F O U N D A T I O N S O F C O M P U T I N G A N D D E C I S I O N S C I E N C E S
Vol. 43 (2018) No. 4
ISSN 0867-6356
DOI: 10.1515/fcds-2018-0019 e-ISSN 2300-3405

Universal Framework For OWL2 Ontology Transformations

Bogumiła Hnatkowska*, Paweł Woroniecki

Abstract. Domain ontologies are valuable knowledge assets with many potential applications, e.g. in software engineering. Their content is often a subject of bi-directional transformations. Unfortunately, a centralized transformation service which can be easily extended with new mappers is not available for ontology users. In consequence, they have to deal with many different translation programs, which have to be installed and learned separately. The paper presents a framework for universal ontology processing, dedicated to ontologies expressed in OWL2. The framework usefulness was verified by a proof-of-concept implementation, for which an existing OWL2 to Groovy translator was adapted. During the integration process, the translator functionality was enhanced with ontology individuals mapping. The exemplary implementation confirmed that the framework with plug-in architecture is flexible and easy for customization. The ontology stakeholders should benefit from the reduced cognitive load and more satisfying transformation process.

Keywords: OWL2, ontology, framework, transformation, translation, mapping, Groovy

1. Introduction

Ontologies are one of the approaches to the reuse and sharing of knowledge. As reusable assets, they have many possible practical applications, also in the area of software engineering, e.g. for domain knowledge extraction [6], database design [19], or source code generation [17]. They can be effectively used in all stages of software development, from requirement specification through analysis, design and implementation to testing. Such applications of ontologies typically require the knowledge included in the ontology to be translated to the proper representation, e.g. UML [6]. On the other side, ontologies can be built or updated by retrieving interesting data from existing artefacts, e.g. source code or UML diagrams [22].

There exist many tools for processing of ontologies, i.e. for edition, validation, reasoning, and transformations, e.g. Protégé [11], or Sigma Knowledge Engineering Environment [27].

* Faculty of Computer Science and Management, Wrocław University of Science and Technology, ul. I. Łukasiewicza 5, 50-371 Wrocław, Poland, Bogumila.Hnatkowska@pwr.edu.pl

Unfortunately, transformation capabilities are offered as standalone, closed solutions, which typically cannot be easily updated or extended with new functionalities. In consequence, a potential user has to download, possibly install and/or configure, and manage many tools to perform particular transformations. To address this inconvenience and to enable additional capabilities, e.g. synchronization mechanisms, authors implemented a universal framework for ontology transformations (called OWL2 Universal Processor Framework, OWL2 UPF).

The framework is understood as a "skeleton of interlinked items which supports a particular approach to a specific objective, and serves as a guide that can be modified as required by adding or deleting items" [23]. Here the interlinked items are framework components, the specific objective is bi-directional ontology transformation, and the framework user can add new mappings, represented as separate components or delete/disable existing ones. The framework itself coordinates the transformation process by calling enabled translations. The purpose of OWL2 UPF is to "simplify the development environment, allowing developers to dedicate their efforts to the project requirements, rather than dealing with the framework's mundane, repetitive functions and libraries" [24].

The framework supports ontologies written in the OWL2 Web Ontology Language (shortly OWL2) [25]. OWL2 uses different syntaxes to exchange knowledge among tools and applications. The primary exchange syntax is RDF/XML, however also other are possible, e.g. OWL/XML or OWL Functional [25]. OWL2 was chosen because of two main reasons:

- OWL2 popularity (about 103000 hits in Google Scholar for a question "OWL2 ontology" against about 11600 hits for "SUMO ontology", where SUMO is declared to be the biggest freely available set of ontologies [13]).
- The framework is an extension to the previous authors' work connected with OWL2 [7].

The rest of the paper is organized as followed. Section 2 presents the genesis of the framework and related works. Section 3 describes the assumptions and main architectural decisions together with the API required by framework extensions. An example of framework application for OWL2 to Groovy transformation is demonstrated in Section 4. Section 5 concludes the paper.

2. Related works

"Ontologies are formalized vocabularies of terms, often covering a specific domain and shared by a community of users. They specify the definitions of terms by describing their relationships with other terms in the ontology" [25]. Therefore, ontologies often are a subject of transformation process being its source or target element – see Fig. 1.

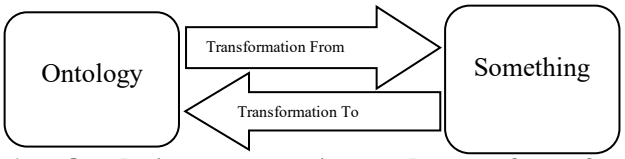

Figure 1. Ontologies as a source/target element of transformation

Transformations in which ontology is a source element are often used in software engineering. Ontology axioms can be transformed to:

- UML class diagram, e.g. [6], [9]
- source code representing domain classes and relationships among them, e.g. Java [17], Java with Enterprise Java Beans and Hibernate [2], Groovy [7], Cool OO language used further by CLIPS rule engine [10]
- source code with unit test, e.g. [12]
- SQL schema, e.g. OWL2toRDB [19] or OWLMap [1].

On the other side, ontologies can be built from other artefacts, e.g.

- natural text, e.g. [18]
- relational database, e.g. DataMaster [14], and OntoBase Protégé plug-ins [21] for importing schema structure and data from relational databases communicated with JDBC/ODBC driver
- UML, e.g. an Eclipse plugin using ATL language for transformation purposes [5], an application in which XSLT transformations are used to convert XMI files with UML models into OWL (XML format) [3], or a Protégé UML Backend plugin [9]
- XML document, e.g. [4].

The tools mentioned above are typically run as separate applications or plugins to Eclipse or Protégé. Every person who wants to use such a tool has to download it, configure (if necessary), become familiar with application interface which can be significantly different than in similar solutions. Only a few from the tools support bi-directional transformations, e.g. [19] (OWL2 – relational database translations).

