Java2OWL: A System for Synchronising Java and OWL

Hans Jürgen Ohlbach
Institute for Informatics, University of Munich, Munich, Germany

Keywords: Java, OWL, Translation, Semantic Programming.

Abstract: Java2OWL is a Java software library for synchronising Java class hierarchies with OWL concept hierarchies. With a few extra annotations in Java class files, the Java2OWL library can automatically map Java class hierarchies to OWL ontologies. The instances of these Java classes are automatically mapped to OWL individuals and vice versa. OWL reasoners can be used as query processors to retrieve instances of OWL concepts, and these OWL individuals are mapped to corresponding instances of the Java classes. Changes of the Java object's attributes are automatically mapped to changes of the corresponding attributes of the OWL individuals, thus keeping Java and OWL synchronised. With minimal programming overhead the library allows one to combine the power of programming (in Java) with the expressivity and the reasoning power of OWL. This paper introduces the main ideas and techniques. The detailed documentation and the open source library itself can be downloaded from http://www.pms.ifi.lmu.de/Java2OWL.

1 MOTIVATION

Object oriented programming languages like Java combine the procedural aspects of programming with some declarative-logical features contained in the Java class hierarchy and the instance-class relationship. Whereas the procedural aspects are as strong as one can expect from modern programming languages, the logical aspects are rather restricted. In particular

- the ontology, i.e. the logical structure, implicitly contained in the class hierarchy of Java programs is only accessible from within the program and can not be exported to other systems;
- the instance-class relationship is fixed at the creation time of an object. Objects can not live independently of classes and can not change their membership relation to classes;
- the class hierarchy in Java is a tree; no multiple inheritance is possible;
- there is no logical reasoning available. This means in particular that fine grained selection of objects is difficult.

Nevertheless, these are desirable features, which is indicated by the rising popularity of modelling languages like UML, and in particular by the more and more widespread use of the logic-based Ontology Working Language (OWL[1]). OWL is the W3C-standard for Description Logics (Baader et al., 2003; Lutz, 2003). Its main features are

- one can define ontologies independently of any particular application or programming language;
- the OWL-language is very expressive and more features are being added as one learns how to extend the logical calculus for them;
- since it is logic based, various forms of reasoning can be performed;
- the class hierarchy can be a DAG, thus, multiple inheritance is allowed;
- individuals can live independently of classes. Their membership relation with classes is determined by logical reasoning, and it can change when new information about an individual becomes available;
- there are expressive languages which can be used to pose complex queries to an ontology. They are evaluated by logic reasoners.

The disadvantage of OWL is the lack of procedural features; OWL is no programming language. For an OWL ontology together with an A-Box (a set of individuals) one can draw all possible inferences, i.e. compute the class hierarchy and the membership relation of the individuals with the classes. After this, however, the ontology becomes a static object which can do nothing by itself.

[1]OWL: http://www.w3.org/TR/owl2-overview/

Ohlbach H..
Java2OWL: A System for Synchronising Java and OWL.
DOI: 10.5220/0004106400150024
In *Proceedings of the International Conference on Knowledge Engineering and Ontology Development* (KEOD-2012), pages 15-24
ISBN: 978-989-8565-30-3

An OWL ontology just living in a file system is not of much use. Therefore application interfaces (API) have been developed which allow programs to load an ontology, access its data, manipulate the ontology, and save it back to files, all within ordinary programs. The most recent one is the Java OWL API[2] for OWL-version 2 (Horridge and Bechhofer, 2011), developed as part of the CO-ODE project[3] and the TONES project[4] in the Manchester University[5]. It allows one to load, access and manipulate the structure of OWL ontologies and to integrate them with OWL reasoners.

Java programs using the OWL API for loading ontologies have two class hierarchies in parallel, the Java class hierarchy and the OWL class hierarchy. For applications where these hierarchies are sufficiently different, this is no problem. If, however, the class hierarchies overlap, one gets a considerable synchronisation problem. Overlapping class hierarchies are of interest if one wants to add procedural features to the classes. Since OWL has no procedural features, the only choice is to define corresponding Java classes with the corresponding methods.

