURI )

ObjectProperty axioms

ObjectPropertyDomain( sys:isGeneratedFrom sys:ConceptSpecificationTemplate )
ObjectPropertyRange( sys:isGeneratedFrom sys:Thing )
ObjectPropertyDomain( sys:implements sys:MetaClass )
ObjectPropertyRange( sys:implements sys:ConceptSpecification )
ObjectPropertyDomain( sys:hasTemplate sys:ConceptSpecificationTemplate )
ObjectPropertyRange( sys:hasTemplate sys:ConceptSpecificationTemplate )
DataPropertyDomain( sys:hasField sys:VisualClass )
DataPropertyRange( sys:hasField sys:Field )
DataPropertyDomain( sys:hasPort sys:VisualClass )
DataPropertyRange( sys:hasPort sys:Port )
```

It is possible to use diagrammatic elements for a DSL development. Therefore, the CoCoViLa system ontology includes several classes for representing diagram elements as subclasses of the DiagramElement class. Individuals of these elements can come from external sources and be linked via URIs to the DSL meta-model ontology. However, diagrammatic language elements can be also created by the CoCoViLa class editor.

For diagrammatic language of CoCoViLa the notion of a visual class is used. The VisualClass class is a subclass of the MetaClass class. Its individuals are the MetaClass class individuals that are extended with an image, ports and fields. The VisualClass class individuals have the data property hasIcon that could be URI to an image of an icon used for denoting a visual class on a toolbar of the CoCoViLa DSL window.

4.3 Domain ontologies

4.3.1 Domain ontologies as formal models of a domain

In MBSD, ontologies are usually considered as conceptual domain models, which are used in domain engineering to describe the problem domain that a software system should support (Puleston, 2008; Tairas et al., 2009; Walter et al., 2014). Domain ontologies are descriptive models of a static part of a domain therefore they do not capture behavioural aspects of a domain. In order to include a kind of behaviour of a domain of interest into the DSL meta-model, the DSL designer must understand the
purpose of the domain ontology and its scope. On the other hand, the DSL designer also needs to take into account the DSL purpose and requirements. Some of the requirements can be modelled using ontologies.

Domain ontologies for the DSL development can be built from scratch or existing ontologies of a domain can be reused. When reusing existing domain ontologies, it might happen that the DSL requirements and the domain ontology do not match in all concepts. For example, the domain ontology can be too general as it is used for development of many different DSLs. In this case the DSL designers together with domain experts need to redesign a given domain ontology.

According to our approach, domain ontologies implemented in a formally defined language like OWL are integrated into the analysis phase as formal domain models. They are developed by domain experts or by ontology engineers together with domain experts or with the DSL designer.

4.3.2 Building domain ontologies

In order to construct ontologies from scratch the ontology engineering methodologies are used [e.g., METHONTOLOGY (Gómez-Pérez et al., 2004)]. More detailed overview of wide range of methodologies can be found in Gómez-Pérez et al. (2004).

The most well-known methodology is METHONTOLOGY that enables the construction of ontologies at the knowledge level. METHONTOLOGY is built on the basis of the main activities identified by the software development process combined with the Knowledge Engineering methodologies.

However, the majority of approaches require extensive technical knowledge of formal languages and techniques for capturing knowledge of a domain.

Therefore, we propose to use the domain expert centric ontology development methodology (Haav, 2011). This methodology was originally developed for building e-government domain ontologies, but it is general enough to be applied in any domain where domain experts themselves are intended to create ontologies.

According to this methodology, main activities of ontology development process are as follows: specification, conceptualisation and implementation. Following the iterative life cycle model early implementation of domain concepts and relationships is foreseen. At the first iteration, basic (central) domain concepts and relationships are arranged to ontological structure and implemented. After that, during each of the iterations, new concepts and relationships are added and implemented. Attributes of individuals of concepts are added in the final stage. After that ontology is evaluated according to the requirements of the domain.

5 Transformations from OWL ontologies to a set of concept specification templates

In order to automatically generate a set of concept specification templates from the given domain ontologies we need to define mappings from OWL to the CoCoViLa modelling language. This process requires taking into account the semantics of the both languages.

After defining mapping rules the transformations need to be implemented and executed. This process involves SPARQL access to the OWL descriptions of ontologies that are RDF documents. In this case, ontologies play a role of web resources.
5.1 Mapping rules from OWL to the CoCoViLa modelling language

In Ojamaa et al. (2015) we have defined a set of mappings between a subset of OWL DL constructs and a subset of the CoCoViLa modelling language constructs. Both languages are declarative languages intended to be used for knowledge representation. OWL DL semantics is given by DL (Baader et al., 2003) and the CoCoViLa modelling language semantics is based on a subset of intuitionistic propositional calculus (IPC) (Mints and Tyugu, 1982, 1990). DL enables to reason whether domain ontology is consistent and complete. Expressive power of SSP that is used for deciding about computational correctness of a program automatically generated from a given specification is equivalent to IPC (Mints and Tyugu, 1982).

Table 1  Mappings between OWL and the CoCoViLa modelling language constructs

<table>
<thead>
<tr>
<th>Acceptable OWL constructs of formal domain ontology (i.e. the CoCoViLa compatible OWL ontology)</th>
<th>The corresponding CoCoViLa modelling language statements</th>
</tr>
</thead>
</table>
| **owl:Class** Class C | specification < concept specification name > {}  
  specification C {} |
| **owl:DataTypeProperty**, rdfs:domain, XML Schema datatypes  
  DataProperty P domain D range &xsd;double | specification < concept specification name1> {<Java type> < variable name >}  
  specification D { double P;} |
| **owl:ObjectProperty**, rdfs:domain, rdfs:range  
  ObjectProperty P domain D range R | specification < concept specification name1> {<concept specification name2> <variable name>;}  