Examples of desktop applications, distributed as jar files are [6], and [7]. The former, written in Java 8, supports translations from SUMO to UML, the latter – written in Groovy – performs translations from OWL2 to Groovy. The Groovy translator was adapted to be a part of a universal framework being able to integrate existing solutions for OWL2 ontologies processing.

3. Framework architecture

3.1. Architecture drivers

"A software framework is a concrete or conceptual platform where common code with generic functionality can be selectively specialized or overridden by developers or users. Frameworks take the form of libraries, where a well-defined application program interface (API) is reusable anywhere within the software under development" [24]. However, frameworks offer some features that make them different from libraries, e.g. [24], [16]:

- Non-modifiable framework code – the core functionality of the framework that cannot be modified and often is distributed in the form of a library, e.g. a jar file.
- Extensibility – the way in which a user can extend the offered framework behavior, e.g. by writing own subclasses.

- Inversion of control – application of the Hollywood Principle according to which the framework sends messages to the user-defined classes, not the opposite.

3.2. Architecture definition

Framework design involves description of its static structure and dynamic behavior [16]. The static aspect mostly concentrates on the meta-model(s) used by the framework. However, it also specifies other important framework components, how they are related, and what they are responsible for. UPF framework components are described in Section 3.2.1, while one of its meta-models in Section 3.2.2. The definition of dynamic aspect answers the question how the framework functionality is provided by the components the framework is built from (see 3.2.3 for further details).

Another crucial part of the framework design is the integration model which determines how the framework is to be used by its clients. There are two types of integration: black-box, and white-box. A white-box "requires clients to supply new subclasses first, before objects can be created and composed", while in a black-box "a client (object) can use the framework by instantiating classes and composing the instances to suit its needs" [16]. The first type of integration model can be realized via inversion of control mechanisms. The second one is typical for libraries. Many tools combine both integration models (the same holds for UPF). The white-box integration model applied in the framework is described in Section 3.2.4.

Inversion of control is often used with dependency injection mechanisms served by specific environments, e.g. Spring. Description of the injection mechanisms used in the proposed solution is given in 3.2.4 together with the integration model.

The UPF needs another client component which calls the framework services for translation purposes (black-box integration). An exemplary implementation of a universal client is presented in Section 3.3.

Selection of the proper implementation platform for framework realization is important as it will have a significant influence on the framework popularity and further development. On the one side, such a platform should enable flexible modular or component architecture, easy to be extended with new translators transparently – no need for a difficult installation or configuration. On the other side, the platform should be successful enough for potential contributors, and let them choose a preferred programming language.

Two implementation environments were considered for framework realization, i.e. OSGi [28], and Spring [29]. Both are open-source, free solutions. The first is "a set of specifications that define a dynamic component system for Java [28]", what means that the components can be replaced or upgraded on the fly, without restarting the system. The components, represented by so-called bundles (plugins), communicate through services. The services have to be registered in the service registry to be effectively used by other bundles (bundles may depend on each other). The components do not need to be written only in Java; it depends on the OSGi implementation, e.g. Apache Felix [30]. What is more, Protégé requires its plugins to be realized as OSGi bundles.

Spring is an application framework providing inversion of control (IoC) container with a flexible dependency injection (DI) mechanism; another important feature is support for Aspect-Oriented Programming. Additionally, many useful Spring modules are present, allowing to, e.g., build complex web applications, give easier access to a database, implement

batch data processing. It is possible, the future direction of UPF extensions, will require these features.

To ensure loosely coupled components in the UPF, DI was necessary. Plugin mechanism was important as well – to support translators. Both considered environments provide required features. The decision has been made to choose Spring. It offers better DI than OSGi with more possibilities like beans scope, relations between beans, integration with logging frameworks, supporting configuration through external files. On the other hand, OSGi provides better support for plugins mechanism. However, Spring beans and loading all jar files from a given directory (with translators) were enough to implement plugins in the UPF, so OSGi was not necessary in this case.

3.2.1. Framework components

Decomposition of the UPF framework on components is shown in Fig. 2. The general rule was that the components should be loosely coupled, with well-defined interfaces, having only one area of responsibility.

The framework serves for universal bi-directional translations of OWL2 ontologies. It is why it uses two types of meta-models which form the backbone of the entire solution. The first meta-model represents the structure of OWL2 ontology and is represented by *org.semanticweb.owlapi.model.OWLOntology* component (see [8]). It is an external component on which the framework depends. The second meta-model represents the structure of the translation input/output. It is represented by the *CommonCodeMetamodel* component. The framework delivers or consumes a set of files (binary or textual) contained within specific folders as the result or input to the translation procedure. The content of this meta-model is described in the next section.

The main processing elements are translation engines. There is one for OWL to code (*OWL2ToCodeTranslationEngine*), and one for code to OWL (*CodeToOWL2Translation-Engine*) mappings. "Code" in this context means anything which can be represented by a set of folders and files, e.g. source code, UML XMI files, binary files (e.g. png, jpg), etc. The common code meta-model used for the representation of an input/output (other than OWL2) of transformation procedure is the only element shared by all translators. It is assumed that its instance will be mapped into/from an instance of the internal meta-model, more specific for a given translator, e.g. Groovy meta-model, Java meta-model, UML meta-model etc. Each translation engine offers the *translate* method, which iterates over the list of registered translators to perform the mapping procedure between OWL2 meta-model and its internal meta-model. It should be mentioned that within a singular session one ontology can be transformed to many different representations, and one can produce many different ontologies from different sources.

