ISSN 1392–124X (print), ISSN 2335–884X (online) INFORMATION TECHNOLOGY AND CONTROL, 2014, T. 43, Nr. 3

Mapping Ontologies to Objects using a Transformation based on Description Logics

Rok Žontar, Ivan Rozman, Vili Podgorelec

University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova 17, SI-2000 Maribor, Slovenia
e-mail: rok.zontar@um.si, ivan.rozman@um.si, vili.podgorelec@um.si

crossref http://dx.doi.org/10.5755/j01.itc.43.3.4952

Abstract. To manage the increasing complexity of computer systems, a need has arisen to process knowledge instead of only data. Ontologies are nowadays widely used to describe domain knowledge, but although a high level of interest is present with researchers, the technology has not yet sufficiently been put into practice. We present an approach that addresses the transformation of abstract ontological concepts into everyday programming technologies in order to ease the development of semantic web applications for solving common engineering tasks. The presented formal mapping and its implementation - the MOOT framework - is an evolution in the field of ontology to object mapping. They rely on description logics to formalize the transformation process and allow for a detailed discussion about the entailed expressivity. We pay special attention to logical characteristics of roles in order to preserve as much expressivity as possible. Furthermore, an evaluation of the system is presented, where its performance and scalability is demonstrated.

Keywords: ontologies; object-oriented programming; description logics; mapping; performance evaluation.

1. Introduction

Computer systems are becoming increasingly complex due to both the growing number of users and their growing demand for functionality. Processors are more elaborate, memory systems are larger, networks are faster, and most importantly, the amount and complexity of data being used is overwhelming. This increasing complexity magnifies the already difficult task that developers face in implementing new technologies, designed to cope with emerging needs. As we face a paradigm shift towards a society of knowledge, the computerized processing of knowledge and the meaning of data is becoming one of the key aspects of computer engineering and software development.

The first step towards this was the formation of ontologies, which are considered as a formal, explicit specification of a shared conceptualization [1]. Their formal semantics allows for describing complex axiomatic structures of knowledge and are therefore believed to be a core enabling technology of the semantic web [2]. Although ontologies and ontology languages have proven to be very popular in the field of research, the engineering industry has yet to adopt them. Some studies attribute this to the lack of available tools [3], the undefined cost to benefit ratio [4] or current organizational cultures [5]. Taking effective use of semantic web technologies requires new skills that developers in the industrial environment generally do not posses. Because of lack of available tools to help them evolve, the question arises whether to tackle this by using complex ontology processing API's or to employ a mapping and ease access to semantic data, while risking some of their expressivity. We support the solution to transform some of the semantic web technologies into a well-known object-oriented environment. Furthermore, we believe that bridging the gap between abstract ontological concepts and everyday programming technologies would largely increase the adoption of ontologies in solving many common engineering tasks [6]. Thereby we focus on the developer's perspective by helping them to access semantically rich information in a familiar way.

The similarity between the ontological and object-oriented worlds [7] has inspired researchers to find new solutions on how to access semantic data. It is undisputed that ontologies have higher expressivity than the object-oriented paradigm [8] and that an object-oriented model can only be created by sacrificing some of its characteristics [9]. Our aim was to develop a mapping which would preserve much of the expressivity that is appreciated with ontologies and transform it to the world of object-oriented software systems. Thereby we base our transformation on the formalisms of description logics (DL), which form the logical foundation of the Web Ontology Language (OWL) [10]. In this paper we present a general model that enables the mapping of semantic web ontologies to object-oriented artefacts. A detailed discussion of the entailed expressivity is given with special attention for the support of different DL languages. An implementation of this model is provided in the form of the MOOT framework, which maps a subset of OWL 2 components to the programming language Java. It provides out of the box support for complex mappings of individuals to objects thereby employing an event-driven model to support logical characteristics like inverse, symmetric, reflective and transitive roles. We believe that supporting these characteristics is a vital factor in building a complete and expressive semantically supported software system. To determine the performance and scalability of the MOOT framework, an experiment is presented, where we evaluate the introduced framework.

2. Related work

Mapping ontology languages to facilitate their use has been researched from two perspectives. One approach treats ontology as a conceptual model or data schema and try to map it to its well-known equivalents in the form of the UML class diagram or Entity-Relationship model. In [11] an approach is presented where an ontological taxonomy is transformed into a relational database schema. A similar approach is presented in [12], where a graph oriented transformation is employed on OWL and transformed into an entity-relational schema. An advantage of this type of transformation is its ability to precisely define constraints against entities. However, the database schema has no native support for hierarchical structures, which need to be induced artificially. Furthermore, logical characteristics of roles are not addressed. Transformation of OWL ontologies into UML diagrams [13] is a similar approach, which can be employed in combination with the Model-Driven-Architecture initiative. Based on UML class models, constraints can be, for example, defined using rules [14] or the Object-Constraint Language (OCL) [15].

On the other hand, the direct mapping of ontological languages to source code has the advantage of quick adoption and a simplified transformation model [7]. One of the first such frameworks is Harmonia [16], which generates Java code for the JADE platform. Its model is unsophisticated and it does not support various features, e.g. multiple inheritance. The authors extended and refined their work in [17] where they defined a solid architectural foundation on which almost all future models build upon. We adopt their approach, by replacing generated functionality with an annotation based configuration model and supporting framework. ActiveRDF [18] is an adhoc framework for mapping RDF to the Ruby programming language. Some open-source projects are also available, like RDFReactor [19] and OWL2Java [20]. RDFReactor maps triples to an object-oriented model. The generated classes serve purely as a proxy for querying data. We employ a different strategy by creating a fully representative web of objects.

OWL2Java employs a similar mechanism as RDF-Reactor, but adds support for OWL. The Protégé ontology editor [21] is able to generate a simple Java API from an ontology. Compared to our system, the mapping depends on a generated vocabulary and does not support logical characteristics of roles. A further attempt to facilitate ontologies and object-oriented programming languages is presented in [22]. APIs a gogo define a domain specific language to tackle the complex mappings between ontologies and conceptual APIs. Their use of a model driven approach is unique and can only hardly be compared to our abstract transformation given below. Sapphire [23] is probably the most feature full framework for dealing programmatically with ontologies at the moment. In order to provide the necessary functionality it generates fragments of bytecode. We believe that this concept is not beneficial, because no additional functionality can be added to the domain model.

In contrast to most of the cited works, our model relies on a formal model defined using DL and a set notation. While most solutions provide language specific point-to-point mapping, we developed a universal model, which can be expanded to support multiple ontological- and programming-languages. The MOOT framework employs a simplified configuration model using annotations, which enable a simple integration and reuse of existing code. Besides that, we provide complete out-of-the-box support for relational inheritance and logical characteristics of roles without intrusive code, which focuses on developers by helping them access semantic data in a native an easy way.