  specification D { R P;} |
| **owl:equivalentProperty**  
  P1 equivalentProperty P2 | specification <concept specification name1> = < variable name2>;  
  P1=P2; |
| rdfs:subClassOf  
  C1 subClassOf C2 | specification <concept specification name1> super <concept specification name2> {... }  
  specification C1 super C2 () |
| **owl:NamedIndividual** NamedIndividual I with DataProperty P and its value 5 | specification <concept specification name>.<variable name> = <value>;  
  I.P=5; |

The OWL constructs for object property characteristics, property restrictions and complex classes cannot be semantically properly mapped to the CoCoViLa language. Therefore, domain ontology structure is restricted accordingly. Table 1 summarises the mappings for OWL constructs that are allowed in the CoCoViLa compatible ontologies previously defined by us in Ojamaa et al. (2015). OWL constructs for what mappings are not defined are not allowed and the corresponding OWL constructs are ignored in the implementation process of transformations. This leads to the limited ability of our
approach to capture full knowledge from OWL domain ontologies as the logical language behind the CoCoViLa program synthesis method is less expressive than DL.

5.2 Implementation and execution of mappings

The previously given mappings are used to automatically generate concept specification templates in the CoCoViLa modelling language. General view of the implementation process of the given transformations is depicted in Figure 9.

Figure 9 General view of the implementation of mappings (see online version for colours)

The transformer is a Java program that implements the defined (see Table 1) transformations by accessing domain ontology as a web resource from the DSL meta-model ontology using SPARQL queries in order to get needed ontological information. As a result, the transformer outputs a set of corresponding concepts specification templates as the CoCoViLa specifications.

The transformer basically consists of two Java programs: OntologyProcessor and TemplateProvider (see Figure 9). The OntologyProcessor program extensively uses Apache Jena library tools for working with ontology. It takes as the input the URL of a RDF document of the DSL meta-model ontology and provides as the output the internal representation of the DSL meta-model. The list of basic steps of the OntologyProcessor program is as follows:

1. Loading the DSL meta-model ontology using Jena tools.
2. Validating the ontology using Jena tools.
3. Invoking Jena inference engine for inferring all classes and for consistency checking of ontology.
4. Extraction of domain ontology classes and related to them subclass axioms, properties and individuals from the DSL meta-model ontology using SPARQL queries. Domain ontology classes should be disjoint in our case.
5. Inserting the results of SPARQL queries into the internal structure of the DSL meta-model.

For example, one of the SPARQL queries used for finding information about what data properties are related to individuals of a class is represented in Figure 10.
The TemplateProvider program takes as the input the internal representation of the DSL meta-model and generates a set of concept specification templates that correspond to the given domain ontology. The generation process takes into account the mapping rules given in Table 1.

In addition, the TemplateProvider program adds the following assertions to the DSL meta-model ontology:

- Assertions of individuals of the ConceptSpecificationTemplate class according to the generated templates.
- Assertions of the isGeneratedFrom object property values that provide links between individuals of the ConceptSpecificationTemplate class and the corresponding domain concepts used in a DSL meta-model.

6 The methodological process of building a DSL meta-model

6.1 The process overview

Basically, two inputs are needed for the creation of the DSL meta-model: its meta-model ontology and the corresponding collection of concept specifications in CoCoViLa. The meta-model ontology is also used as a resource that is dynamically accessed by the system using SPARQL queries (see for example, Figure 10) during the automated process of the creation of the DSL meta-model.

During the design phase of a DSL development, the DSL designer creates the DSL meta-model ontology and imports domain ontologies represented in OWL into the DSL meta-model ontology. An overview of the process of development of a DSL meta-model is depicted in Figure 11.

**Figure 10** An example of a SPARQL query

| WHERE {
| ?o rdf:type owl:NamedIndividual; |
| rdf:type sys:VisualClass; |
| rdf:type ?dc. |
| ?dc rdf:type owl:Class. |
| ?ps rdf:type owl:DatatypeProperty; |
| rdfs:domain ?dc; |
| rdfs:range ?type. |
| FILTER (?c = ?dc) |
| & & isTRI(?ps), |
| & & lstrstarts(str(?ps), |
| & & lstrstarts(str(?dc), "http://www.w3.org/2000/07/owl#") |
| & & lstrstarts(str(?type), "http://www.w3.org/2001/XMLSchema#") |
| ) |} |
When implementing a DSL, the DSL designer by using the CoCoViLa system automatically generates a set of concept specification templates and corresponding individuals and relationships (object property values) of the DSL meta-model ontology. This process automatically links domain ontologies to the DSL meta-model ontology.

After that, the DSL designer has to insert manually individuals and necessary property values into the DSL meta-model ontology. These are mainly related to the system ontology concepts and their relationships with the implementation of the domain ontology concepts.

In order to finish the implementation phase of a DSL, the DSL designer has to create concept specifications from the generated concept specification templates. The automated process of generation of a DSL meta-model and specifically its internal representation uses the DSL meta-model ontology instances and a collection of concept specifications related to the DSL meta-model.

The overall process of building a DSL is illustrated by an example from the geometry domain.

6.2 An example of the development of the geometry DSL

Let us consider a simple example of the creation of a meta-model ontology for the Geometry DSL. In Figure 12, a fragment of the geometry domain ontology is depicted in OWL functional style syntax, where the prefix ‘geo’ denotes the geometry ontology elements.

In order to create the corresponding DSL meta-model ontology, the CoCoViLa system ontology (see Figure 8) and domain ontology (see Figure 12) are imported to the geometry DSL meta-model ontology.

For implementing the DSL, the designer generates a set of concept specification templates from the given domain ontology using the transformer program.
Figure 12  A fragment of the geometry domain ontology description