Each specific translator is expected to implement one-directional mapping. The *translate-CodeToOntology* and *translateOWL2Ontology* are abstract methods, implementation of which the framework extensions need to deliver. The first takes an instance of common code metamodel (a set of elements: files and folders), and on this basis produces an instance of the *OWLOntology* class. The second method does the opposite – takes as an input an instance of the *OWLOntology* class and produces its representation in terms of a set of elements (files and folders) which can be later written to a specified destination. The design of translators' API is flexible – it specifies required output and necessary input but does not force a concrete

realization. The translator's developer can decide, e.g., to process ontology axioms sequentially or in parallel.

The *Owl2MetamodelUtils* component wraps auxiliary classes for ontology reading and writing with the use of external libraries (marked in grey). This component is to be used by the framework universal client (see Section 3.3).

The *Logger* is another important framework component. It can be used by specific translators to log processing information, warnings and/or translation errors. Its detail description is given in Section 3.2.4.

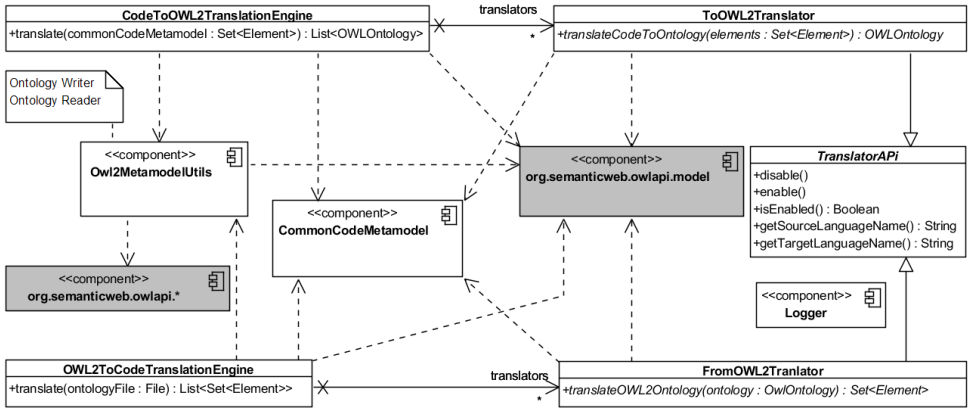

Figure 2. UPF framework architecture

3.2.2. Common code meta-model

It is assumed that all specific translators will use the same common code meta-model (see Fig. 3 for details) to represent the input/output to the translation process. The abstract class *Element* is the root class in the meta-model. It represents a named element and is further refined by two subclasses: *Folder* and *File*. The *File* is an abstract template class, parametrized by *T* type used for a description of the file content. The parameter is further instantiated to String (for text files), and array of bytes (for binary files). The *Folder* class is equipped with a set of operations for manipulation of the folder content – the semantics of some was formally defined with the use of Object Constraint Language (OCL). The common code meta-model also contains some auxiliary classes (marked with grey) which support reading/writing of the meta-model content.

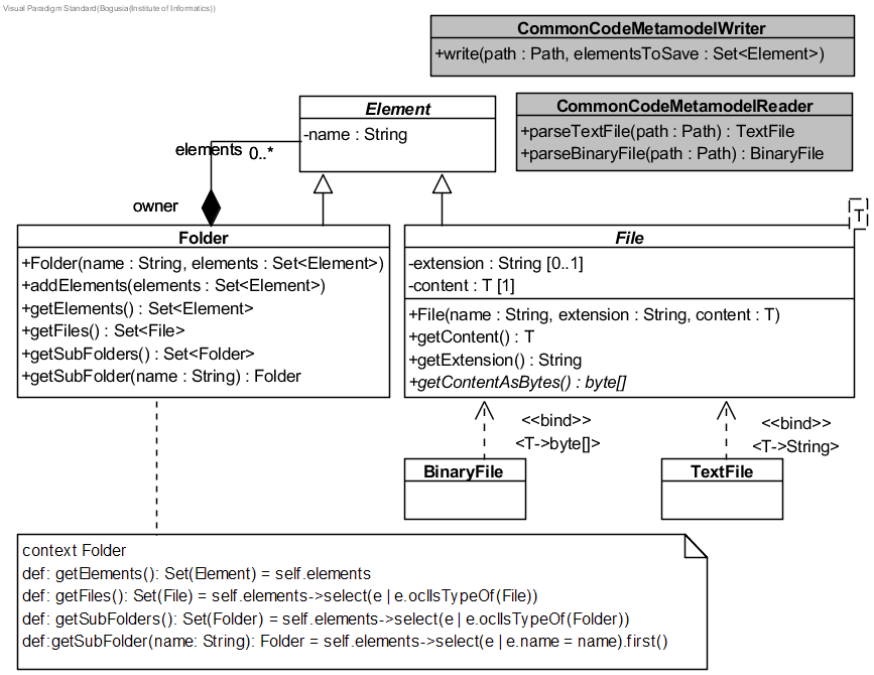

Figure 3. Common meta-model class diagram

3.2.3. Framework behavior

The UPF framework intended behavior can be represented in a readable manner with the use of a sequence diagram. For the simplicity, Fig. 4 presents one-way transformation process. A client has to create an instance of the *OWL2CodeTranslationEngine* class and call its *translate* method passing an ontology file as an argument. That starts the translation procedure in which for each registered and enabled translator its *translateOWL2Ontology* method is called. The translator engine gathers the translation results, which later are returned to the client. The client can further process the results in any way, e.g. by writing them to a predefined path.