3. Mapping ontologies to objects

The transformation of data from one form into the other is well known in the world of software engineering. Techniques like serialization and deserialization are already established as is the well-known object-relational mapping process. By defining the mapping from ontologies to objects we follow the aforementioned techniques and adopt their highly regarded features in the context of the Semantic web. We define the mapping of ontologies to objects as a two stage process. Phase one includes the transformation of abstract ontological concepts to constructs of an object-oriented programming language using generation of source code and its configurations. The second phase is responsible for performing the actual mapping of semantic data, represented in the form of individuals, into objects. To describe this process, we use DL to provide formal semantics and the set notation [24] to describe the components included in the transformation. DL is a family of knowledge representation formalisms based on a combination of frames and semantic networks [25]. Its main features include formal logic-based semantic and finite reasoning capabilities. Our model is based on the differentiation between the terminological TBox and axiomatic ABox parts. We can therefore define the ontology schema as a model (O_M), which consists of a terminological ($T\,Box_M$) and axiomatic part ($A\,Box_M$) analogue to DL:

$$\begin{aligned} O_M &\equiv < T\,Box_M, ABox_M > \\ T\,Box_M &\equiv < C, R_T, H, E, D_T > \qquad (1) \\ ABox_M &\equiv < I, R_A, D_A > \end{aligned}$$

The terminological model is a 5-tuple consisting of sets of concepts C, object relations R_T, hierarchy H, equivalence E and datatype relations D_T with the following definitions:

$$\begin{aligned} R_T &\equiv \{(a,r,b) | a \in C \wedge b \in C \wedge T \sqsubseteq \forall r^{-}.a \wedge T \sqsubseteq \forall r.b\} \\ H &\equiv \{(a,b) | a \in C \wedge b \in C \wedge a \sqsubseteq_d b \wedge a /= b\} \\ E &\equiv \{(a,b) | a \in C \wedge b \in C \wedge a \equiv b \wedge a /= b\} \qquad (2) \\ D_T &\equiv \{(a,v,t) | a \in C \wedge t \in DT \wedge T \sqsubseteq \forall v^{-}.a \wedge T \sqsubseteq \forall v.t\} \end{aligned}$$

On the other hand, the axiomatic part of the model is represented as a 3-tuple, consisting of individuals I , object relations R_A and datatype relations D_A, which we define as follows:

$$\begin{aligned} I &\equiv \{(x,a) | a \in C \wedge x : a\} \\ R_A &\equiv \{(x,r,y) | x \in I \wedge y \in I \wedge r \in R_T \wedge (x,y) : r\} \ (3) \\ D_A &\equiv \{(x,r,y) | x \in I \wedge v \in D_T \wedge (x,t) : v\} \end{aligned}$$

The abstract object-oriented model OO, which serves as the endpoint of the transformation is defined as follows:

$$OO \equiv < (I, C, H, R, V\,) > \qquad (4)$$

where I denotes interfaces, C concrete classes, H the hierarchy between interfaces, R the relations between the interfaces and V the variables.

Now let us define a function f which transforms the terminological model into the appropriate object-oriented one and the axiomatic data into objects, respectively. The function f is defined as follows:

$$f = \begin{cases} f_T \\ f_A \end{cases} \qquad (5)$$

where f_T is an injective function used to transform the terminological model into the object-oriented OO:

$$f_T : T\,Box_T \rightarrow OO. \qquad (6)$$

The surjective function f_A is defined analogously to f_T and is used to transform the axiomatic model into an object graph denoted as Obj:

$$f_A : ABox_M \rightarrow Obj. \qquad (7)$$

The described transformation functions provide only an abstract definition of the actual mapping phase, which is dependent on the chosen ontology language, object-oriented programming language and development platform. Therefore, each mapping has to specify details that rely on the expressivity of the programming language. The expressivity of transformation is presented in the next section, where we discuss the loss of expressivity regardless of a concrete programming language. A detailed description of the mapping process of the MOOT framework, which maps OWL individuals to Java objects, is given in Section 5.

4. Expressivity of transformation

In order to assess the expressivity preserved by the transformations, we must first examine the expressivity itself as it is defined in DL. As already mentioned DL is a set of language characteristics with well-defined semantics. Each set of characteristics adds some form of expressivity, which are joined to form languages. A DL language is a subset of characteristics, which forms a comprehensive frame for a particular task [25]. Some of the most discussed DL languages are frame languages $\mathcal{FL}_0$ and $\mathcal{FL}^-$ [26], attributive language $\mathcal{AL}$ and its extended version with complements $\mathcal{ALC}$ [27]. In the semantic web, DL languages form the basis for the web ontology language OWL, with $\mathcal{SHOIN}$ $(\mathcal{D})$ [28] as the logical foundation for OWL 1 and $\mathcal{SHOIQ}$ $(\mathcal{D})$ [29] as its equivalent for OWL 2. Table 1 presents a short overview of each language with their appropriate DL constructs. We thereby resort to the use of standard DL notation of symbols, where A and B denote atomic concepts, C and D complex concepts, R and S resemble abstract roles and V concrete or datatype roles.

To assess the ability of maintaining any level of expressivity, we must not examine each of the given languages. Programming languages are not often examined from the standpoint of expressivity, although some considerable differences exist among them. One has to consider many characteristics and try to minimize the expressivity loss when mapping from a more expressive ontological language to a less expressive programming language. This can be accomplished statically, by arranging classes in their correct hierarchical structure, or dynamically, by employing the realization of logical characteristics and constraints. We start our examination with the least expressive language and discuss whether each characteristic can be preserved.

4.1. $\mathcal{FL}_0$ and $\mathcal{FL}^-$

First, we would like to examine the expressivity of the frame language $\mathcal{FL}_0$. As it can be deduced from Table 1, $\mathcal{FL}_0$ consists of atomic concepts, roles, their intersection and the universal quantification. Considering transformation of atomic concepts to classes and roles to methods as trivial, we turn our attention to the intersection and universal quantification. The semantics of the intersection (Listing 1) can be modeled as a logical conjunction of two concepts. As such, the newly formed concept has all the characteristics of each concept occurring in the intersection. Translating this into the hierarchical structure of object-oriented code, a class needs to derive from both of the classes represented in the intersection. The final construct of

Table 1. Expressivity of different DL languages

Construct	Syntax	Languages
Atomic concept	A	
Role	R	
Intersection	$C \sqcap D$	
Universal quantification	$\forall R.C$	$\mathcal{FL}_0$
Limited existential quantification	$\exists R.T$	
Top concept	$\top$	$\mathcal{FL}^-$
Bottom concept	$\bot$	
Atomic negation	$\neg A$	$\mathcal{AL}$
Complex negation	$\neg C$	
Union	$C \sqcup D$	
Full existential quantification	$\exists R.C$	$\mathcal{ALC}$
Cardinality restriction	$\geq nR \leq nR$	
Nominals	$\{a_1, \dots, a_n\}$	
Role hierarchy	$R \sqsubseteq S$	
Inverse role	R^-	
Concrete roles	$\forall R.V$	$\mathcal{SHOIN}\ (\mathcal{D})$
Qualified cardinality restriction	$\geq nR.C \leq nR.C$	
Role inclusion axioms	$R \circ S$	$\mathcal{SHOIQ}\ (\mathcal{D})$

$\mathcal{FL}_0$ is universal quantification, which defines that the range of the given role consists only of individuals defined by a certain concept. As an independent construct, the universal quantification can be confidently transformed into a method using the given domain and range. The domain needs to be defined to identify the class in which the method resides and the range for the return type of the method. A problem can arise when one combines the universal quantification with an intersection, thereby defining a new concept:

$$Y\ oungResearcher \equiv Student\ n\ Researcher$$

$$GraduateStudent \equiv$$

$$Student\ n\ \forall takesCourse.GraduateCourse$$

The rules for transforming the intersection enable us to identify the GraduateStudent concept and transform it into a subclass of Student. On the other hand we cannot directly transform the universal quantification. Due to the limitations of modern object-oriented languages an overriding of methods is not possible in the return type of the method. So only the left part of the intersection will be transformed, while the universal quantification on the right will be ignored. This weakens the expressivity of the transformation due to the fact that we lose some information about graduate students which will not be inferred. Even though, this is not a huge setback, because even DLs need the support of a reasoner to infer this kind of knowledge.