```
SubClass axioms

SubClassOf( geo:Square geo:Rectangle )
SubClassOf( geo:Rectangle geo:TwoDimShape )
SubClassOf( geo:Circle geo:TwoDimShape )
SubClassOf( geo:Triangle geo:TwoDimShape )
SubClassOf( geo:TwoDimShape geo:GeometricShape )

DataProperty axioms

DataPropertyDomain( geo:Diagonal geo:Rectangle )
DataPropertyRange( geo:Diagonal xsd:double )
DataPropertyDomain( geo:Height geo:Rectangle )
DataPropertyRange( geo:Height xsd:double )
DataPropertyDomain( geo:Width geo:Rectangle )
DataPropertyRange( geo:Width xsd:double )

DataPropertyDomain( geo:Perimeter geo:TwoDimShape )
DataPropertyRange( geo:Perimeter xsd:double )
DataPropertyDomain( geo:Area geo:GeometricShape )
DataPropertyRange( geo:Area xsd:double )
```

For example, the following templates in Figure 13 are automatically generated from the OWL class `Square` and its super-classes described in the given geometry domain ontology.

Figure 13  Some domain specific concept specification templates in the CoCoViLa textual modelling language

```
specification GeometricShape {  
    double Area;  }

specification TwoDimShape super GeometricShape {  
    double Perimeter;  }

specification Rectangle super TwoDimShape {  
    double Height, Width, Diagonal;  }