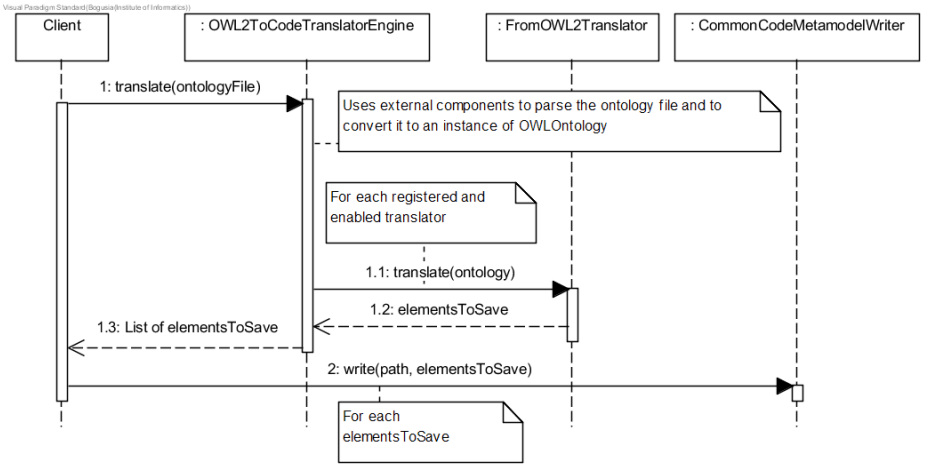

Figure 4. OWL2 to "code" translation process

The translation procedure in the opposite direction, i.e. to OWL2 ontology, is very similar. Again, the client has to instantiate a proper engine (now *CodeToOWL2Translation-Engine*) and call its *translate* method passing a set of elements (folders, files) as an input. The elements can be read with the *CommonCodeMetamodelReader* class. Next, the engine calls the *translateCodeToOntology* method for each registered translator, gathers the translation results (OWL2 ontologies) and return them to the client in a list. The client can write ontologies with the use of the *OntologyWriter* class from *Owl2MetamodelUtils* component.

3.2.4. Integration model

The UPF framework provides several core components. However, its full utilization requires implementation of two component types: translators and clients. They are used in the way presented in Section 3.2.3.

All the required elements can be delivered in numerous ways. This section presents the recommended approach. It is based on the assumption that each translator and each client is implemented as a separate module (which can be compiled to a jar file). This way they are loosely-coupled and can be modified without affecting other components.

Delivery of a new translator requires implementation of one of the interfaces: *FromOWL2Translator* or *ToOWL2Translator*, dependently of the direction of transformation. The translator should introduce all necessary notions, e.g. used meta-model(s) to represent source/target language on its own (an example is described in Section 4.1).

To keep the components loosely-coupled and to enable the injection mechanism, it is assumed that implemented translators will be visible to the clients as beans. This can be achieved, e.g. by adding *Spring Context* dependency ([26]) in the translator module. This dependency enables application of *@Component* annotation directly on the translator class, which marks the translator as a Spring-managed component.

It is strongly advised to use one common root package name (e.g. *owl2converter*) for all implemented modules, especially for translators. This way the clients may look for beans in the given root package and are guaranteed to find all translator beans (present in the class path) during component scan phase.

Implementation of a client depends heavily on its purpose, e.g. it can be created specifically to work only with selected translators (in particular one) or to work with any number of translators. Regardless of the number of translators the client is designed to work with, it must be prepared for UPF integration. The client is expected to have a specific Spring configuration file, which imports other configurations provided by the core of the UPF framework, e.g. *CommonCodeMetamodelConfiguration* (to manipulate instances of common code meta-model), *OWL2MetamodelConfiguration* (to manipulate instances of OWL2 ontology) etc.

The client can disable (by calling *disable*() method) or enable (by calling *enable*() method) some of the injected translators. It can also check the current translator state by *isEnabled*() method.

The UPF framework also provides a logging mechanism. It is recommended to inject a *Logger* bean instance in clients and translators and to use it to log performed operations and/or their effects, e.g. translation errors. Default logging mechanism can be adjusted, e.g. by providing custom logging configuration.

Logging mechanism is based on the SLF4J interface (https://www.slf4j.org/). Logger instances are injected with IoC mechanisms to the proper classes. Logback (https://logback.qos.ch) was selected as the implementation of the SLF4J API used by the UPF framework. Any other SLF4J implementation may be applied to create *Logger* bean instead of Logback. This requires overriding the default *Logger* bean in the custom Spring configuration.

3.3. Universal client

To make the UPF framework easier to run, it is delivered with an implementation of a universal client. The universal client is a simple Java form-based application which uses framework components to perform transformations selected by a user. The client assumes that implementations of specific translators (jar files) are gathered in *translators* folder. It automatically, using the reflection mechanism, loads all existing translators, which are instantiated as Spring beans, and later – on the basis of user selection – enables only one translator and uses it for transformation. The user can select an existing ontology file in the case of OWL2 to source code translation or define a new ontology name being the target for the opposite translation. The user can also select a folder containing artefacts to be converted to OWL2 or in which the transformation results will be stored – see Fig. 5.

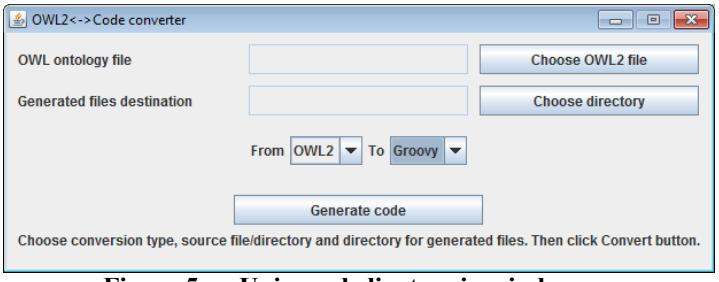

Figure 5. Universal client main window

4. Framework application example

4.1. OWL2 to Groovy translator

Domain models (a set of classes described by attributes and properties reflecting relationships between classes) can be derived automatically from ontology axioms. It speeds up the development process as the reasonable part of the source code could be generated, and it is written using more friendly-syntax from programmers' point of view.