The extended logic $\mathcal{FL}^-$ is basically $\mathcal{FL}_0$ extended with limited existential quantification and a top concept. The expressivity of the limited existential quantification is maintained by the transformation using the same strategy as with the universal quantification. We can confidently argue that the roles that have defined domain and range can be preserved by the mapping. An example is a Bachelor, who is defined as a person who has a bachelor's degree:

$$Bachelor \equiv P\ erson\ n\ \exists hasBachelorDegree$$

The top concept is used here to denote any possible individual. In order to incorporate this in the transformation, we need to introduce a new top class all others are derived from. This ensures that any object that represents an individual can be reduced to this basic type. This is similar to the root classes from modern object-oriented languages.

4.2. $\mathcal{AL}$ and $\mathcal{ALC}$

The attributive language $\mathcal{AL}$ is sometimes referred to as a minimal set of language characteristics, which is of practical interest [25]. It adds to the expressivity of $\mathcal{FL}^-$ by introducing the atomic negation of concept and the bottom concept, which is used to calculate inconsistencies. Atomic negation is a complex operation, which can be used by the transformation to create a class in the hierarchical structure of the represented ontology:

$$UnemployedPerson \equiv \neg\ EmployedPerson$$

The bottom concept has no direct transformation in the object-oriented world, because it is used only in the process of calculating inconsistencies [30] and has no real value when working with semantic data. Therefore, we can confidently dismiss it in the transformation procedure.

$\mathcal{ALC}$ is probably the most discussed of the DL languages [27]. It is obtained by adding further constructs to $\mathcal{AL}$. The union of concepts $\mathcal{U}$ and the full existential quantification $\mathcal{E}$ are the most prominent of them. Complex negation of concepts $\mathcal{C}$ is another feature, but as it has been proven it can be equally expressed using the union and the full existential quantification of concepts and vice versa [25]. Therefore the languages $\mathcal{ALUE}$ and $\mathcal{ALC}$ are semantically equivalent. Due to this theorem we use the union of concepts and the full existential quantification in order to determine whether our transformation preserves the expressivity of $\mathcal{ALC}$. The union of concepts can be regarded as a logical disjunction. In the class hierarchy, this is expressed as a class higher in the hierarchy. The transformation procedure takes this into account by placing the new concept as a superclass of both classes used in the union operation. So each individual representing either of the concepts will be automatically regarded as a union of them both. Next listing is an example of a union operation where it defines a faculty employee as either a professor, teaching assistant or a researcher:

$$FacultyEmployee \equiv$$
$$Professor \sqcup TeachingAssistant \sqcup Researcher$$

The full existential quantification is an extension to the limited existential quantification, already known from $\mathcal{FL}^-$. Earlier we argued that the transformation preserves the expressivity of the limited existential quantification due to the fact that no specific concept is declared as the range of the role. On the other hand, the full existential quantification allows specifying a concept in the range of a role. An example of this is shown below, where a master is defined as a person who received a Master's degree:

$$Master \equiv Person \sqcap \exists\ hasDegree.MasterDegree$$

This restriction allows for precise selection of individuals based on the relationships they have. In the object-oriented world, classes are used to describe the structure of an object. Although hierarchy is used to specify different types of objects, the base signature of a method cannot be predefined in a subclass. Therefore, a transformation cannot preserve this type of expressivity. In order to support full existential quantification, some other form of dynamic object transformation would be necessary.

4.3. $\mathcal{SHOIN}$ ($\mathcal{D}$) and $\mathcal{SHOIQ}$ ($\mathcal{D}$)

So far we have established a common basis for the discussion of specific semantic web logics. As we have shown, the transformation supports a wide variety of $\mathcal{ALC}$ constructs. It is not surprising that only some of the components are partially entailed in the transformation as most of them require a reasoner in order to support their full expressivity. This addresses the issue of dynamic classification of individuals, which is something most of the other mappings do not consider. In order to evaluate some of the non-terminological components of $\mathcal{SHOIN}$ ($\mathcal{D}$) and $\mathcal{SHOIQ}$ ($\mathcal{D}$), a different model is needed which manages the relations between objects at runtime.

The $\mathcal{SHOIN}$ ($\mathcal{D}$) [28] DL language is the logical foundation of the Web Ontology Language OWL DL. It provides the semantics for the more abstract ontological components and allows for a finite reasoning process. $\mathcal{SHOIQ}$ ($\mathcal{D}$) [29] is an extension of $\mathcal{SHOIN}$ ($\mathcal{D}$) developed to cope with the increasing complexity of OWL 2 ontologies. We will assess $\mathcal{SHOIN}$ ($\mathcal{D}$) and $\mathcal{SHOIQ}$ ($\mathcal{D}$) along each other due to their high amount of commonalities. Both of them are resembled by a set of logical characteristics described using the six letters. S is an extension of $\mathcal{ALC}$ by adding transitive roles. In order to support transitive roles one has to establish a supporting system, which ensures that the references between objects are indeed transitive. Some mappers employ a strategy of establishing these relations at the time of mapping and ignore them later on. Others generate source code, which ensures that the correct references are put in place. Although these solutions proved well, they also have a drawback due to the introduction of high amount of code in the domain classes. Therefore we prefer a non-invasive approach which is reflection based and hidden from the developer. The source code of the domain objects should remain unaltered and be easily configured by the developer.

The same strategy could also be applied to hierarchical roles $\mathcal{H}$ and their extensions in complex role inclusion axioms $\mathcal{R}$. Thereby it is necessary to distinguish between role characteristics that provide additional functionality (such as symmetry and reflectivity), from those that are of a restrictive nature (such as asymmetry and irreflectivity). While the former needs to support the addition of new references among objects according to the characteristics, the latter needs to prevent it. Role inheritance is a special feature within this set. It allows for defining roles, which are more specific than others. This is something unknown in the object-oriented world where polymerphism is the preferred choice for expressing specific relations. But at the same time, role inheritance is not problematic from the standpoint that each role has a unique name. Therefore, role inheritance can be successfully tackled using the previously mentioned solutions. Role inclusion axioms are a novelty introduced in OWL 2. In their current form they cannot be directly transformed in the object-oriented world, but an advanced dynamic model could be used as for other logical characteristics.

Nominals $\mathcal{O}$ represent a concept similar to enumeration in the object-oriented world. The main difference between those two is that nominals are formed by using instances, whereas enumerations are formed using primitive types. The use of nominals results in a highly coupled TBox and ABox. For this reason, their use is omitted in some of the sublanguages e.g. OWL Lite [31]. Because object-oriented languages do not allow to define classes based on their instances, it is only possible to map a nominal concept to a class. But at runtime it is impossible to verify whether the instance data match concrete objects.

Inverse roles $\mathcal{I}$, in contrast to symmetric roles, are not a typical logical characteristic because the operation involves two roles. The first one is the original role, while the latter one is its inverse. This enables the use of different concepts that act in the domain and range of the role, which is impossible to do using symmetric roles. As other logical characteristics, inverse roles could also be supported at runtime using a dynamic model.

Cardinality restrictions in its unqualified $\mathcal{N}$ as well as qualified $\mathcal{Q}$ form are an important part of the OWL language and are used to classify individuals similar to the universal and existential quantification. By default, roles cannot be restrained with a minimum or maximum number of instances. The only restriction available is to define new concepts which have certain cardinality constraint defined on a role:

$$Teacher \equiv Personn \sqcap \geq 1 teaches$$

$$Professor \equiv Personn \sqcap \geq 1 teaches.GraduateStudents$$

An exception of that applies to roles that are defined as functional and have their cardinality fixed at one. For any other form of cardinality the same restrictions apply as already mentioned by the full existential quantification. The mapping can support only the identification of a new concept in the hierarchy, but cannot dynamically classify objects without the use of inference.