As an illustrating example, consider the partial modelling of university structures. There are people, students, tutors, lecturers, professors, secretaries, technical staff etc. There are programmes of study, Bachelor, Master and PhD programmes. There are lectures, seminars, excursions etc. There are scientific areas, natural sciences, humanities, social sciences etc. All this can be modelled in OWL, but there is no chance to add procedural features to these OWL classes. This can only be done in corresponding Java classes. Combining Java classes and their methods with corresponding OWL classes as the basis of OWL-reasoning, however, may turn out to be a very useful thing. The Java methods do the computation, and the OWL classes can in particular be used to select sets of individuals by determining the instances of OWL concept expressions.

The Java2OWL library presented in this paper targets the synchronised coexistence of Java classes and OWL classes. It makes it possible to exploit the strengths of both systems without too much programming overhead.

2 A TYPICAL WORKFLOW

A typical workflow in the development and application of Java programs using the Java2OWL library consists of two major phases:

1. the preparation phase, which consists of the programming and the compilation phase,
2. the Java application phase which starts with the steps a Java application has to perform in order to work with the Java2OWL library.

2.1 The Preparation Phase

This phase consists of the following steps:

1. **Background Ontology.** Most applications which want to use ontologies do not start from scratch, but us already existing ontologies, or develop specialised ontologies for this application. Therefore the Java2OWL library assumes the existence of a "background ontology" with predefined classes and properties. This background ontology can be loaded into the Java2OWL-compiler which then extends it with OWL classes generated from Java classes.
2. **Annotating Java Classes.** Java classes which are to be mapped to OWL classes must be annotated in a special way. The class-level annotations control the generation of OWL classes and the method-level annotations control the generation of the OWL-properties. Typically, the Java class has getter methods, setter methods, and for collection-valued attributes, adder and remover methods. The annotations of the getter methods are sufficient to tell the system how to access and manipulate the Java-attributes and to synchronise them with the OWL-properties.
3. **Compiling the Extension Ontology.** The annotated Java classes are mapped to newly generated OWL classes which are added to the background ontology. The getter methods are mapped to OWL-properties, either data properties, mapping objects to data types like integer, or object properties, mapping objects to other objects. The so extended background ontology is then saved as "extension ontology".
4. **Extending the Extension Ontology.** In some applications it may be necessary to manually extend the extension ontology, for example by adding special individuals. Therefore one can manipulate the extension ontology outside the Java2OWL-system in an almost arbitrary way before it is used in the application program.
5. **Injecting Synchronisation Code into the Java Classes.** Since Java classes and OWL classes coexist in the Java2OWL system, their instances also

[2]OWL API: http://owlapi.sourceforge.net/
[3]CO-ODE project: http://www.co-ode.org/
[4]TONES project: http://www.tonesproject.org/
[5]Manchester University: http://owl.cs.manchester.ac.uk/

coexist. This means, for each Java object which is an instance of an annotated Java class, there is a corresponding OWL individual. This OWL individual should mirror the attributes of the Java object. If the Java object changes its attributes, the changes should automatically trigger changes of the attributes of the OWL individual. The code for forwarding the changes of the Java-attributes to the OWL side must be contained in the setter, adder and remover methods of the Java class. In order to relieve the Java programmer from writing this "synchronisation code" the Java2OWL library contains a "Synchroniser" class which automatically injects the "synchronisation code" into the compiled Java classes. This can be done either in a separate step after the ordinary Java compilation, or it can be done "on the fly" in the class loader.

2.2 The Java Application Phase

As soon as all the preparations have been successfully completed, i.e. the Java application with the annotated classes has been build and the extension ontology is ready to be used, the application can start. These are the typical steps, the application must do before its main job can start:

1. **Setting Up the System.** This is simply done by creating a J2OManager. The J2OManager itself creates all other components and installs a default OWL-reasoner. One can create several J2OManagers, but they are completely independent of each other and do not interact with each other.
2. **Loading the Extension Ontology**
3. **Linking the Extension Ontology.** In this step the internal data structures are created which keep track of the correspondences between Java classes and OWL classes.
4. **Creating Java-objects and OWL Individuals.** The system is now ready to create instances of classes. It is possible to instantiate Java classes and let the Java2OWL-system map them to OWL individuals. The other direction is also possible. OWL individuals contained in the A-Box of the extension ontology can be mapped to instances of the corresponding Java classes.
5. **Keeping Java-objects and OWL Individuals Synchronised.** Changes to the attributes of instances should only be done by calling setter, adder, and remover methods of the Java-objects. The injected synchroniser-code automatically forwards the changes to the OWL-side.
6. **Reclassification of Instances.** OWL is logic based. Therefore OWL reasoners can derive information which was only implicitly contained in the OWL ontology generated from the Java side. This derived information can then be mapped back to the Java side. There are two typical situations where this can occur:
   - An Object's Class May Be Changed. Consider a Java class `Student` with attribute `semester`, and a subclass `Freshman` whose semester is frozen to the value 1. An instance of the Java class `Student` whose semester is set to 1 should now become an instance of `Freshman`. This is not directly possible in Java. In the Java2OWL-system, however, the Java-objects are contained in wrapper-objects. Therefore it is possible to exchange a Java-object inside a wrapper object with another Java object. In the `Student`/`Freshman`-example this would be performed by creating an instance of `Freshman` and transferring all non-static fields of the `Student`-instance to the new `Freshman`-instance. The new `Freshman`-instance then replaces the `Student`-instance inside the wrapper object.
   - An Object's Attributes May Be Changed. Consider a Java class with a *transitive* `has-Part` attribute. If this is mapped to OWL then an OWL reasoner can derive the transitive closure of the `has-Part` relation. Mapped back to Java the `has-Part` attribute is automatically filled with the transitive closure of the `has-Part` relation.

   The application program can ask the Java2OWL-system at any time to do this reclassification of instances whose attributes have been changed.

3 JAVA CLASS HIERARCHY VERSUS OWL CLASS HIERARCHY

The intrinsic Java ontology which is implicitly contained in the Java class hierarchy is simpler than the OWL ontology: mono-inheritance causes the Java class hierarchy to be become a tree, and the constructor mechanism for generating new instances of classes causes Java objects to be an instance of exactly one Java class. In contrast to this, OWL has multi-inheritance, and OWL individuals may be instances of several classes. Mapping a Java ontology to an OWL ontology is therefore not problematic. This part of the Java2OWL library is purely technical.

The Java2OWL library, however has three further features which are more problematic:

- it can map OWL individuals back to Java objects,
- it can attach attributes to Java objects, which are not foreseen in the Java class definition, and which must therefore be immediately forwarded to the OWL side,
- it can reclassify Java objects when further information becomes available.

If OWL individuals are to be mapped back to Java objects, we have to deal with the problem that an OWL individual may be an instance of several OWL classes. Therefore one can not just create an instance of *one* particular Java class as corresponding Java object. If extra attributes can be attached at Java objects by forwarding them to the OWL side, this may cause a new situation. For a Java object o as an instance of the Java class C which corresponds to a unique OWL object o' as instance of the OWL class C' the OWL instance o' may suddenly become an instance of other OWL classes, and therefore there is no longer this unique correspondence between o and o'. The problem turns up when the Java object is reclassified to incorporate the new information.

The solution to these problems in the Java2OWL library so far is *experimental* because no real application where these problems turned up have been tried. The solution is to encapsulate the correspondences between Java objects and OWL individuals in *individual wrappers*. An individual wrapper encapsulates an OWL individual together with *several* Java objects.

The typical example which illustrates the multi-inheritance phenomenon is the OWL class `Amphibious-Vehicle` as subclass of `Ship` and `Surface-Vehicle`. Since in Java there can not be a class `Amphibious-Vehicle` extending `Ship` and `Surface-Vehicle`, there is a problem. In Java one could in principle define either `Ship` as a class and `Surface-Vehicle` as an interface, and then define a class `Amphibious-Vehicle` extending `Ship` and implementing `Surface-Vehicle`, or the other way round. In this case, however, there can not be instances of `Surface-Vehicle`, which might not be a good idea. Therefore when mapping an Amphibious-Vehicle instance from OWL to Java, Java2OWL creates an individual wrapper containing a Java Ship instance and a Java Surface-Vehicle instance. It takes care that the common attributes of both Java objects are pointer-equal.

Even when multi-inheritance is avoided at the OWL side, there can be situations where an OWL individual must be mapped to several Java objects. As an example consider a Java class `Student` and a Java class `Teacher` which have nothing to do with each other, except that the `Teacher` class has an attribute `teaches` (some students). Both `Student` and `Teacher` are mapped to the OWL ontology. Now suppose, `James` is an instance of `Student`, and we add the extra information that `James` is not only a student, but teaches another student `Tom`. This information is not attached to the Java object `James`, but forwarded to the corresponding OWL individual `James'`. At this moment, `James'` becomes an instance of the OWL class `Teacher'`. If this information is mapped back to the Java side, besides the Java object `James` as instance of the Java class `Student`, we need also an instance, say `James-Teacher`, of the Java class `Teacher`. Both Java objects can be wrapped in an individual wrapper, which at first glance, solves the problem.