To perform a proof-of-concept application of the UPF framework, the OWL2 to Groovy translator presented in [7] was adapted as one of the translators which are under the framework control. The translator was rewritten to meet the UPF contracts. It consists of three primary components:

- engine – a set of classes responsible for intermediate translation results, including translation between OWL2 and Groovy meta-models as well as source code generation,
- meta-model – a container for classes of Groovy meta-model, used by the engine component to represent OWL2 to Groovy translation results (abstract syntax),
- result codebase – a folder reflecting the structure of the resulting source code; it contains subfolders with a constant content, e.g. *exceptions* subfolder with a definition of all exceptions used by Groovy code, or *owlSimpleType* subfolder with Groovy classes representing its OWL2 equivalents like *Language*, *Token*, *Name* etc.

The OWL2 to Groovy translator was integrated with the UPF framework and can be run with the use of the universal client.

An extract of a Groovy source code resulting from the translation of the following OWL2 ontology fragment (see Listing 1) is shown on the listing 2.

```
<Declaration> <ObjectProperty IRI="#adjacentRegions"/>
</Declaration>
<Declaration> <Class IRI="#Region"/> </Declaration>
<SymmetricObjectProperty>
      <ObjectProperty IRI="#adjacentRegions"/>
</SymmetricObjectProperty>
<ObjectPropertyDomain>
```

```
      <ObjectProperty IRI="#adjacentRegions"/>
      <Class IRI="#Region"/>
</ObjectPropertyDomain>
<ObjectPropertyRange>
      <ObjectProperty IRI="#adjacentRegions"/>
      <Class IRI="#Region"/>
</ObjectPropertyRange>
```

Listing 1. Exemplary ontology (fragment)

The ontology contains only one *Region* class, and one object property called *adjacentRegions*. The property is symmetric one. This is directly reflected in Groovy code, see e.g. methods *addAdjacentRegions*, *removeAdjacentRegions*, which implementation ensures the proper instances to be linked on both sides (see Listing 2).

```
trait RegionTrait implements ThingTrait {
  private static Set<RegionTrait> allInstances = [].toSet()
  private List<RegionTrait> adjacentRegions = []
    …
  void addAdjacentRegions(RegionTrait adjacentRegions) {
    if (adjacentRegions == null
      || this.adjacentRegions.contains(adjacentRegions))
      { return }
    this.adjacentRegions.add(adjacentRegions)
    adjacentRegions.addAdjacentRegions(this)
  }

  void removeAdjacentRegions(RegionTrait adjacentRegions) {
    if (adjacentRegions == null
      || !this.adjacentRegions.contains(adjacentRegions))
            { return }
    this.adjacentRegions.remove(adjacentRegions)
    adjacentRegions.removeAdjacentRegions(this)
  }
}
```

Listing 2. *RegionTrait* – the partial result of OWL2 to Groovy transformation process

In comparison to OWL2 to Java transformation, present in Protégé tool [11], the generated API (interface) to manipulate *Region's* instances is similar. However, the default implementation of *Region* interface in Java does not take into consideration that *adjacentRegion* is a symmetric property. Nothing is checked or assured.

4.2. Transformation of OWL2 individuals

OWL2 ontology often contains individuals being instances of a particular ontology class. Individuals can be described by data properties and/or object properties. In the context of objected-oriented language, individuals can be represented by objects. Attributes of these

objects may be partially filled if there are proper axioms determining attributes' values in the ontology.

Transformation of OWL2 individuals to code can be useful in various cases. An example can be an OWL2 ontology describing genetics – structures of genetic code fragments and what influence they have on health and behavior of a subject (a man or an animal). Individuals in the ontology represent specific genes, e.g. blood group gene. Such ontology can be used as both: a model (classes and relationships between them) and a database (individuals) for different kind of software, e.g.:

a) Software to DNA analysis to find potential diseases on the genetic background. The DNA code may be retrieved from a person and given as an input to the software. The input is then processed to detect health risks. The more individuals in the ontology (gene code fragments), the better the analysis results.
b) Software for scientists to compare genetic codes between different species, e.g. which one is more immune to the specific diseases and why.

There may be many software programs using the same knowledge (and hence the same ontology), but implemented to provide different functions. Moreover, they can store data internally in different ways, e.g. SQL database, NoSQL database or even flat files.

Mapping of individuals is a stage of OWL2 to the Groovy translation process. To serve instances three classes are always generated:

- *IndividualWrapper* – a wrapper class containing a name of the OWL2 individual and its instance (object),
- *IndividualRepository* – a repository class holding instances of *IndividualWrapper* and providing access to them by their names,
- *IndividualsDataSource* – a class with one factory method responsible for creation of instances, setting values of their attributes, and adding these instances to *IndividualRepository*.

The content of *IndivualWrapper* and *IndividualRepository* classes is constant. On the other hand, the content of *IndividualsDataSource* is dynamically generated based on axioms found in the translated ontology.

Mapping of a *DataPropertyAssertion* axiom requires setting the attribute value of the related object to the value specified by an OWL2 literal. Table 1 presents non-trivial mappings of OWL2 literal types to types implemented in the translator, based on Groovy native types extended with additional validation reflecting original OWL2 semantics.