It is important to highlight the good support the mapping offers for concrete roles ($\mathcal{D}$). But it is also worth noting that in order to take effective use of datatype roles one needs to precisely define the domain and range type of the role. If the range is not specified, the content could be considered as a character string. For most of the commonly used datatypes, corresponding object-oriented datatypes and mappings exist. The reason for this is the use of XML Schema for the OWL datatypes. It must be stressed that OWL does not impose restrictions on the cardinality of data roles. Thus, a role may appear zero, one or more times. Mapping this directly into the program code would cause ambiguity and further complicate the use of semantic data. For this reason and following the example of other tools mentioned in the related work, we suggest limiting the data characteristics to only one per role. This means that one could have a concept of a single instance of each data role.

To summarize the findings from the expressivity check, we can say that the proposed model provides good support for a large number of expressivity components. Unfortunately, we were not able to provide support for the full range of functionality that is prescribed by the different DLs. However, regarding the differences between the object-oriented programming languages and ontologies, that was not expected. Besides the quantification and cardinality restrictions, the most challenging obstacles are to provide the support for logical characteristics of roles, which must be established at runtime. Our research identified two distinct solutions, which seem prominent to tackle these challenges. One could establish the logical characteristics of semantic data within the mapping process of individuals to objects. An advantage of this solution is its simple implementation and effective query answering capability. But on the other side, it does not preserve the logical characteristics throughout the life cycle of the objects when manipulating them. To ensure full support for logical characteristics throughout the whole process, a solution is needed in the form of a dynamic model that instantly responds to any change in the relationship between two objects and adapt to it accordingly.

5. The MOOT framework

The encouraging results gained from the expressivity check lead us to the development of a framework, which implements the proposed strategies and follows the formal model presented in Section 3. We chose OWL 2 [32] as the source ontology language and Java as the target object-oriented language. OWL is the most widely used ontological language at the time of writing and has been chosen because of a wide variety of tools supporting the language. On the other hand, we chose Java, a modern object-oriented language, as the endpoint of the transformation. Although Java enforces some restrictions on the conceptual model (e.g. disallows multiple-inheritance) and thereby reduces the expressivity, we nevertheless address it because of its prevalence among developers of information systems.

The MOOT framework is comprised of two individual components, which tightly rely on each other to enable the generation of object-oriented source code and establish the mapping between ontological individuals and objects. To ease the development process, the system is comprised of a code generator (the MOOT Generator) and a factory (the MOOT Mapper) that performs the mapping itself. A model of the framework is presented in Fig. 1. The MOOT Generator is responsible for creating object-oriented source code based on the definitions of the ontological components, whereas the MOOT Mapper performs the actual mapping where individuals are loaded into the system, transformed into their corresponding object representation and populated with data.

Overall the framework relies on an ontology processing API and a reasoner. The first is used to process the ontology and the latter is used to provide additional information of the ontology. The architectural composition of the system allows for the addition of new APIs for managing ontologies. Further more, weak coupling between all components enables the support of different reasoning mechanisms. With the abstraction of the inference procedure, we were able to provide support for a wide range of different reasoning mechanisms and their implementations. The only requirement for each of them is that their implementation is compatible with the API used to process the ontology. By excluding programming

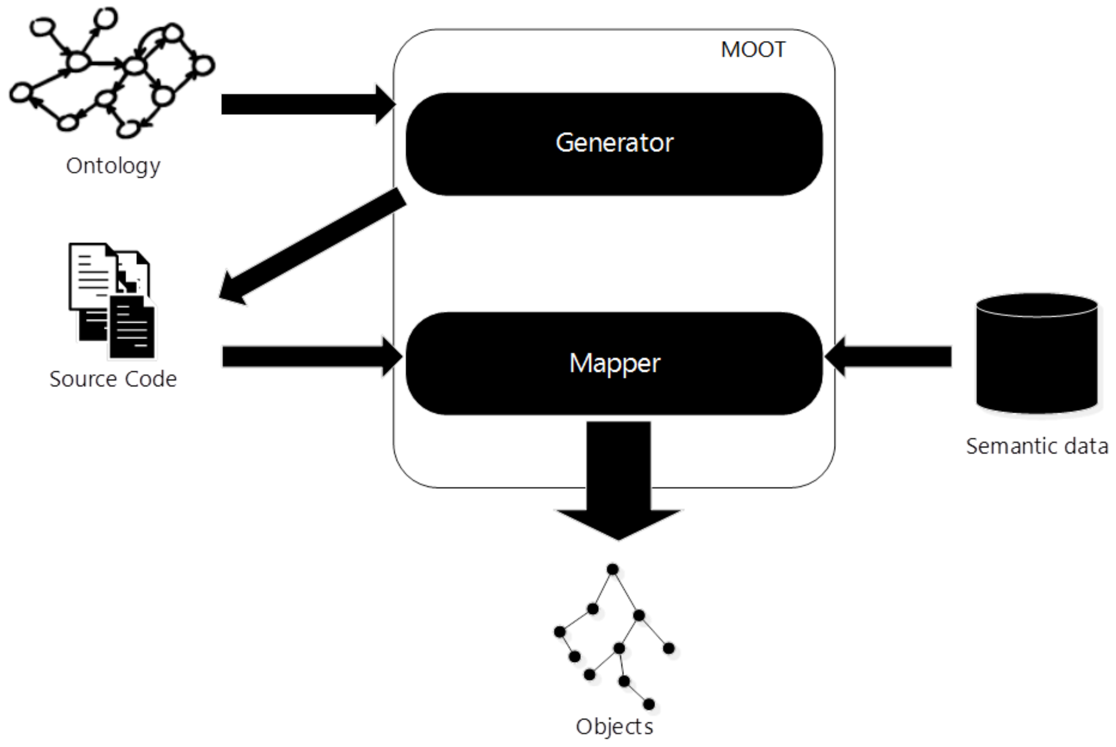

Figure 1. Model of MOOT framework

interfaces for handling ontologies and reasoning mechanisms, we were able to provide the possibility of using our framework also with the future versions of

We argue that the presented MOOT framework, which automatically maps ontology individuals by generating Java source code, conforms to f. The MOOT Generator is an implementation of f_T and the MOOT Mapper is an implementation of f_A, respectively. The framework combines the generated source code and its configuration using predefined annotations, which resemble key components of OWL. In this way, we kept the code clean and more readable. Moreover, the generated source code and its configuration can be added to existing source files and therefore enable the reuse of existing classes. If this is not needed, they are automatically added to the files generated from an ontology schema. The detailed description of the main components follows below.

5.1. The MOOT Generator

The MOOT Generator is responsible for creating object-oriented source code based on the definitions of the ontological components. The process begins with a user defined OWL ontology that is loaded into the system. Any valid OWL and OWL 2 ontology is supported and can be processed in various notations, e.g. RDF/XML, Turtle or Manchester. The ontology is then converted into the intermediate object model, which is a direct implementation of the formal model defined using DL and the set notation in Section 3. The intermediate object model consists of classes that represent concepts, object- and datatype-properties. Using these classes, an in-memory graph of interconnected objects is created, which is a direct representation of the ontology and serves as a hub in the process of transforming the conceptual ontological model into object-oriented source code. It allows for greater abstraction and facilitates data processing by eliminating the coupling between the code generation component and the API for processing ontological documents. From the intermediate object model, source code is generated using predefined template files. These are text files, defined using special markup, which is populated using actual data at runtime. Templates are used in order to achieve an expandable model, which on one hand supports several types of programming languages and on the other hand, in combination with the intermediate object model, supports different types of ontological languages.