specification Square super Rectangle {  
    Height = Width;  }
```

In addition, the transformer program links the domain ontology classes to the instances of the `ConceptSpecificationTemplate` class via the `isGeneratedFrom` object property value. Afterwards, the designer adds manually some additionally needed property values into the DSL meta-model ontology and completes the templates to become the CoCoViLa concept specifications.
For example, the generated concept specification template of the concept Square can be manually completed by corresponding equations for calculating, for example, values of the variables Diagonal and Area. After that the template becomes the complete concept specification for the concept Square (see Figure 3) that is related to its template via the hasTemplate object property value.

As a result, the final geometry DSL meta-model ontology individuals and property values may be as depicted in Figure 14, where the prefix ‘meta’ denotes the geometry DSL meta-model ontology elements. Figure 14 basically shows that the concept specification template CST1 is generated from the Rectangle class of the domain ontology and it is the template for the concept specification CS1 that is implemented by the visual class VC1.

The DSL meta-model enables to link the other domains in the analogous way as described above. The consistency of the DSL meta-model ontology is checked by using ontology inference provided by Apache Jena.

Given a set of extended concept specification templates (see Figure 13 for templates) and the Geometry DSL meta-model ontology (see Figure 14), the extension of the CoCoViLa system is able to automatically generate the Geometry DSL meta-model that has a certain internal representation.

The corresponding diagrammatic geometry DSL (see Figure 15) is automatically generated from its meta-model (i.e. internal representation) and is made available for the user for application development. The DSL meta-model ontology contains also system ontology concepts (e.g. Icon) and their instances, therefore the diagrammatic language is easy to generate.

**Figure 14** The DSL meta-model ontology instances (a fragment)

<table>
<thead>
<tr>
<th>Individuals and property values of the geometry DSL meta-model ontology</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassAssertion{ sys:ConceptSpecificationTemplate meta:CST1 }</td>
</tr>
<tr>
<td>ClassAssertion{ sys:ConceptSpecification meta:CS1 }</td>
</tr>
<tr>
<td>ClassAssertion{ sys:VisualClass meta:VC1 }</td>
</tr>
<tr>
<td>ClassAssertion{ geo:Rectangle meta:Rectangle }</td>
</tr>
<tr>
<td>ObjectPropertyAssertion{ sys:isGeneratedFrom meta:CST1 meta:Rectangle }</td>
</tr>
<tr>
<td>ObjectPropertyAssertion{ sys:hasTemplate meta:CS1 meta:CST1 }</td>
</tr>
<tr>
<td>ObjectPropertyAssertion{ sys:implements meta:VC1 meta:CS1 }</td>
</tr>
<tr>
<td>DataPropertyAssertion{ sys:hasIcon meta:VC1 &quot;<a href="http://www.cs.ioc.ee/cocovila/icons/rect.png%22%5E%5Exsd:anyURI%7D">http://www.cs.ioc.ee/cocovila/icons/rect.png&quot;^^xsd:anyURI}</a></td>
</tr>
</tbody>
</table>

The concepts of the diagrammatic language (square, circle, and rectangle) are visible in the horizontal palette (toolbar) as icons. The icons can be activated to create new objects (instances). Three objects have been added to the diagram. The palette also contains tools for selecting objects and connecting ports.

On the application specific modelling level, we can use this DSL for implementation of different applications. For example, we may create an application for calculation of the
area that is difference of areas of a circle and a square where diagonal of a square is equal to a diameter of a circle as shown in Section 3.

**Figure 15** An example of the Geometry DSL (see online version for colours)

7 **An evaluation of the approach**

From our previous experience of using CoCoViLa in modelling and simulation of physical systems we have learned that tighter integration of domain knowledge with the system is needed in order to achieve consistency of a DSL with the corresponding domain knowledge and representation of different kinds of knowledge in a way that is reusable and independent on the internal representation of the DSL meta-model used by CoCoViLa. Therefore, the main evaluation criteria for the approach were a level of alignment of domain knowledge captured in domain ontologies with DSL meta-models and independence of reusable knowledge from the internal representation of DSL meta-models.

The domain of the IT security risk analysis is one of the domains where we have used CoCoViLa successfully for building a threat modelling tools for educational purposes in IT security study programs. The previously developed DSL uses an attack trees methodology and consists of components to model attack trees and to perform computations on the trees. We have presented preliminary ideas related to ontology driven redevelopment of this DSL in Ojamaa et al. (2015). Currently, this DSL is entirely re-implemented according to the methodology presented in this paper.