Table 1. Mapping between OWL2 literal and Groovy types

OWL2 (XML) type	Groovy type	Native Groovy type
normalizedString	NormalizedString	No (wrapper of String)
token	Token	
language	Language	
NMTOKEN	NMTOKEN	
Name	Name	
NCName	NCName	
nonPositiveInteger	NonPositiveInteger	No (wrapper of BigInteger)
negativeInteger	NegativeInteger	
nonNegativeInteger	NonNegativeInteger	
unsignedLong	UnsignedLong	
unsignedInt	UnsignedInt	

unsignedShort	UnsignedShort	
unsignedByte	UnsignedByte	
positiveInteger	PositiveInteger	
dateTime, time, date, gYearMonth, gYear, gMonthDay, gDay, gMonth	XMLGregorianCalendar	Yes

Mapping of an *ObjectPropertyAssertion* axiom is realized by establishing appropriate references between object instances. Instantiation of individuals must be realized in the proper order. This problem is solved with the topological sorting algorithm defined for a directed acyclic graph [15]. This algorithm enables the translator to detect cyclic dependencies, but it does not handle them – an exception is thrown in this case. This problem will be resolved in the next version of the program. A solution to this problem seems to be quite straightforward. First, all the individuals will be created as default objects with the use of the default constructor, later values of their attributes will be set by with public setters.

The usage of individuals mapped from an OWL2 ontology in the source code typically is reduced to invocation of the *initializeIndividualsRepository* method from the *IndividualsDataSource* class and then retrieving created objects from *IndividualRepository* with *getIndividual* method, accepting individual's name as a parameter. Names of all individuals can be accessed through another method from the repository – *getIndividualNames*. The proper usage of the above mentioned elements will be illustrated with a simple example. Listing 3. presents a fragment of an ontology containing individuals.

```
ClassAssertion(<#Box> <#box1>)
ClassAssertion(<#Box> <#box2>)
ClassAssertion(<#Producer> <#producer>)
ClassAssertion(<#Product> <#product>)

ObjectPropertyAssertion(<#aggregates> <#box1> <#box2>)
ObjectPropertyAssertion(<#producedBy> <#product> <#producer>)
DataPropertyAssertion(<#productName> <#product>
  "Product"^^xsd:string)
```

Listing 3. Exemplary ontology with individuals (fragment)

The ontology declares four individuals: *box1*, *box2*, *producer*, *product* where *box1* is related with *box2* by *aggregates* assertion, and *product* is related with producer by *producedBy* axiom. Moreover, *productName* property is set for *product*. These individuals are transformed to Groovy objects in the way presented in Listing 4.

```
private static IndividualWrapper createBox2() {
    return new IndividualWrapper('box2', Box.createInstance())
}

private static IndividualWrapper createBox1() {
    Box individualInstance = Box.createInstance()
    individualInstance.with {
      setAggregates([IndividualRepository.getIndividual('box2')
```

```
            .individualInstance])
    }
    return new IndividualWrapper('box1', individualInstance)
  }

  private static IndividualWrapper createProducer() { … }

  private static IndividualWrapper createProduct() {
    Product individualInstance = Product.createInstance()
    individualInstance.with {
      setProducedBy([IndividualRepository.getIndividual('producer')
            .individualInstance])
      setProductName('Product')
    }
    return new IndividualWrapper('product', individualInstance)
  }

  static void initializeIndividualsRepository() {
    IndividualRepository.addIndividual(createBox2())
    IndividualRepository.addIndividual(createBox1())
    IndividualRepository.addIndividual(createProducer())
    IndividualRepository.addIndividual(createProduct())
  }
```

Listing 4. Part of *IndividualsDataSource* class containing transformed individuals

Individuals are instantiated in the proper order, e.g. *box2* before *box1* and *producer* before *product*. Product's name is also set to the value specified by OWL2 literal ("Product").

IndividualWrapper class has been chosen to make it easier to distinguish objects created by the transformation of ontology from objects created directly in the client's software, e.g. by developers. This distinction may be useful in some cases, e.g. to find a source of potential errors. Sometimes this wrapper class is unnecessary, but in such a case each wrapper may be transformed to its enclosed object.

Alternatively, *IndividualWrapper* could be ignored, so *IndividualRepository* would contain pure individual objects. It could be extended with new helper methods like a method to get individual's name for a given object. However, a dedicated structure like a wrapper with the individual name contained in the wrapper attribute seems to be less error-prone and more efficient – there is no need to iterate over all individuals in repository to find an object equal to the given one just to find the name.

To actually initialize *IndividualRepository* it is required to call the static method *initializeIndividualsRepository* from *IndividualsDataSource* class. Afterwards, the repository is ready to use. The repository provides four public and static methods:

- *getIndividual(String name)* – returns an instance of *IndividualWrapper* created for individual with given name,
- *getIndividualNames()* – returns a set of all individual names stored in the repository,
- *addIndividual(IndividualWrapper individual)* – adds an individual to the repository, where the individual is provided as a wrapper object; the method prevents from storing duplications (individuals with the same name) by throwing

an exception in such case; this method is used by *IndividualsDataSource* and is likely not to be applied manually by developer,

- *addIndividual(String name, Object individualClassInstance)* – a helper method which creates a wrapper's object using the provided individual's name and then works in the same way as the previous *addIndividual* method.

A simple example of repository usage is presented in Listing 5.

```
1 void sampleRepositoryUsage() {
2   IndividualsDataSource.initializeIndividualsRepository()
3   Set<String> allIndividualNames =
4       IndividualRepository.getIndividualNames()
5   List<IndividualWrapper> allWrappers = allIndividualNames.collect
6   {
7     IndividualRepository.getIndividual(it)
8   }
9   List<Object> allIndividualObjects =
10      allWrappers.collect { it. getIndividualInstance() }
11
12  // assuming we know there is box1 individual that should be
13  // an instance of Box class
14  Box box1 = IndividualRepository.getIndividual('box1')
15      .getIndividualInstance(Box)
16  Box box1Again = (Box) IndividualRepository.getIndividual('box1')
17      .getIndividualInstance()
18
19  String someIndividualName = allWrappers.first().getName()
20 }
```

Listing 5. Sample usage of *IndividualRepository* class

There are several methods used in Listing 5. First of all, the repository of individuals is initialized (line 2) as described earlier. Then the names of all individuals are fetched from the repository (lines: 3-4). Afterward, instances of *IndividualWrapper* are retrieved from the repository with the *getIndividual* method providing individual's name (lines: 5-8). The lines 9-10 present how to retrieve a concrete object from the wrapper (call of the *getIndividualInstance* method of the *IndividualWrapper* class), so it is used to transform a list of wrappers into a list of concrete objects. There is also another version of *getIndividualInstance* (lines: 14-15) with the argument being an expected type of the individual's object and returning object cast to this type. If the type is incorrect then an exception is thrown. This way the *Box* instance is retrieved directly and no cast is needed. The same effect can be achieved with non-argument *getIndividualInstance* method, and manual cast (lines: 16-17). At the end of Listing 5., there is an example of how to retrieve an individual's name from its wrapper with the *getName()* method (line 19).

Although most axioms related to individuals are supported in OWL2 to Groovy translator, some of them are not supported yet. Among supported axioms, there are *ObjectProperty*, *InverseObjectProperty*, *DataProperty*, *Literal*. The following axioms are not supported at the moment: *SameIndividual*, *DifferentIndividuals*.

Individual equality (axiom *SameIndividual*) can be supported in at least two ways.

One possibility is to create a single object for all equal individuals and add it to the *IndividualRepository* under each of their names. Querying such repository for any of the

individual's names returns the same object, so it fulfills the equality condition – one individual can be replaced with another equal individual.

Alternatively, many objects could be created – one per "the same" individual. All of them must be instances of the same class with equal values of all attributes. In this case, the class should implement the *equals* and *hashcode* methods to compare values of all attributes. Thanks to that all objects representing the same individuals will be equal. However, it is important to consider mutability. Each change in the state of the object must be synchronized with all equal objects. To achieve this, each of these objects could implement an observer pattern to be notified of the change in the state of its siblings and to be able to synchronize it.

Both solutions seem to be consistent with the semantic of *SameIndividual* axiom although they are different in terms of object-oriented programming. The former (one object for all equal individuals) is much easier to be implemented and less error-prone, so this transformation will be probably chosen.

Unique name assumption for individuals can be axiomatized in OWL2 ontology by usage of *DifferentIndividuals* axiom. Without this axiom we cannot conclude if two individuals are the same or different. It would be tough to reflect this ambiguity in OO language's code. Therefore, unique name assumption for individuals is present by default in OWL2 to Groovy converter. Hence axiom *DifferentIndividuals* is not going to affect the transformation and so will be ignored.

5. Summary and further works

The paper presents the architecture of the UPF framework for OWL2 bi-transformations. The proof-of-concept implementation of the framework, including OWL2 to Groovy translator, confirmed that the architecture is feasible and easy to be extended with new translators. The architecture is flexible allowing plug-in of any number of mappers. The issue of errors that may occur during processing is also addressed. Errors can be logged e.g. in text files, separately by each translator. The ontology user should benefit from the reduced cognitive load (one installation instead of many) and more satisfying transformation process (the same universal client can be used for all translators). The framework implementation is available under the link [20].

At that moment there is only one translator adapted for the framework API. However, implementation of the other one is in progress. The correctness and completeness of a new translator is a challenge which realization takes some time. The translation rules must be elaborated and thoroughly tested before the translator publication. This is the primary reason why the authors have checked the UPF usefulness for only one plug-in. Nevertheless, it can be said with full conviction that the framework serves as a center which helps in management of OWL2 transformation capabilities with the following benefits one can observe:

- Standardization: The framework standardize the way the translators are built and the way they can be called; the translators can also be used outside the framework, e.g. by a web service. The side effect of standardization is a potential for concurrent development of mappers by different teams.
- Easy installation of new translators: The whole solution has a component architecture with well-defined interfaces used for communication; installation of a new translator

is nothing more than placing a jar file in the proper place in the folder structure; the framework will recognize and utilize it automatically.

- Reusing of functionalities/configuration: Basic functionalities (reading/writing instances of OWL2 meta-model, and common code meta-model, logging mechanisms) are reused. Some of these functionalities are supported by existing libraries which configuration could be time-consuming – the framework enables their application without any additional activities (however, the existed configuration can be changed).
- One end-user tool instead of many: Instead of having many different tools (translators) separately installed there exists one universal client (a kind of system overlay) collaborating with the UPF framework to provide translation capabilities.

The OWL2 to Groovy translator has also been extended concerning its previous version (see [7]) with the transformation rules for OWL2 individuals. The proposed solution is similar to [11], where ontology individuals are represented indirectly as instances of *WrappedIndividualImpl* class. In OWL2 to Groovy translator, individuals are represented by instances of *IndividualWrapper* class and managed by *Individual Repository* singleton instance.

The UPF framework is dependent on Spring platform. It is assumed that translator modules are deployed as beans visible to the framework clients. It is a kind of limitation but not too much as translators can be implemented in any language processed by JVM machine. Authors tried to port the solution to the Protégé tool, however without success. The OSGi implementation used in Protégé would require rework of majority parts of the existing framework implementation.

The UPF framework can be extended with synchronization mechanisms for round-trip engineering. Now, the synchronization, if any, must be done manually. Another abstraction layer can also be introduced to allow processing of other than OWL2 ontologies.

The OWL2 to Groovy translator – an add-in to the framework – can also be improved. The expected feature is a configuration file, which settings could control the transformation process, e.g. define if to add or not comments representing OWL2 axioms in Groovy code to improve its readability.

References

[1] Afzal H., Waqas M. and Naz T., OWLMap: Fully Automatic Mapping of Ontology into Relational Database Schema. (IJACSA) International Journal of Advanced Computer Science and Applications , vol. Vol. 7, no. No. 11, 2016, 7-15.