The code generation component employs a well-established strategy [17] by creating interface-class combinations for each ontological concept. Each interface represents one concept, whose methods resemble roles in the ontological domain. Interfaces are used to support multiple inheritance, which is, in Java, disallowed with classes. Each interface has a class, which implements all of its defined and inherited methods. This allows hiding some of the implemented code and equips a developer with a clean and easy to understand class model. The static structure of the ontology is transformed using simple transformation rules, which are detailed in Table 2.

The initial implementation of the MOOT Generator did not include any support of inference mechanisms. This led to the problem of completeness, because the resulting source code was missing methods and the class hierarchy was incomplete. The reason why missing hierarchical links occurred was because we have built the object model on top of semantically weak data. Depending on the API used for processing ontological documents, it is sometimes not possible to derive the whole hierarchy of concepts and to identify the domains and ranges of roles. To solve this problem, we introduced a reasoner into the architecture of the system. This could be done without negative consequences to the performance, because it has been proven that the TBox reasoning is very effective [33]. Only by including ABox statements the complexity rises significantly. With an empty ABox, as it is in our case, the reasoner is used only to infer the entire hierarchical structure of the ontology and to identify the concepts that appear in the domain and range of roles. In this way, we ensure that the implicit ontological knowledge is included as well and that the acquired structural data of the ontology are complete. Although the use of a reasoner is encouraged, it is up to the end user to decide whether to use a reasoner or not. Furthermore, one can also choose among a multitude of supported implementations. For us, Pellet [34] was the reasoner of choice, because its reasoning engine is among the most widely used ones.

Fig. 2 presents a class diagram of a simple generated schema. One can identify the main interface `Student`, which implements the `OWLThing` top interface. The Student interface defines a pair of get-set methods to access a datatype role called `indexNumber` and a get method `takesCourse`, which is used to retrieve the related courses. The class `StudentImpl` is an implementation of the `Student` interface. It hosts all the implemented fields and access methods thereby following the POJO principle. As one can see, no additional code is being generated which is used to perform the mapping or provide any additional functionalities. This is all handled by the MOOT Mapper.

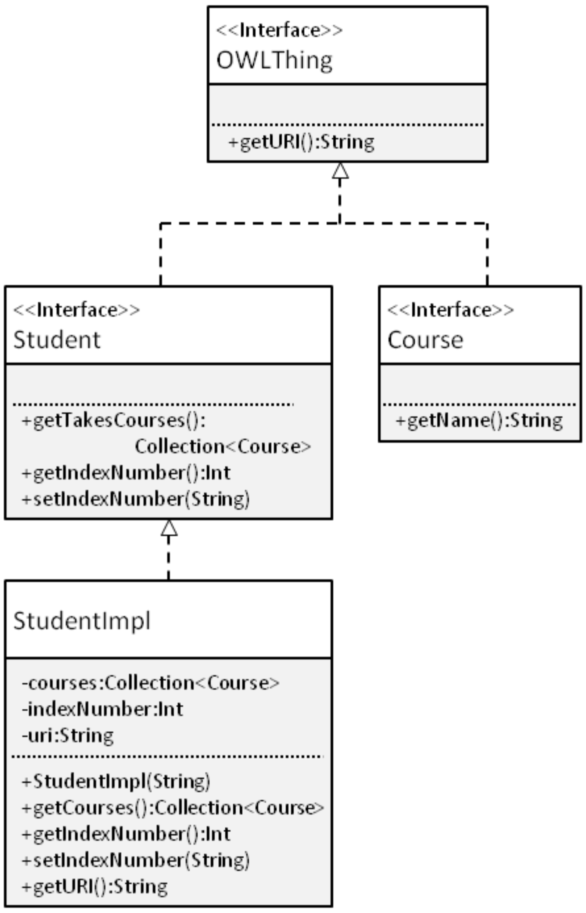

Figure 2. A class diagram of the generated source code

5.2. The MOOT Mapper

While the generation of source code usually takes place only once per ontology version, the second stage of the process is executed numerous times. This is the actual mapping where individuals are loaded into the system, transformed into their corresponding object representation, populated with data and returned to the user. In order for this transformation to happen, some kind of configuration mechanism must be put into place. It is necessary to provide an automatically generated baseline, which can be influenced by developers at the implementation phase. Modern methods, such as the conventions over configuration [35], have proven to be unsuitable due to the uncertainty of ontological URIs. Therefore, we needed to find a different solution to allow dynamic loading of data and mapping them to the correct class files. From the development of the software systems, mainly two solutions to this problem are used in the majority of cases. The first uses a text or XML file to store the configuration properties, while the latter stores the configuration in the form of code annotations. Regardless of the method, the reading and configuration management is a performance expensive and time-consuming operation. Therefore, we decided for the latter method, as it allows for a transparent way of combining source code and its configuration. Furthermore, this kind of configuration makes

Fig. 3 provides an example of an interface with attached annotations. The interface is identified by the `OWLClass` annotation which provides the URI of the concept. This is needed in order to identify the individuals belonging to the concept. The next annotation is an `OWLDatatypeProperty` annotation. It is used to identify datatype roles. There is only one annotation on the get method for each role. Naming conventions are used to find the correct method when setting values during the mapping process. Finally, the abstract roles are identified using an `OWLObjectPropery` annotation. Such roles may have multiple annotations attached to them depending on any special characteristics they provide. For example, the alumni role is defined as an inverse to the `graduateFrom` role as indicated by the `OWLInverseOf` annotation.

The actual process that utilizes annotations in order to map semantic data to objects is called materializa-

Table 2. Transformation rules from OWL and DL concepts to Java code

OWL	**DL**	**Java**
Concepts		
Top class	$\top$	Interface OWLThing
Bottom class	$\bot$	/
Class	C	I extends OWLThing, C implements I
Subclass	$C_1\ C \sqsubseteq C_2$	I_1 extends I_2
Equivalent classes	$C_1 \equiv C_2$	C implements I_1, I_2
Disjoint classes	$C_1 \sqcap C_2 \equiv \bot$	I_1 and I_2 define the same method with different return types
Intersection	$C_3 \equiv C_1 \sqcap C_2$	I3 extends I_1, I_2
Union	$C_3 \equiv C_1 \sqcup C_2$	I_1 extends I_3, I_2 extends I_3
Object properties		
Domain	$\top \sqsubseteq \forall R^{-}.C$	I has method R
Range	$\top \sqsubseteq \forall R.C$	R is of type Collection$< I >$
Equivalent property	$R_1 \equiv R_2$	R_1, R_2 use same backing field
Functional property	$\top \sqsubseteq \leq 1R$	R is of type I
Datatype properties		
Domain Range	$\top \sqsubseteq \forall V^{-}.C$	I has field R
	$\top \sqsubseteq \forall V^{-}.D$	V is of type D
Individuals		
Individual	$x : C$	x instanceof I

tion. Materialization is the process of manufacturing facilities described on the basis of semantic information. During this process the annotations are read using reflection. This is a process of reading data from the source code after it has already been compiled into byte code. The entire process takes place in several steps, which combines the query for semantic data and reflective operations to create and populate objects using it. Although the concept of materialization is defined a process that transforms an object from an abstract to a concrete form, in our case it is understood as a specific mapping of semantic data described into software objects. Fig. 4 shows the pseudo code for the process.