The DSL meta-model ontology for the attack tree DSL consists of the following four ontologies: the CoCoViLa system ontology, domain ontology for the threat modelling method (see Ojamaa et al., 2015), ontology of generic reusable components of the CoCoViLa simulation toolbox, and ontology of a library of attack models. The meta-model of the attack tree DSL is automatically generated on the basis of this meta-model ontology and the corresponding diagrammatic DSL is provided for the application developer.
After this completely ontology-driven implementation of the attack tree DSL we had opportunity to compare the both implementations and evaluate the approach. Taking into account the evaluation criteria given above the following assessments can be made.

- **A level of alignment** of domain ontologies with DSL meta-models was improved in the structural parts of the specification of modelling concepts as well as building correct inheritance structures of concepts. There is no need to specify all axioms manually as some axioms can be derived according to the automatic classification of instances using a DL reasoner. In addition, capturing types of data properties from the given domain ontology can reduce inconsistency of variable types in the CoCoViLa specifications and the corresponding Java classes. It is also possible to automatically propagate changes in ontology to a DSL i.e. capture the evolution of a domain. All these benefits were achieved due to using automated transformations from domain ontologies to the CoCoViLa specifications. However, limitations concern the fact that domain ontologies in OWL can represent more knowledge about the domain than our method can extract and use in the CoCoViLa extension as we have only limited scope for valid transformations (see Section 5.1).

- **A level of independence** of reusable knowledge from the internal representation of DSL meta-models was grown. The separation of different kinds of knowledge about the system, domain and a DSL into modular OWL ontologies makes the knowledge more reusable. Taking into account that the system ontology does not change often, it is rather easy to adjust the system to a new domain and build a DSL for this domain. It is also easy to build and change visual part of DSLs as the designer just needs to provide required URIs of visual objects as instances of the corresponding classes of the DSL meta-model ontology and the system automatically generates a new diagrammatic DSL.

In addition, it is possible to identify the following general advantages of the ontology driven DSL development process provided in this paper:

- DL reasoning services can be used for validation of domain models represented as formal domain ontologies. DL inference on DSL meta-model ontology is useful for debugging DSL meta-models.

- Support for the distributed DSL development process is provided by using distributed artefacts (e.g. images, multi-media resources, linked data etc.) linked to a DSL meta-model and used as components of a DSL.

Some of the limitations of the approach have been discussed above. In addition, the implementation of the approach is tightly related to the existing CoCoViLa system and language. However, general principles of the approach could be used by other DSL development tools.

8 Conclusions and future work

This paper demonstrates how ontologies can be semi-automatically integrated into the DSL development process. Representing domain models and the system model as OWL ontologies and linking them together to form a unified DSL meta-model ontology makes
it possible to semantically integrate software artefacts into the coherent DSL meta-model. In addition, distributed external artefacts can be integrated into the meta-model.

The main result of this paper is a systematic approach to a partial automation of the design and implementation phases of the DSL development process by using formal domain ontologies. We have prototypically implemented the approach as an extension to the DSL development tool CoCoViLa. In this paper, we have presented ontological structure of the DSL meta-model ontology, mapping rules from OWL to the CoCoViLa modelling language, and the implemented framework of the approach. In addition, we have presented a methodological process of semi-automated construction of DSL meta-models within the implemented framework and illustrated it with examples.

In this paper, we did not pay attention to semantic integration of artefacts from external tools and models into our framework. In principle, this is possible but the process requires the commitment to a common system ontology or availability of system ontologies of external tools.

Our future work will be related to combining rules represented in Semantic Web Rule Language (SWRL, http://www.w3.org/Submission/SWRL/) with domain ontologies. This will allow to model behavioural aspects (e.g. equations) of a domain. For this purpose we need to define and perform corresponding transformations from SWRL to relations in the CoCoViLa modelling language.

Acknowledgements

We are grateful to anonymous reviewers for their useful comments and to the Estonian Research Council for Institutional Research Grant IUT33-13.

References