[2] Athanasiadis I. N., Villa F. and Rizzoli A.-E., Ontologies, JavaBeans and Relational Databases for enabling semantic programming, in 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), Beijing, China, 2007.

[3] Gasevic D., Djuric D., Devedzic V., and Violeta D., Converting UML to OWL ontologies, in Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW Alt. '04). ACM, New York, NY, 2004, 488-489.

[4] Hacherouf M. and Bahloulb S. N., DTD2OWL2: A New Approach for the Transformation of the DTD to OWL, Procedia Computer Science, vol. Vol. 62, 2015, 457-466.

[5] Hillairet G., ATL Use Case - ODM Implementation (Bridging UML and OWL). SIDo Group from the L3I lab in La Rochelle, 2007. http://www.eclipse.org/atl/usecases/ODMImplementation/. [Accessed 5 April 2018].
[6] Hnatkowska B., Automatic SUMO to UML translation, e-Informatica Software Engineering Journal, vol. 10, nr 1, 2016, 51-67.
[7] Hnatkowska B. and Woroniecki P., Transformation of OWL2 property axioms to Groovy. In Tjoa A., Bellatreche L., Biffl S., van Leeuwen J., Wiedermann J. (eds) SOFSEM 2018: Theory and Practice of Computer Science. SOFSEM 2018. Lecture Notes in Computer Science, Vol. 10706, Edizioni della Normale, Cham, 2018, 269-282.
[8] Horrige M. and Bechhofer S., The OWL API: A Java API for OWL ontologies. Semantic Web, Vol. 2, Issue 1, 2011, 11-21.
[9] Knublauch H., Case Study: Using Protege to Convert the Travel Ontology to UML and OWL. In: Evaluation of Ontology-based Tools, 2nd International Workshop, Sanibel Island, Florida, USA, 2003.
[10] Meditskos G. and Bassiliades N., CLIPS-OWL: A framework for providing object-oriented extensional ontology queries in a production rule engine, Data & Knowledge Engineering, Vol. 70, Issue 7, 2011, 661-681.
[11] Musen, M.A., The Protégé project: A look back and a look forward. AI Matters. Association of Computing Machinery Specific Interest Group in Artificial Intelligence, 1(4), June 2015. DOI: 10.1145/1557001.25757003
[12] Nasser V., Du W., and MacIsaac D., An ontology-based software test generation framework, in: The 22nd International Conference on Software Engineering and Knowledge Engineering, SEKE, San Francisco Bay, USA, 2010.
[13] Niles I. and Pease A., Towards a Standard Upper Ontology, in: Proc. of the International Conference on Formal Ontology in Information Systems - Volume 2001, FOIS'01, ACM, New York, USA, 2001, 2-9.
[14] Nyulas C., O'Connor M. and Tu S. W., DataMaster – a Plug-in for Importing Schemas and Data from Relational Databases into Protégé. In in Proc. 10th International Protg Conference,, Budapest, Hungary, 2007.
[15] Rao R., Lecture 20: Topo-Sort and Dijkstra's Greedy Idea, https://courses.cs.washington.edu/courses/cse326/03wi/lectures/RaoLect20.pdf. [Accessed 30 May 2018].
[16] Riehle D. and Gross T., Role Model Based Framework Design and Integration, in Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia, Canada, 1998.
[17] Stevenson G. and Dobson S., Sapphire: Generating Java Runtime Artefacts from OWL Ontologies, In: Salinesi C., Pastor O. (eds) Advanced Information Systems Engineering Workshops. CAiSE 2011. Lecture Notes in Business Information Processing, vol 83. Springer, Berlin, Heidelberg, 2011.
[18] Volker J., Hitzler P. and Cimiano P., Acquisition of OWL DL Axioms from Lexical Resources. Lecture Notes in Computer Science, 4519, 670-685.
[19] Vyšniauskas E., Nemuraitė L. and Paradauskas B., Preserving Semantics of OWL 2 Ontologies in Relational Databases Using Hybrid Approach, Information Technology and Control, Vol. 41, No. 2, 2012, 103-115.
[20] Woroniecki P., OWL2 to Groovy Converter, https://bitbucket.org/pworoniecki/owl2-converter. [Accessed 3 June 2018].

[21] Yabloko L., OntoBase, https://protegewiki.stanford.edu/wiki/OntoBase. [Accessed 3 June 2018].
[22] Zedlitz J., Jörke J. and Luttenberger N., From UML to OWL 2, In: Lukose D., Ahmad A.R., Suliman A. (eds) Knowledge Technology. Communications in Computer and Information Science, Berlin, Heidelberg, Springer, 2012, 154-163.
[23] "Framework. BusinessDictionary.com", WebFinance, Inc., http://www.businessdictionary.com/definition/framework.html. [Accessed 3 June 2018].
[24] "Framework". Techopedia, https://www.techopedia.com/definition/14384/software-framework. [Accessed 3 June 2018].
[25] OWL 2 Web Ontology Language. Document Overview (Second Edition), W3C, 11 December 2012, https://www.w3.org/TR/owl2-overview/. [Accessed 3 June 2018].
[26] Spring Context, MvnRepository, https://mvnrepository.com/artifact/org.springframework/spring-context [Accessed 9 May 2018].
[27] The Sigma knowledge engineering environment, http://ontologyportal.github.io/sigmakee/. [Accessed 3 June 2018].
[28] OSGi Alliance, The Dynamic Module System for Java, https://www.osgi.org/ [Accessed 30 September 2018].
[29] Spring Framework, https://spring.io/projects/spring-framework [Accessed 30 September 2018].
[30] Apache Felix, https://felix.apache.org/ [Acessed 30 September 2018].

Received 4.07.2018, Accepted 19.10.2018