The procedure requires an ontology document, as well as annotated interfaces which resemble concepts as an input. For a number of selected concepts, all individuals are retrieved. For each of them, an object of the corresponding interface is dynamically generated, which is then populated using its datatype roles. From the technological perspective, ontologies use URIs to uniquely define concepts, properties and individuals. This prevents ambiguity between components that share the same name. The MOOT framework preserves URIs in order to precisely define the transformed component's name. If two or more components share the same name, they should be separated by different namespaces. In the current version of the framework, we support the following methods: (a) load single objects defined by a class and an individual's URI and (b) load all individuals of a given class or a collection of classes. The system also supports two techniques for loading related individuals. If individuals are mapped using deep loading, all related individuals are retrieved and their logical characteristics are applied, whereas when shallow loading is requested none of these are taken into consideration. To ensure that for each resolved URI the same object is returned, we keep an internal reference counter. This is also the time when observers, which realize sub-role relations and other logical characteristics, are registered.

In the implementation of the MOOT mapper, particular attention has been paid to the provisioning of logical characteristics of roles, which were identified as a crucial factor in maintaining a high level of expressivity. Our literature review has shown that the existing approaches (presented in the related work section) do not acknowledge them as an important part of ensuring the integrity of semantic data. We believe that inverse, symmetric, reflective and transitive roles are vital factors in building a complete and expressive semantically supported software system. To maintain a high level of expressivity, the relationships between objects need to physically exist in memory, and not only be expressed in some meta-data structure. In order to support this type of relationship we have resorted to the event model, which implements the observer design pattern [36]. In our case any collection serves as an observer and is observable at the same time. At the time of mapping, the role characteristics are read from the annotations and a complex network of references is being automatically put in place. The implemented event model ensures that after each change of a collection element all registered observers are notified and the change is automatically pushed across all relevant objects. This ensures that all the necessary relationships are present and can be easily navigated within the object graph. This has a lot of

Table 3. Annotations and their DL equivalents

Annotation	DL equivalent	Location
`@OWLAsymetricProperty()`	Asymmetric role	Method
`@OWLClass(String uri)`	Concept	Interface
`@OWLDatatypeProperty(String uri)`	Datatype role	Method
`@OWLDisjointWith(String uri)`	Disjoint concepts	Interface
`@OWLEquivalentClass(String uri)`	Equivalent concepts	Method
`@OWLFunctionalProperty()`	Functional role	Method
`@OWLInverseOf(String uri)`	Inverse role	Interface
`@OWLIrreflexiveProperty()`	Irreflexive role	Method
`@OWLObjectProperty(String uri)`	Role	Method
`@OWLReflexiveProperty()`	Reflexive role	Method
`@OWLSubPropertyOf(String uri)`	Role inheritance	Method
`@OWLSymmetricProperty()`	Symmetric role	Method
`@OWLTransitiveProperty()`	Transitive role	Method

```
@OWLClass(id="http://example.org/univ.owl#University")
public interface University extends Organization {
   @OWLDatatypeProperty(id="http://example.org/univ.owl#researchId")
   String getResearchId();
   void setResearchId(String value);

   @OWLInverseOf(id="http://example.org/univ.owl#graduatedFrom")
   @OWLObjectProperty(id="http://example.org/univ.owl#alumni")
   Collection<Person> getAlumni();
}
```

Figure 3. Example of an interface with annotations

```
Input: Annotated interfaces, Ontology document
Output: Objects
  for all concepts c from interfaces do
   load individuals of type c from ontology
   for all i in individuals do
      create object o for i of type c
      populate o with datatype roles
      if deep loading enabled then
        for all abstract roles where domain equals c do
          load abstract roles recursive
        end for
        register observers on roles
      end if
   end for
end for
```

Figure 4. Materialization procedure for mapping individuals to objects

advantages over a conventional system based on semantic triples, where roles are loaded for each query separately. We provide a support for irreflective and asymmetric roles in a similar manner by checking if a change to the collection's elements does not inflict any contradictory statements in the knowledge base. If such a state would be detected, the addition of this element to the collection is prohibited.

6. System evaluation

The performance of a system (or framework) and its scalability are crucial factors for its adoption in the software development process. Providing only functionality without the corresponding performance characteristic can result in a quick rejection. Our motivation for the experiment was to evaluate the system's performance, scalability and to see whether the MOOT framework can be effectively put into practice. In order to observe and to predict the behavior of the framework, we executed benchmarks and analyzed their results using standard statistical methods. To gain comparable results, we used the LUBM dataset and a benchmarking framework. The Lehigh University Benchmark (LUBM) [37] is well-established and a de-facto standard dataset for evaluating performance of semantic web technologies. In addition to an ontology, it provides a data generator, which ensures reproducible synthetic data of different sizes. The ontology itself describes a university environment, with fine grained concepts that include departments, professors, graduate- and undergraduate-students, courses and publications, as well as a variety of simple and complex roles between them. In our experiment we used the LUBM0 dataset generated using the default parameters. It consists of 15 files with approximate 100,000 triples and about 17,000 individuals.

We evaluated the performance by measuring the execution time of tasks, which are most frequently used. These tasks include: (i) the start-up phase; (ii) the mapping of single individuals; and (iii) the mapping of all individuals. The first, or start-up, task is where the mapping factory is created and the ontology model is loaded. The second task uses the mapping factory to load single individuals. The third stage maps all individuals using shallow and deep loading.

To achieve statistically reliable and comparable results, we used a micro benchmark framework discussed in [38]. It measures the time of a task, while ensuring that external influences can be neglected. Each task is first executed once (First measurement), after that the same task is executed until 60 repeated executions (Mean measurement) yield statistically insignificant differences. In this manner, it ensures the standard deviation (*SD*) and confidence interval (*CI*) to be within the tolerance threshold. All benchmarks were executed on a workstation with an Intel E8400 Core2 Duo processor and 8GB of DDR3 RAM. The operating system used in the experiment was Windows 7 and virtual machine Java SE version 7u9 with 4GB of dedicated memory. The version of the OWL API [39] used to process ontologies was 3.3.

6.1. Start-up phase

In the start-up or initialization phase we measured the execution time the framework takes to initialize all factories and load the ontology document. The results are presented in Fig. 5, where we plot the execution time in relation to the size of dataset. The top higher values represent measurements from the first execution, while the lower represents the mean values of 60 consecutive measurements. It can be undoubtedly recognized that the performance varies depending on the size of the dataset. Although the first executions proved to be significantly slower than their means, it must be noted that standard deviation (*SD*) of measurements was always below 5%. The extreme difference between the first execution and the rest, which can reach to a factor of 12, can be, according to [40], generally regarded as usual and brought back to virtual machine optimization.