There is, however, a further problem. Suppose `James` does not teach another student `Tom`, but James teaches himself, `James`. The Java instance `James-Teacher` has an attribute `teaches` of type `Student`. This attribute has to be filled with James, but which James, the `Student` instance `James` or the `Teacher` instance `James-Teacher`? In this case it is clear by the type of the `teaches` attribute, that the Student instance `James` is the only Java object which can be put into the `teaches` attribute. In other examples, however, there may be several choices. The current implementation is such that it takes the first one whose Java class fits the class of the attribute. Only practical applications can show what is really needed in these situations.

4 COMPONENTS OF THE LIBRARY

This section gives a brief overview of the main components of the system.

4.1 Annotated Java Classes

The first aspect a programmer who wants to use Java2OWL is confronted with, is the structure of the Java classes to be mapped to OWL classes. The following restrictions to the structure of the Java classes are important:

- There must be a constructor method with an empty argument list. It is used when OWL individuals are mapped to Java objects.
- All attributes which are to be mapped to OWL-attributes must be accessible by getter, setter, adder and remover methods.

- The mapping of Java classes to OWL classes is controlled by special annotations of class and method definitions.

4.1.1 The Class-level Annotation

A simple example illustrates the annotation of classes:

```
@J2OWLClass(name = "Person",
    OWLSuperClasses = "LivingThing")
public class TestPerson {...}
```

The `name`-attribute `Person` causes the mapping of the Java class `TestPerson` to a newly created OWL class `Person`. The `OWLSuperClasses`-attribute `LivingThing` causes the generated OWL class `Person` to become a subclass of the OWL class `LivingThing`, which must be part of the background ontology.

Besides `name` and `OWLSuperClasses`, three further attributes can be used in the annotation: `EquivalentClass`, `synchronise` and `naming`. `EquivalentClass` with an OWL class expression in Manchester Syntax[6] as value causes the generated OWL class to be equivalent to the given class expression. `synchronise` triggers the insertion of synchroniser code into the setter, adder and remover methods. `naming` controls the assignment of names to the generated OWL individuals.

4.1.2 The Method Level Annotations

The Java2OWL annotations control the mapping of Java-attributes to OWL-properties. OWL distinguishes two kinds of properties:

OWLDataProperty: these describe basic datatype-valued attributes of individuals. OWL has a number of basic data types built-in, for example, integer, float, double, etc., but also strings. Examples could be a string-valued `name`-attribute, or an integer-valued `age`-attribute.

OWLObjectProperty: these describe relations between OWL individuals. Examples are the `hasPart` relation between physical objects, or the `hasParent` relation between persons.

Both property types can be *functional* or *non-functional* i.e. *relational*. Functional properties have a single value, whereas relational properties can have any number of values, including no values at all.

On the Java side there are the instance-variables which, depending on their type, can take primitive data types as values, references to other Java-objects, but also references to container objects like sets, arrays etc. It is good Java practice to keep the instance variables private and to access them with getter and setter methods. Therefore the mapping from Java-attributes to OWL-properties is specified by annotating getter methods instead of the instance variables. Java2OWL assumes the following conventions:

getter Methods: They have no arguments and usually return the value of a private variable. Since Java is strongly typed, the return type of the getter methods can be used to determine the kind of OWL-property to be generated. The following cases are distinguished:

- The return type of the getter method is a primitive data type or the type `String`. In this case a *functional* OWLDataProperty is generated.
- The return type of the getter method is an array or a container class for primitive data types or strings. In this case a *relational* OWLDataProperty is generated.
- The return type of the getter method is a Java2OWL annotated Java class. In this case a *functional* OWLObjectProperty is generated.
- The return type of the getter method is an array or a container class for a Java2OWL annotated Java class. In this case a *relational* OWLObjectProperty is generated.

setter Methods: Setter methods usually overwrite the value of a private variable. They are called with at least one argument and need not return a value. The type of the *first argument* must be such that it accepts the result of the corresponding getter method.

adder Methods: Collection-valued attributes, arrays or collections, need not change the whole set at once, but add single elements one by one. Adder methods therefore take an element of the collection as first argument and add it to the set.

remover Methods: remover methods for collection-valued attributes take an element of the set as first argument and remove it from the set.

clearer Methods: clearer methods for collection-valued attributes empty the whole collection at once.