It must be noted that although each file sizes only a few hundred KB, the combined dataset achieves a size of almost 8MB, which has an impact on the performance. While digging deeper, we normalized the elapsed time by the number of loaded individuals. Fig. 6 presents the normalized mean data value in relation to the size of dataset. Each measurement is calculated by dividing the mean time by the number of individuals of the dataset. Interestingly, its values first rapidly decline and then stabilize following an inverse logarithmic curve. This allows us to predict the time

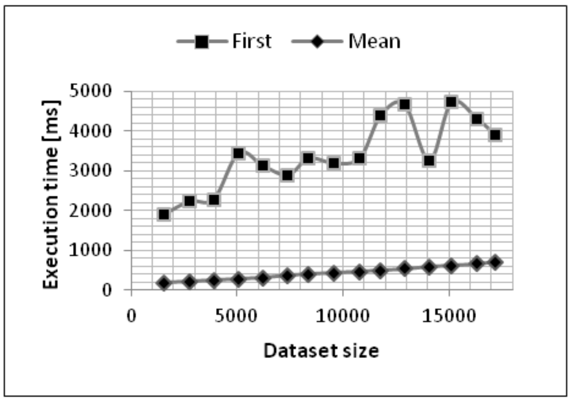

Figure 5. Execution time taken to initialize the framework in relation to the size of dataset

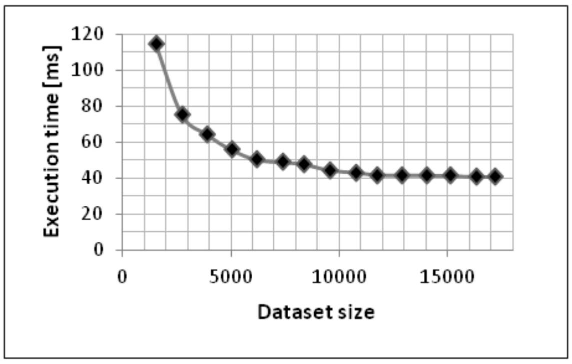

Figure 6. Time taken to initialize the framework divided by the number of individuals in each dataset

necessary to complete the start-up phase for large datasets.

Further examinations proved our hypothesis that the performance at this stage mainly depends on the time the supporting ontology framework needs to load the ontology model. In our case, this is accomplished using the OWL API and we do not have the ability to influence it at all.

6.2. Loading single individuals

The level of performance for mapping single individuals constitutes a very important aspect of the system's overall performance. Ideally, one would want a predictable behavior, which is independent from the type of concept and the dataset size. Therefore, we evaluated the performance by loading individuals from different concepts and compared them using shallow and deep loading.

The gathered data lead us to the conclusion that shallow mapping of single individuals was indeed independent from the size of the dataset. The system achieved values, which deviated less than 5%, independently from the size of the dataset. On the other hand, different concepts proved to have very diverse execution times. An example of this is presented in Fig. 7, where the mapping times of two concepts are plotted. The concept UndergraduateStudent averaged a time of 273ms (*SD*: 6ms, *CI*: ±3ms), while, on the other hand, the concept Publication averaged a value of 80ms (*SD*: 2ms, *CI*: ±1ms). While investigating this phenomenon, we discovered that the times correlate with the number of datatype properties of a given concept. When we compared the times with deep loading, we received very diverse results. This was due to the different numbers and types of child individuals that had to be loaded. Therefore no statistically relevant data can be provided.

6.3. Loading single individuals

The final test was comprised of mapping all individuals for a given dataset. Although the MOOT framework does not support the mapping of individuals from all concepts simultaneously, we achieved this by retrieving all individuals from all concepts consecutively. The results displayed in Fig. 8 clearly resemble a linear increase of execution times as the dataset sizes increase. Also, the clear separation between shallow and deep loading is present here more than ever.

Regardless of the difference, we were able to determine a linear regression function with a coefficient of data. This allows us to draw precise predictions for the behavior of the system, even for large datasets. The difference between an individual mapped shallow and deep was averaged at a factor of 5.5 with a *SD* of 6%. After further investigation, we discovered that much of the difference can be brought back to reading annotations and creating collections, while reading role values from datasets, and virtual machine optimization played only a minor difference.

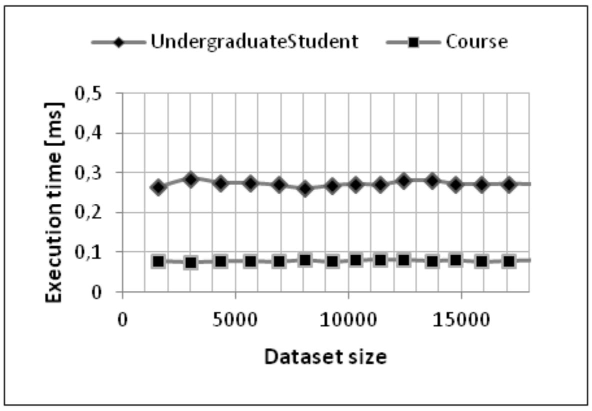

Figure 7. Execution times to map single individuals in relation to the size of dataset

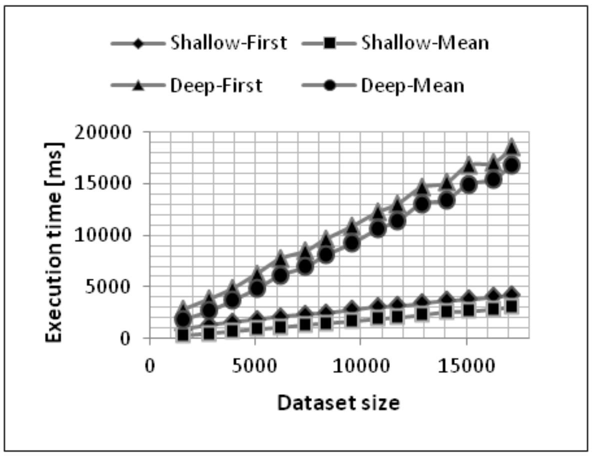

Figure 8. A comparison between the execution times of deep and shallow loading

7. Conclusions

Providing developers with a simple programming model is an important aspect in the development of semantic web technologies. Existing systems have laid a solid architectural foundation for the mapping of ontological concepts to object-oriented programming models, but mostly ignoring expressivity features like logical characteristics of roles. We addressed this by formally defining a transformation based on description logics which allows efficient mapping of ontologies to objects while preserving much of the expressivity. In this manner, an expressivity check has been performed to assess the transformations ability to handle complex DL constructs. Based on this, the MOOT framework was implemented that maps OWL 2 ontologies and individuals to Java components and objects, respectively. In our belief, the presented framework hides the complexity of ontologies and aligns the access to semantic data to what developers are used to. Our framework thereby reduces the gap between ontological systems and everyday programming environments. To evaluate the framework performance, benchmark tests have been performed and their results analyzed.

The acquired results show that the introduced framework confidently handles data independently from a given dataset. Depending on the dataset size, the start-up performance varies, but mapping single individuals has proven to be unaffected by that. The variations between different concepts can be drawn back to the number of datatype roles that need to be set. To map whole sets of individuals, a clear linear regression function was determined, which allows for predicting execution times in the future. Future work on the framework will include support for some reasoning features and optimization of the mapping process, especially the extensive use of reflection related operations gives us room for further improvements. Although, in our opinion, the trade-off between performance characteristics and a simplified method of accessing knowledge already clearly swings in favor of the later.

Acknowledgments

This paper was produced within the framework of the operation entitled "Centre of Open innovation and ResEarch UM". The operation is co-funded by the European Regional Development Fund and conducted within the framework of the Operational Programme for Strengthening Regional Development Potentials for the period 2007–2013, development priority 1: "Competitiveness of companies and research excellence", priority axis 1.1: "Encouraging competitive potential of enterprise and research excellence".

References

[1] **T. R. Gruber.** A translation approach to portable ontology specifications. *Knowledge Acquisition,* 1993, Vol. 5, No. 2, 199-220.

[2] **T. Berners-Lee, J. Hendler, O. Lassila.** The semantic web. *Scientific American*, 2011, pp. 29-37.