For each attribute to be mapped there must be exactly one annotated getter method which returns the entire set of values, either a single object if there is just one value, i.e. the attribute is functional, or an array or a collection of objects if there are multiple values.

For each attribute there is a particular group of getter, setter, and optionally adder and remover methods which belong together. Therefore it is convenient to

[6] Manchester Syntax: http://www.w3.org/2007/OWL/wiki/ManchesterSyntax

annotate the getter methods only and specify in the annotation the other methods belonging to the group.

A typical example for an annotated getter method is

```
@J2OWLProperty(name = "hasName",
    setter="setName")
public String getName() {
    return name;}
public void setName(String name) {
    this.name = name;}
```

The specification of `@J2OWLProperty` is:

```
public @interface J2OWLProperty {
    String name() default "";
    boolean local() default false;
    String setter() default "";
    String adder() default "";
    String remover() default "";
    String clearer() default "";

    boolean transitive() default false;
    boolean symmetric() default false;
    boolean asymmetric() default false;
    boolean reflexive() default false;
    boolean irreflexive() default false;
    boolean total() default false;
    int atleast() default -1;
    int atmost() default -1;
    boolean addRangeAxiom() default true;}
```

The meaning of these attributes is:

name: is the name of the corresponding OWL-property.

local: If this flag is set to true then the generated name is prefixed with the class name.

setter, adder, remover: These are the comma separated names of the corresponding methods which are responsible for the same attribute.

clearer: This must be the name of a parameterless method which empties collection valued attributes.

transitive, symmetric, asymmetric, reflexive, irreflexive: These are properties of OWLObjectProperties.

total: If this flag is set to true it enforces that the functional properties is total, but only for the OWL class generated from this particular Java class.

atleast, atmost: They specify the minium and maximum number of role fillers for relational properties.

addRangeAxiom: If `addRangeAxiom` is true then corresponding range type axiom is specified for the OWL-property generated from the getter method.

4.2 The Java2OWL Compiler

The Java2OWL compiler, implemented in the class `J2OCompiler`, generates OWL classes from annotated Java class files. For the preparation phase it has a main method such that it can be called to generate the extension ontology from a given list of compiled Java classes.

The program reads the background ontology, creates an extension ontology, reads the class files, analyses its annotations, fills the extension ontology with the translated Java classes and saves the extension ontology. Error messages are printed to System.out.

Notice that not all Java class files to be translated need to be given as input. For a given class C to be compiled to OWL the J2OCompiler automatically translates the following classes:

- all annotated superclasses of C,
- all annotated *static* inner classes of C,
- all annotated range type classes of the annotated getter methods in C.

In the application phase (see Sect. 2.2) its methods can be used to link the Java classes with the OWL classes i.e. to build the internal data structures necessary for managing the correspondences between Java and OWL.

4.2.1 Compile Time Errors

The `J2OCompiler` analyses annotated Java class and combines the extracted structures with the background ontology. In this step it tries to detect as many programming errors as possible.

Here are examples for errors it can detect:

- Inconsistencies in the annotations of getter methods. For example, a relation can not be declared reflexive and irreflexive at the same time.
- Java data types which can not be mapped to OWL data types:
  The Java data types which can be mapped to OWL data types are the primitive Java data types boolean, byte, short, int, long, float, double and the String type. All other built-in Java data types, for example BufferedString, are not mapped to OWL data types. A specification like

```
@J2OWLProperty(name = "hasName")
public BufferedString getName() {
   return name;}
```

therefore causes an error.

- Classes without necessary Java2OWL annotations:
  Consider a class `Person` with a getter method

```
@J2OWLProperty(name = "hasAddress")
public Address getAddress() {
   return address;}
```

  This causes a functional Object Property `hasAddress` to be created, which maps individuals of type `Person` to individuals of type `Address`. If there is no class `Address` in the background ontology and the Java class `Address` is not annotated, the relation `hasAddress` makes no sense in the extension ontology.

The `J2OCompiler` can of course not detect all errors which may course trouble at run time. Therefore run time error handling is also necessary (see Sect. 4.4.2).

4.3 Synchronisation between Java and OWL

4.3.1 Java to OWL Synchronisation

Java to OWL synchronisation means that changes to attributes of Java objects are forwarded to the corresponding OWL individuals. The Java2OWL library provides two different possibilities to do this:

Life Synchronisation. This means that all changes to the attributes of Java objects are immediately forwarded to the corresponding OWL individual. This is only possible if the annotated Java classes got 'synchroniser code' injected. Life Synchronisation can be turned on and off at any time. This can be done at class level, i.e. for all instances of a given class. The activation/deactivation of class level life synchronisation can be overwritten by activating/deactivating it for single objects.

Block Synchronisation. This means that changes to the attributes of Java objects are not forwarded to OWL for a while, and at some time the application decides to do this 'en bloc' for all attributes of the object.

4.3.2 The Java2OWL Synchroniser

The `J2OSynchronizer`-class is used to insert extra 'synchroniser code' into the setter, adder, remover and clearer methods of the compiled Java class files. It can be used in two ways:

- the J2OCompiler calls it when a corresponding flag is set,
- the `J2OSynchronizerAgent` calls it in its premain-method. This causes a further transformer to be installed in the class loader. This transformer calls the J2OSynchronizer to inject the synchronizer code when a compiled class is loaded.

The bytecode manipulation for injecting the synchroniser code has been programmed with the Byte Code Engineering Library[7] (Apache Commons BCEL™) (Dahm, 1999). An alternative would have been to use AspectJ[8] (Colyer et al., 2004). With the BCEL-library, however, certain optimisations of the byte-code were possible, which were not supported by AspectJ.

4.4 The J2OOntology Manager

The `J2OOntologyManager` is a kind of interface between the OWL API and the application. The OWL API consists of three main components:

- the `OWLOntologyManager` which stores all the data belonging to an ontology,
- a `OWLDataFactory` which creates the structures necessary to interact with the ontology,
- a reasoner. There can be several reasoners, but only one can be active within a J2OManager at any time.

The current version of the Java2OWL library supports three reasoners, HermiT (Birte Glimm et al, 2010), Pellet (Sirin et al., 2007) and FaCT++ (Tsarkov and Horrocks, 2006).

Since the interaction with the OWL API is sometimes quite cumbersome, the `J2OOntologyManager` hides the pecularities of the OWL API. Its main tasks are:

- loading and saving ontologies;
- setting up reasoners;
- getting information about the components of the ontology;
- making changes to the extension ontology.
- querying the ontology with expressions in Manchester Syntax. OWL individuals as answers to the queries are automatically mapped to Java objects.

4.4.1 The Interaction with the OWL API

The OWL API stores information about an ontology in two ways:

[7] http://commons.apache.org/bcel/
[8] AspectJ: http://www.eclipse.org/aspectj/

- The `OWLOntologyManager` in the OWL API itself has an internal representation of the ontology. Various interface method can access this information.
- The reasoner must of course also have an internal representation of the ontology. Since reasoners in different programming languages are available (FaCT++, for example, is written in C++), the information about the ontology must be forwarded from the `OWLOntologyManager` to the reasoner.

Changes to an ontology are therefore done in three steps:

1. Logical axioms are created and stored in a list. This does not yet change the ontology itself at all.
2. The `OWLOntologyManager` is asked to apply the changes, i.e. to integrate the axioms into the ontology.
3. The new information is forwarded to the reasoner. This may either be done as soon as the `OWLOntologyManager` integrates a new axiom, or the changes may be buffered and made effective at a later step. In this case the ontology may have become inconsistent before the reasoner has a chance to check it. The `J2OOntologyManager` makes sure that whenever the reasoner is asked to do something, the buffer is first flushed to the reasoner.

4.4.2 Sources of Inconsistencies in the Extension Ontology

Inconsistency is not a relevant notion in an ordinary programming context. Therefore it is a reasonable question to ask where inconsistencies in Java2OWL-managed ontologies can come from. Certain sources of inconsistencies, which can be detected by the compiler, have been discussed in Sect. 4.2.1. In this section we discuss inconsistencies which may show up at run time.