[3] **V. Janev, S. Vraneš.** Applicability assessment of semantic web technologies. *Information Processing & Management*, 2011, Vol. 47, No. 4, 507-517.

[4] **E. Della Valle, G. Niro, C. Mancas.** Results of a survey on improving the art of semantic web application development. *In: The 10th International Semantic Web Conference, Bonn, Germany*, 2011.

[5] **T. Pellegrini, A. Blumauer, G. Granitzer, A. Paschke, M. Luczak-Rösch.** Semantic web awareness barometer 2009 – comparing research- and application- oriented approaches to social software and the Semantic Web. *In: Proceedings of I-KNOW'09 and I-SEMANTICS'09, Graz, Austria*, 2009, pp. 518-529.

[6] **V. Podgorelec, M. Grešak.** Supporting the Study Process using Semantic Web Technologies. *Electronics and Electrical Engineering,* 2011, Vol. 116, No. 10, pp. 105-108.

[7] **R. Žontar, M. Heričko.** Adoption of object-oriented software metrics for ontology evaluation. *In: Proceedings of the Fifth Balkan Conference in Informatics, Novi Sad, Serbia,* 2012, pp. 298-301.

[8] **P. Bartalos, M. Bieliková.** An approach to object-ontology mapping. *In: IIT, SRCâĂŞStudent Research Conference, Bratislava, Slovakia*, 2007, pp. 9-16.

[9] **C. Puleston, B. Parsia.** Integrating object-oriented and ontological representations: a case study in Java and OWL. *In: International Semantic Web Conference, Karlsruhe, Germany,* 2008, pp. 130-145.

[10] **F. Baader, I. Horrocks, U. Sattler.** Description Logics as Ontology Languages for the Semantic Web. *Lecture Notes in Artificial Intelligence,* 2005, Vol. 2605, pp. 228-248.

[11] **E. Vysniauskas, L. Nemuraite.** Transforming ontology representation from OWL to relational database. *Information Technology and Control*, 2006, Vol. 35, No. 3, 333-343.

[12] **J. Trinkunas, O. Vasilecas.** A graph oriented model for ontology transformation into conceptual data model. *Information Technology and Control,* 2007, Vol. 36, No. 1, 126-132.

[13] **J. Trinkunas, O. Vasilecas.** Ontology transformation: From requirements to conceptual model. *Computer Science and Information Technologies*, 2009, Vol. 751, 52-64.

[14] **O. Vasilecas, D. Kalibatiene, G.Guizzardi.** Towards a formal method for the transformation of ontology axioms to application domain rules. *Information Technology and Control,* 2009, Vol. 38, No. 4, 271-282.

[15] **D. Kalibatiene, O. Vasilecas.** Application of the Ontology Axioms for the Development of OCL Constraints from PAL Constraints. *Informatica*, 2012, Vol. 23, No. 3, 369-390.

[16] **D. J. Pastor, J. Padget.** Towards HARMONIA: automatic generation of e-organisations from institution specifications. *In: 3rd Workshop on Ontologies and Agent Systems, Melbourne, Australia,* 2003, pp. 31-38.

[17] **A. Kalyanpur, D. J. Pastor, S. Battle, J. Padget.** Automatic mapping of OWL ontologies into Java. *In: 16th International Conference on Software Engineering and Knowledge Engineering, Banff, Canada,* 2004, pp. 98-103.

[18] **E. Oren, R. Delbru, S. Gerke.** ActiveRDF: Object-oriented semantic web programming. *In: Proceedings of the International World-Wide Web Conference, Banff, Canada,* 2007, pp. 817-823.

[19] **M. Völkel.** RDFReactor – From Ontologies to Programatic Data Access. In: *Poster Proceedings of the Fourth International Semantic Web Conference, Galway, Ireland*, 2005, pp. 55-60.

[20] **M. Zimmermann.** Owl2Java. [Online], http://www.incunabulum.de/projects/it/owl2java.

[21] **Stanford University.** The Protégé Ontology Editor and Knowledge Acquisition System. *[Online]*, http://protege.stanford.edu/

[22] **F. Parreiras, C. Saathoff.** APIs a gogo: Automatic generation of ontology APIs. *In: International Conference on Semantic Computing, Berkeley, USA,* 2009, pp. 342-348.

[23] **G. Stevenson, S. Dobson.** Sapphire: Generating Java runtime artefacts from OWL ontologies. *In: Proceedings of the Advanced Information Systems Engineering Workshops, London, UK,* 2011, pp. 425-436.

[24] **G. Meditskos, N. Bassiliades.** CLIPS-OWL: A Framework for Providing Object-Oriented Extensional Ontology Queries in A Production Rule Engine. *Data & Knowledge Engineering*, 2011, Vol. 70, No. 7, 661-681.

[25] **F. Baader, D. Calvanese, D. McGuinness, D. Nardi, P. Patel-Schneider.** The description logics handbook. *Cambridge University Press*, 2003.

[26] **R. J. Brachman, H. J. Levesque.** The Tractability of Subsumption in Frame-Based Description Languages. *In: Proceedings of the 4th National Conference on Artificial Intelligence, Austin, USA*, 1984, pp. 34-37.

[27] **M. Schmidt-Schauß, G. Smolka.** Attributive concept descriptions with complements. *Artificial Intelligence*, 1991, Vol. 48, No. 1, 1-26.

[28] **F. Baader, I. Horrocks, U. Sattler.** Description logics as ontology languages for the semantic web. *Springer, Berlin–Heidelberg,* 2005.

[29] **I. Horrocks, O. Kutz, U. Sattler.** The Even More Irresistible SROIQ. In: *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning, Lake District, UK*, 2006, pp. 57-67.

[30] **F. M. Donini, M. Lenzerini, D. Nardi, A. Schaerf.** Reasoning in description logics. *Principles of Knowledge representation*, 1996, Vol. 1, 191-236.

[31] **D. L. McGuinness, F. van Harmelen.** OWL web ontology language overview. *W3C recommendation, [Online],* http://www.w3.org/TR/owl-features.

[32] **B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, U. Sattler.** OWL 2: The next step for OWL. *Journal of Web Semantics*, 2008, Vol. 6, No. 4, 309-322.

[33] **C. Lutz, U. Sattler, L. Tendera.** The Complexity of Finite Model Reasoning in description logics. In: *Proceedings of the 19th International Conference on Automated Deduction, Miami Beach, USA*, 2003, pp. 60-74.

[34] **E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, Y. Katz.** Pellet: A practical OWL DL reasoner. *Journal of Web Semantics*, 2007, Vol. 5, No. 2, 51-53.

[35] **N. Chen.** Convention over configuration. [*Online*], http://softwareengineering.vazexqi.com/files/convention_over_configuration.pdf.

[36] **G. Erich, H. Richard, J. Ralph, V. John.** Design patterns: elements of reusable object-oriented software. *Addison Wesley, Reading*, 1995.

[37] **Y. Guo, Z. Pan, J. Heflin.** LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 2005, Vol. 3, No. 2, 158-182.

[38] **B. Boyer.** Robust Java benchmarking, Part 1: Issues. *IBM, [Online]*, http://www.ibm.com/developerworks/-java/library/jbenchmark1/index.html.

[39] **M. Horridge, S. Bechhofer.** The OWL API: A Java API for OWL Ontologies. *Semantic Web Journal*, 2011, Vol. 2, No. 1, 11-21.

[40] **B. Goetz.** Java theory and practice: Anatomy of a flawed microbenchmark. IBM. [Online], http://www.ibm.com/developerworks/java/library/jjtp02225/index.html.

Received December 2013.