Here are some examples for inconsistencies:

- Inconsistencies between the background ontology and the annotations of getter methods:
  In a background ontology, one might for example specify a *functional* property `hasName`, and in a Java class a getter method. An annotated getter method may be

```
@J2OWLProperty(name = "hasName")
public String[] getNames() {
    return names;}
```

  where the return type is an array. This is not inconsistent with the functionality of `hasName` as long as the result of the `getNames()`-method has just a single element. As soon as the `getNames()`-method returns a longer array, it becomes inconsistent with the required functionality of `hasName`.
  This is a case where a compiler could issue a warning, but the situation may be intended and there may be no problem at run time.
- Constraint Violations:
  The annotation

```
@J2OWLProperty(name = "hasFriend",
    atmost = 3)
public Set<Person> getFriends() {
    return friends;}
```

  specifies that one can have at most three friends. If `getFriends()` returns a set with more than three friends, this contradicts the constraint `atmost = 3`.

4.4.3 Checking the Consistency of the Extension Ontology

Checking for inconsistencies by the reasoner is an expensive operation. Therefore it is a strategic decision when to check the consistency of the extension ontology. During development and debugging it is useful to find inconsistencies as early as possible. To this end, the `J2OManager` can be put into *debug mode*. It causes each change to the ontology to be immediately forwarded to the `OWLOntologyManager` and to the reasoner and to ask the reasoner to immediately check the consistency.

If the system is not in debug mode then changes to the ontology are buffered as long as possible. The changes to the ontology are forwarded to the `OWLOntologyManager` and to the reasoner

- either when the reasoner is asked to compute some information,
- or when new information overwrites old information, and it is necessary to retrieve the old information in order to generate the corresponding remove-axioms.

4.5 The J2OClass Manager

The class `J2OClassManager` manages the correspondences between the Java classes and the OWL classes. To this end it has two auxiliary data structures:

`ClassWrapper` which stores the pair [Java Class, OWL Class] together with information about the mapped attributes.

`PropertyWrapper` which stores the correspondences between the Java getter, setter, adder, remover and clearer methods on the one side, and

the corresponding OWL-property on the other side.

If the `J2OClassManager` is asked to get for a Java class the correspondence to the OWL class it does not return an OWL class, but a `ClassWrapper`. From the `ClassWrapper` one can get the OWL class. The same holds for the attributes.

4.6 The J2OIndividual Manager

The `J2OIndividualManager` manages the correspondences between the Java objects and the OWL individuals. The main purposes of the `J2OIndividualManager` are therefore

- mapping Java objects to OWL individuals and vice versa and
- synchronising the Java object's attributes with the corresponding OWL properties.

The correspondences between Java objects and OWL individuals are encapsulated in the class `IndividualWrapper`. Individual wrappers are actually the object level front end to the OWL ontology. Each individual wrapper contains one OWL individual and, due to multi-inheritance of OWL one or more corresponding Java objects. The most important operations, an individual wrapper can perform are:

- activating/deactivating life synchronisation
- block synchronisation with OWL
- reclassification of the Java objects,
- attaching extra attributes to the objects which are not foreseen in the Java class definitions. These attributes are actually attached at the OWL individual in the OWL ontolgoy.

5 PERFORMANCE OF THE SYSTEM

A thorough performance analysis of the Java2OWL system is not really possible. The input are Java programs and OWL ontologies, and there is no clear output which can be measured. Moreover, a significant part of the system is the OWL reasoner, and this is not under control of the Java2OWL system. The complexity of the OWL reasoning tasks depends on the OWL constructs used in the application. The complexity classes range from polynomial to undecidable. Typical examples are in the PSPACE range, which means that heuristics have an important influence on the behaviour of the reasoners.

The few experiments whose results are listed below might give a rough impression on the performance of the various parts of the system. They have been performed with a Java class `Student`, with attributes `name` and `semester` and a subclass `Freshman` which are students in the first semester. The classes have been translated to OWL and then a number of measurements have been performed.

In all experiments a sequence of lists of `Student` instances have been created, and for each list a certain operation has been performed. The time it took to perform the operation on the list has been measured, and divided by the length of the list. The resulting times in milliseconds indicate the time it took to perform a single operation. The lengths of the lists are 2000, ..., 10000. The measurements were done with a Dell notebook with a 2.2 GHz dual core processor.

Java to OWL Axioms. Each `Student` has been translated to OWL axioms, but the axioms are not yet integrated into the ontology. The times in milliseconds per operation are:

size	2000	4000	6000	8000	10000
ms	0.19	0.04	0.01	0.032	0.0055

It is not clear why the times are so different. One effect could be the Just-in-Time compilation of Java which after a while decides to compile the Java methods into native machine code.

Java to OWL Indivdiduals. In this experiment the translated Student instances are integrated as OWL individuals into the OWL ontology. The times in milliseconds per operation are:

size	2000	4000	6000	8000	10000
ms	0.39	0.16	0.016	0.011	0.03

Java to OWL Individuals with Consistency Test. In this experiment the translated Student instances are integrated as OWL individuals into the OWL ontology, and each time a consistency test is performed by the OWL reasoner. All three reasoners are tried. The times in milliseconds per operation are:

size	2000	4000	6000	8000	10000
FaCT++	0.14	0.046	0.013	0.07	0.09
Pellet	0.13	0.076	0.064	0.077	0.1
HermiT	0.17	0.120	0.037	0.031	0.09

Java-OWL Synchronisation. This time the `Student` instances which have been mapped to OWL get their semester changed to 1, and this is forwarded to the ontology. It does not involve OWL reasoning. The times in milliseconds per operation are:

size	2000	4000	6000	8000	10000
ms	0.07	0.012	0.0077	0.11	0.0075

Querying OWL. In this experiment we take the `Student` instances whose semester had been changed to 1, and map them to OWL individuals. Afterwards the OWL reasoner is asked to retrieve all instances of the OWL `Freshman` class. All three reasoners are tried. The operation is much more expensive than the previously investigated operations. Therefore only much smaller lists of `Student` instances are tried. The times in milliseconds per operation are:

size	200	400	600	800	1000
FaCT++	0.455	0.187	0.167	0.22	0.18
Pellet	7.57	8.02	12.3	15.3	22.4
HermiT	5.14	10.0	17.12	27.8	37.6

For this class of examples FaCT++ is about 20 times faster than the other reasoners. Since OWL reasoning is a heuristically controlled search there may well be other classes of examples where the other reasoners are faster.

OWL to Java. In this experiment the `Student` instances whose semester had been changed to 1, and which had been mapped to OWL individuals are deleted, and afterwards reconstructed as `Freshman` instances form the OWL individuals. The times in milliseconds below indicate how long it took to turn an OWL individual into a Java object.

size	2000	4000	6000	8000	10000
ms	0.119	0.13	0.222	0.198	0.238

Reclassification. In this experiment the `Student` instances whose semester had been changed to 1 are reclassified to `Freshman` instances. All three reasoners are tested. The times in milliseconds per operation are:

size	200	400	600	800	1000
FaCT++	1.11	0.43	0.29	0.29	0.3
HermiT	5.48	9.49	16.22	25.73	35.08
Pellet	7.58	13.13	16.36	20.05	29.24

Again, FaCT++ is much faster than the other reasoners.

The times vary quite a lot with the different lengths of the lists. An important observation is, however, that the times per operation do not depend much on the size of the data. Sometimes the operations become even faster with increased amounts of data.

6 SUMMARY

This paper gives a short introduction into the Java2OWL library for synchronising Java with OWL. A more substantial description is contained in the technical report (Ohlbach, 2012). It can be downloaded from http://www.pms.ifi.lmu.de/Java2OWL. From this website one can also download the library itself, and even the whole NetBeans project.

The system has not yet been tested in a real application. It is planned to use it for a university wide information system. It is quite clear that real applications will require further changes to the library. Suggestions for improvements may be E-mailed to ohlbach@lmu.de.

REFERENCES

Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Birte Glimm et al (2010). Optimising ontology classification. In Patel-Schneider, P. F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J. Z., Horrocks, I., and Glimm, B., editors, *Proc. of the 9th Int. Semantic Web Conf. (ISWC 2010)*, volume 6496 of *LNCS*, pages 225–240, Shanghai, China. Springer.

Colyer, A., Clement, A., Harley, G., and Webster, M. (2004). *Eclipse AspectJ : Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley.

Dahm, M. (1999). Byte code engineering. In *Proceedings JIT'99*, pages 267–277. Springer-Verlag, Springer-Verlag.

Horridge, M. and Bechhofer, S. (2011). The OWL API: A Java API for OWL ontologies. *Semantic Web*, 2(0):11–21.

Lutz, C. (2003). Description Logics with Concrete Domains—a survey. In *Advances in Modal Logics Volume 4*. King's College Publications.

Ohlbach, H. J. (2012). Java2OWL – a system for synchronising Java and OWL. Technical Report PMS-FB-2012-02, Institute for Informatics, University of Munich.

Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semantics*, 5(2):51–53.

Tsarkov, D. and Horrocks, I. (2006). FaCT++ Description Logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer.