Towards an Architecture to Bind the Java and OWL Languages

Jose M. Alcaraz Calero[1], Jorge Bernal Bernabé [1 2], Juan M. Marín Pérez[1 2],

Diego Sevilla Ruiz[3], Felix J. García Clemente[3], Gregorio Martínez Pérez[2]

[1]Cloud and Security Lab

Hewlett Packard Laboratories Bristol

Filton Rd, Stroke Gifford

BS34 8QZ Bristol, UK

[2]Department of Information and Communications Engineering

University of Murcia

Facultad de Informática, Campus de Espinardo,

30071 Murcia, Spain

[3]Department of Computing Engineering and Technology

University of Murcia

Facultad de Informática, Campus de Espinardo,

30071 Murcia, Spain

(jose.alcaraz-calero@hp.com, jorgebernal@um.es, juanmanuel@um.es,

dsevilla@um.es, fgarcia@um.es, gregorio@um.es)

01/05/2012

Abstract

There exists a well-known trade off between imperative and declarative languages. Imperative languages such as Java are suitable to describe processes and how these processes should be done. Declarative languages such as OWL are used to describe systems and what is available in them. Each one has advantages and disadvantages and each one is more convenient in some circumstances than in others. For example, graphical interfaces might be easily implemented using the Java language whereas the description of the state of a system might be easily described in the OWL language. Hence, this proposal describes a binding process between Java and OWL to provide an architecture enabling the usage of both languages during software development and at run-time. The architecture allows getting the advantages of both programming paradigms, declarative and imperative, together. As a result, the reasoning capabilities provided by OWL could be used in Java applications whereas the flow control of applications and the I/O functions available in Java could be used seamlessly for OWL ontologies. The aim is to speed up software developments using the facilities provided by both types of languages. As a proof of concept, the proposed architecture has been implemented as an open source framework, and some technical details of this implementation together with statistical results of performance and scalability are discussed in this paper.

Categories and Subject Descriptors

I.2.4 [Knowledge Representation Formalisms and Methods]: Representation languages; F.3.2 [Semantics of Programming Languages]: Operational semantics; D.3.1 [Formal Definitions and Theory]: Semantics

Keywords

Semantic Web, OWL Ontologies, SWRL Rules, Java Object-Oriented Programming Language, Knowledge Representation

1 Introduction

In our previous research, [9, 33, 7] providing authorization systems based on the Semantic Web formal methods for distributed systems, grid and cloud scenarios, respectively, we realized that the underlying formalism provided as foundation for the Semantic Web fits very well the design constraints of these systems. We also devised a clear necessity and convenience of mixing the Java and OWL languages together in order to enable a clear binding between the formal methods available under OWL and the imperative high level programming functionalities available in Java.

The Java language, on the one hand, allows an imperative design and specification of programming tasks, as well as a data abstraction based on classification, interface offering and inheritance. In this sense, it facilitates dealing with side-effect tasks such as obtaining inputs, processing them, and producing output. Recent versions of the Java language standard provide better procedural and data-based abstractions, such as, among others, genericity [8], parametric classes [3], and loops for collections.

OWL [12], on the other hand, is a declarative language developed by the World Wide Web Consortium (W3C), which has become a *de facto* standard for authoring ontologies in the Internet. An ontology is defined as a set of statements used to describe an application domain. Rather than specifying procedures, OWL describes a domain: component elements and relationships among them. Using reasoning processes, that original information and relations can lead to new, unforeseen knowledge. The statements in an ontology are of two different specification classes: TBox (terminological) and ABox (assertional). The former defines the schema of the application domain elements, including concepts, relationship characteristics, domain constraints, etc. The latter defines specific “instances” of the applications domain, i.e. an actual scenario with a set of individuals and the relationships among them. Thus, TBox is sometimes compared with object-oriented classes, whereas ABox is associated with instances of those classes. In contrast to Java, OWL is a highly expressive, declarative language with several semantic features for representing application domains. OWL specifications are interpreted and processed by OWL reasoners to validate the information against the domain restrictions, and to infer new information.

While each of the languages is well-suited for the set of domains in which each of them is widely used, sometimes the need arises to combine them in the most seamless way possible. Combining them means overcoming their limitations to achieve the great expressiveness of OWL and, *at the same time*, allowing to perform algorithmic processes, input, and output with the data as seamlessly as possible.

This language combination then results specially useful in scenarios where applications have to manage knowledge-related information. For instance, Web-based distributed decision support systems or health care applications are just two examples which are usually based on rules and knowledge data to take decisions based on information provided by knowledge engineers. These applications are commonly composed of a knowledge-related part and a procedural part. The former keeps the rules and knowledge information, whereas the latter is in charge of carrying out the data processing tasks like, for instance, monitoring several patient parameters in case of a medical application.

There are a few proposals whose aim is to offer an effective binding between the Java and OWL languages. However, many of them do not resolve some of the following requirements, which have been the main motivation of this research work: i) effective isolation between the OWL knowledge engineer and Java developer roles which enable a collaborative development; ii) support the extensibility of existing Java applications, instead of being focused on new developments, i.e. support for legacy applications. Note that this requires a complete transparent abstraction layer; iii) easy integration of existing OWL ontologies in new or existing developments; iv) automatic injection of the inferred knowledge in one language into the other one at run-time stage (seamless bidirectional interaction); v) effective management of OWL inconsistencies in order to provide resolution capabilities to the inference process.

Thus, the main purpose of this research is to describe an architecture to enable an efficient binding between Java and OWL dealing with the previous requirements. This binding enables software developers to use both languages to build and extend their applications, resulting in applications composed of both OWL and Java. The declarative and semantic features of OWL are offered to Java applications at run-time and, at the same time, Java applications endow OWL ontologies with procedures for carrying out I/O functions and information processing. While other approaches allow Java programs to access OWL data and the result of the reasoning processes, it usually requires some hand-coded logic to collect the newly reasoned data, new instances created, etc. from the mapping-generated classes or from the OWL itself through some API. Our aim in this research is that new instances, and the modification of instance data values, happen automatically *as a result* of the reasoning process, and are incorporated into the application state as new Java instances and updated values of instance data members.

The rest of the paper is structured as follows: Section 2 describes the Java and OWL languages, providing a comparison of their main features. Section 3 provides an overview of current Java-OWL binding approaches motivating the necessity of our contribution. The proposed architecture is explained in detail in section 4. Then, Section 5 provides a description of the proof of concept implementation, and Section 6 offers a performance and scalability analysis of the implementation. Finally, the paper concludes with some remarks and future directions in Section 7.

2 Java and OWL Overview

RDF [25] is a structured language for representing application domains. OWL is built on top of RDF in order to extend its expressiveness. In this context, expressiveness is defined as the set of constructors that can be employed to describe the application domain. Current OWL 2 [28] proposal defines three different profiles [27] attending to the provided expressiveness and computational complexity, namely, OWL 2 EL, OWL 2 QL, and OWL 2 RL. The first two are targeted for particular ontologies with large number of classes/properties and instances, respectively, limiting their expressiveness in favour of high performance reasoning algorithms. In turn, OWL 2 RL is aimed at providing scalable reasoning without sacrificing too much expressive power. Concretely, this profile is suitable for reasoning over the ABox component, and the reasoners for this profile can be implemented by means of a general purpose rule engine. Furthermore, OWL can be extended with SWRL [19] to allow expressing conditional knowledge. Although SWRL is not decidable by itself, a syntactic restriction called DL-Safe context [4] can be applied to ensure decidability. Therefore, OWL 2 RL jointly with SWRL DL-Safe (henceforth called OWL/SWRL) have been selected as the declarative language for this research.

Regarding the Java language, recent versions include features such as annotations, parametric classes, and genericity, among others. This proposal makes extensive use of Java annotations, as explained later in Section 4.1, so at least Java version 5 is required.

From the declarative perspective, Java describes application domains using *classes*, *instances*, and *attributes*. A *class* is used in Java to model sets of world entities that are composed of variations of a given set of *attributes*. Thus, a Java *instance* is defined as a particular realization of a given class and all its attributes. This is one of the facts for which Java requires default attribute values and class constructors in order to know how to instantiate such values for a newly created instance.

OWL uses *concepts* and *properties* for describing the application domain. A *concept* represents a world entity. A *property* represents either a feature of a concept or a relationship between concepts. Properties can be associated with one or more concepts, indicating that the same property is used to describe such concepts. For example, the property *name* can be associated to a *User* concept and to a *Computer* concept to describe them. In this context, an OWL *individual* is defined as a particular realization of either a *concept* or a *property*.

Java mechanisms for specifying characteristics of the domain instances include classes and interfaces. Instances can aggregate different perspectives and properties by means of the interfaces they offer, and are related to others by offering a common interface. The meaning of those instance properties is established by the instance's only class. OWL, by contrast, orthogonally separates classes and properties, allowing individuals to belong to several classes, and to be related by arbitrary set of properties. While Java only allows single inheritance between classes, OWL offers multiple inheritance both between concepts and between properties, allowing a more fine-grained specifications. Also, more complex relations between OWL classes can be established. For instance, several classes can be stated to be *equivalent*, *disjoint*, etc., and even can be described by *extension* (specifying the set of elements that a class contains), by *intension* (specifying as instances elements that hold (or do not hold) a property), and even by *union*, *intersection*, *complement* of other classes. Both languages have a TOP-class which is the parent of all other classes. Additionally, OWL also defines a BOTTOM-class which is the child of all other classes. Finally, OWL allows for richer property relations to be stated: properties can be made *transitive*, *reflexive*, etc., opening the description of the domain to *future* knowledge inferred by these relations. Note the important semantic mismatch between the languages.

While OWL follows the *open world assumption*, Java follows the *closed world assumption*. Essentially, the open world assumption states that new, unforeseen knowledge can be incorporated to the reasoning process in the future. This fact turns OWL suitable to be used in open environments where domain descriptions are not closed, application knowledge may not be complete, and applications can exchange knowledge without having to know each other in advance. Java is limited, by its closed world assumption, to the types defined at compile time, and behaviour hard-coded in algorithms. Even when it can manage dynamic loading of classes, the set of *semantic* knowledge (i.e. relations) available to manage these new classes and instances is very limited.

This difference in the representation of world entity concepts, characteristics, relationships, and semantics have direct implications when a bridge between these language is established, since it entails that new OWL individuals can correspond with several Java instances and vice versa.

Regarding the imperative perspective, Java provides *methods*, *conditional statements*, *loops*, etcetera to control the program flow, whereas OWL does not provide any constructors for these purposes. However, real applications sometimes have to perform arbitrary processes on their data. To cope with that, OWL reasoners usually provide an API to manage data I/O into the reasoner, and to control the reasoning processes at run-time. In this sense, while Java provides advanced I/O functions directly mediated by the operating system, which enable not only local and remote communications but also data interchange with devices like mouse and keyboard using OS interruptions, OWL only enables the data exchange by means of URLs and data streams, leaving aside any interactive and OS interruption-mediated process in which input/output information from external devices is involved. Thus, this OWL reasoner API is used from programming languages like Java in order to build the final application that implements *how* the information is loaded and retrieved into/from the OWL reasoner and *how* such information is combined with graphical interfaces, data-processing libraries, and external devices. This kind of unidirectional binding Java-to-OWL has been widely covered in the design of OWL reasoners and is well established by now.

This research work aims to provide not only the usual binding of Java-to-OWL, but also the opposite OWL-to-Java, which allows the Java instances to receive all the inference provided in OWL automatically (that is, new Java object instances are created as a result of the reasoning process, and data values of instances are also updated with inferred values.) This fact is even more challenging when the proposed architecture is bound to fulfill the requirements previously described in Section 1, such as transparent integration and legacy application support. This is the clear differentiating value of your research work.

3 Related work

Some research works highlight the need of integrating object-oriented languages and ontological representations [13], providing case studies in health care and life sciences which would benefice of an integration between Java and OWL [34]. Fenzel et al [15] describe three different approaches to tackle the OWL and Java binding: direct, indirect, and hybrid mappings. The direct mapping converts OWL concepts and properties in Java classes and attributes whereas the indirect mapping defines a set of high level Java (meta-)classes and maps the OWL concepts and properties in Java instances of such high level classes. Finally, an hybrid mapping is defined as the combination of the direct and indirect approach.

Some well-known examples of indirect mapping are the OWL API [18] and Jena API [10] for managing OWL ontologies. They provide a Java and OWL mapping but they do not establish a bridge between the languages since normal Java instances are not involved of the semantics and vice versa.

An early attempt to establish a direct mapping among these languages is provided by [22]. This work automatically generates Java classes (and attributes) from an OWL ontology using a direct mapping, i.e. OWL-to-Java mapping. The idea behind this research work is to generate a set of Java classes (and properties) that semantically represent the same application domain described in the OWL ontology. This approach has limitations. Firstly, it produces a complex set of automatically generated code, not ensuring that the generated code matches 100% with the semantics available in the OWL. Secondly, this mapping do not cover the complete OWL 2 RL and SWRL constructors, losing semantics compared to our proposal. Moreover, authors only try to provide two analogous representations of the application domain. but they do not enable any bridge between the languages at run-time. That is, after this mapping, a new Java instance created at run-time will not be converted to an OWL individual and vice versa which is the main aim of our proposal.

Another proposal [14] fosters the usage of object-oriented languages to model ontologies. Authors establish a mapping from ontological concepts to object-oriented constructors. However, this proposal only covers the conceptual perspective and it does not provide a mapping among two real languages.

Several are the research works focused on developing an architecture to bind the Java and OWL languages. *Jenabean* [11], *RDFReactor* [37], *Kazuki* [2], *Owl2Java* [38] and *Jastor* [1] are some of the proposals to provide a Java-to-OWL and OWL-to-Java binding processes. *JenaBean* is a persistence API to save Java *instances* in RDF syntax. This tool only offers a syntactical conversion of Java instances and does not cope with any semantics mismatch at all. Thus, there is no real binding, just RDF is used as an XML Schema for describing Java instances.

The *RDFReactor* tool provides an effective mapping between RDF and Java. This is a complete direct mapping which tries to do a real binding between RDF and Java. However, it just works with RDF expresiveness and all the semantics and constructors provided by OWL and SWRL constructors are left aside. In contrast, they are fully covered in our proposal. Moreover, it is based on the automatic generation of new Java classes without attributes, but with get and set methods acting as proxy for retrieving the property information, stored in the RDF reasoner. Thus, this approach imposes an important syntactic restriction due to the fact that Java classes cannot contain real Java attributes managed at RDF level. This is an issue to enable the support for legacy applications that already have such attributes.

Kazuki and *Owl2Java* are similar tools that enable an OWL-to-Java binding by means of a complete direct mapping following an analogous approach to that provided by *RDFReactor*. These proposals incorporate a reasoner in the architecture in order to apply the OWL semantics, infer new knowledge, and to provide reasoning processes. However, both tools share the same limitations. Firstly, they do not deal properly with OWL inconsistencies and possible problems in the reasoning processes. Secondly, they do not hide the usage of the OWL reasoner to the Java developers. Finally, these approaches are not suitable in legacy systems, due to the mandatory automatic generation of new Java classes used as proxy access to the information stored in the OWL reasoner.

ActiveRDF [31] is an interesting proposal to perform a binding between the Ruby and OWL languages. This proposal does not rely on a schema and fully conform to RDF(S) semantics [26]. The key of this framework is to use RDF asserts with a non-triplet based API which allows the interaction with the RDF asserts in an object-oriented fashion. The advantage of this proposal is the selection of a dynamically typed language as Ruby for the binding. Since Ruby semantics are quite different from Java, the binding approach provided by that proposal is not directly applicable to Java based developments.

A more recent and interesting approach is *Jastor*. This tool is an evolution of the previous proposals and provides an OWL active object model to Java users. Java classes provide get and set methods implemented by dynamically accessing to the data stored in the OWL reasoner. This direct mapping approach enables the inclusion of an OWL reasoner, providing an efficient way of tackling the Java-to-OWL interaction. Despite the technological advantage with respect to its predecessors, this approach still has some lacks. Firstly, *Jastor* is based on a type-safe access. This kind of access makes the approach unable to dynamically assign attribute to classes which do not already have those properties statically defined [32]. Secondly, this approach does not cover the mapping of new OWL individuals generated at run-time into the Java instances. Thus, when new instances are generated by the inference processes, they are not reflected in the Java application. Note the importance of this fact since it could cause unexpected behaviours in real applications. Thirdly, *Jastor* does not provide methods to notify the Java program when any inconsistency arises in the OWL knowledge base. Finally, it is not able to hide the details of the OWL reasoner to Java developers, since it requires special mechanisms to create Java instances instead of using the standard *new* operator.

Recently, MOOOP [15] has appeared as another interesting proposal for carrying out the OWL-to-Java conversion. Authors propose an approach based on an hybrid mapping. While this kind of mapping provides very powerful capabilities and direct mapping between languages, it uses Java instances for representing OWL concepts and individuals. Thus, a real bridge between the languages cannot be established because Java instances belonging to *regular* Java classes do not receive the semantics available into the OWL.

Our proposal in this paper leverages current language mapping and binding proposals by providing a bidirectional bridge between the OWL and Java languages. OWL-to-Java is based in a simplified version of the direct mapping provided by Kalyampur, whereas the opposite conversion is a novel approach. In fact, the main contribution of this paper does not lay in the mapping done in both directions, but in the alternative approach used for establishing the bridge between languages.

Contrary to the analyzed related work, our approach recognizes the difficulty of representing the same semantics in both OWL and Java, and tries to keep the Java semantics and OWL semantics separate. All the declarative Java semantics can be directly mapped to the OWL semantics in a natural way (explained in the following sections) and it is only that subset of semantic constructors that is directly mapped into OWL at compilation time. Conversely, if a user starts the integration process from an OWL ontology, a set of Java classes is generated that contain only a subset of the OWL semantics coded. As an important number of the OWL semantics is not directly represented in Java code, the key aspect of our approach resides in the bridge performed between Java instances and OWL individuals at run-time. This bridge enables Java instances and OWL individuals to be synchronized between languages at any time. The Java instances can be directly translated to OWL individuals; then, the OWL reasoner can apply the complete OWL semantics over the OWL individuals to produce new knowledge (this is usually referred to as assertional reasoning or realization reasoning process.) This set of new inferred statements is the result of the application of the OWL semantics over the Java instances. The bridge then re-injects these new inferred facts into the Java instances, that now reflect the new values in their properties. From the Java point of view, we maintain a natural representation of Java classes and instances that get their properties seamlessly updated as a result of an OWL reasoning process. This effectively provides a bidirectional binding between both languages.

This proposal overcomes some lacking features of previous approaches, such as providing support for a complete semantic bridging between languages in which instances and individuals are kept synchronized between two different languages and they receive the semantics of the other language automatically at run-time. Moreover, it also supports the requirements stated in Section 1.

4 An Architecture to Bind Java and OWL 2 RL + SWRL-DL Safe

Software developments usually entail two different time stages: development and run-time. Each one attends to the creation of different software artifacts. While the former deals with the definition of the classes, interfaces, methods, etc. which conform the application, the latter manages instances of such elements. These two stages of development have driven the design of the proposed architecture, which has been divided into development and run-time stages. Following sections go into the architecture details which take part in each stage.

4.1 Development Stage

According to their needs, developers may start a new development by defining the application domain using OWL. This is the so-called ontology-driven development [23]. On the other hand, developers may start a new development defining the classes and processes by means of the Java language. This is the so-called class-driven approach. In case of legacy systems, the application may be already using an OWL ontology, or it may be implemented in Java. Regardless of the development strategy, the main purpose of the proposed architecture at this stage is to generate an equivalent model in both languages. The automatic generation of such models is performed by means of two different compilers: an OWL to Java compiler in case of ontology-driven developments and a Java to OWL compiler in case of class-driven developments. These compilers are a key part of the architecture, since they are in charge of generating output code with extra information which is needed to carry out the binding at run-time. The two approaches are covered by the architecture and they are described in detail in the following subsections.

4.1.1 Ontology-Driven Development

The ontology-driven development starts by describing an ontology. Thus, the first step might be to model its TBox component. This component describes all the domain concepts, relationships, restrictions, etc. As a result, an OWL ontology (without instances) is obtained that will drive the rest of the software development process. In this approach, Java classes should be automatically generated from the OWL ontology. These Java classes can be used in order to provide the procedural part of the application, dealing with I/O functions, etc. Advanced development environments such as Protégé [30] can be used to describe the application domain, restrictions, and semantics by using a declarative language like OWL, enhanced with SWRL rules and reasoners, which may reduce the need of defining Java code for this purpose.

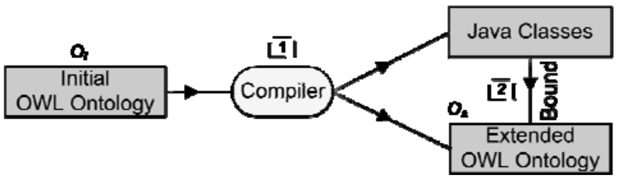

Figure 1 - OWL-Java binding process in a ontology-driven approach

Figure 1 shows the process in the architecture to achieve the binding in ontology-driven developments. The compiler receives the initial OWL ontology—concretely, the TBox component—and produces both a Java class hierarchy and an extended OWL ontology.

The main purpose of this compiler is not to generate a set of Java classes providing a Java equivalence for all the constructors available in OWL—that is the approach promoted by [22]. Instead, this compiler is intended to generate basic Java classes following a set of conversion rules, providing these classes with enough information for the architecture to be able to carry out the binding process of instances of such classes at run-time. This binding enables the synchronization of Java instances and ABox instances of the OWL ontology.

Table 1 shows the rules driving the conversion of the main concepts from OWL to Java. The compiler produces a Java interface for each OWL class, together with a class implementing that interface. This way, multiple inheritance of OWL classes, unavailable in Java, can be supported by using interfaces. Regarding properties, interfaces are also used in order to support OWL multi-domain properties. This time, each OWL property generates a Java interface exposing *get* and *set* methods. Then, all classes belonging to the property domain in OWL implement this interface. When multiple ranges are defined for an OWL property, the get and set methods use a *Map* to match each range with the value of the property associated to that range. This property conversion approach tries to overcome the Java type-safe access. If the reader is more interested on a detailed description of the problems derived of the type-safe access, multi-domain, multi-inheritance and how to overcome these issues, [22] provide a detailed explanation. Finally, unless a cardinality restriction is specified for a property, a Java *Collection* is used in order to enable multiple values of the property. The rest of OWL constructors do not affect the generation of the Java class hierarchy performed by this compiler. However, it is important to remark that these constructors are kept in the OWL ontology for further uses at run-time.

Concept	OWL	Java
Class A	A rdf:type class	interface IntA class A implements IntA
Inheritance	B rdfs:subClassOf A	interface IntB extends IntA
Property	P owl:ObjectProperty	Java object type
	P owl:DataProperty	Java primitive type
Cardinality	exact 1	N/A
	another cardinality	Collection < type >

Domain	P domain A	interface IntA f {
		setP();
		getP(); }

Range	P range A P range B	Map<range,value>

Table 1 - Conversion rules for the OWL to Java compiler

Besides the generation of the Java classes and properties that represent the concepts defined in the original OWL ontology, additional information should be added to both Java classes and OWL ontology to enable instance synchronization at run-time. For this purpose, the Java classes generated by the compiler contain information by means of Java annotations. On the other hand, the compiler also adds information to the OWL ontology to reference the Java classes using the OWL URIs available in the ontology. Thus, the compiler output is composed of the generated Java classes with annotations plus an extension to the original OWL ontology which includes some needed extra information for the alignment between both languages. This alignment is what the label #2 of Figure 1 represents, i.e., the Java classes and the final OWL ontology are bound.

Java annotations are a set of syntax resources that can be inserted in classes, attributes and methods. These annotations provide meta-information about the annotated resource that can be conveniently extracted. The annotations added by the compiler to the generated Java classes determine the Java elements which have been created by the presence of a constructor in the OWL ontology. The annotations employed are shown in Table 2 *@OWLClass* annotation defines a Java class-OWL concept alignment. Likewise, *@OWLDataProperty* and *@OWLObjectProperty* define an alignment between Java attributes and OWL properties.

Annotation	Scope	Binding
@OWLCLass	Class	Java Class - OWL Concept
@OWLDataProperty	Attribute	Java attribute (primitive type) - OWL Datatype Property
@OWLObjectProperty	Attribute	Java attribute (Object) - OWL Object Property

Table 2 - Annotations Used in the Alignment

Notice that only those Java classes and attributes with annotations are generated from OWL classes and properties. The rest of Java elements which may be lately inserted by the developers, including other classes, interfaces, attributes, etc. are not annotated and thus are not bound. This distinction provided by the annotations is useful to differentiate utility code (meaningless from the ontology point of view) from Java elements bound to the ontology. It is also important to remark that, for the alignment process, the compiler needs to know the package in which to insert the Java classes, as well as the URI prefixes to be used in the ontology.

For the legacy ontology-driven applications in which the Java classes are already generated, they only need to be extended with the Java annotations in order to attach the alignment information between OWL and Java languages.

Regarding OWL, the original ontology is extended in order to bind the concepts with the newly generated (or legacy available) Java elements. In particular, each OWL class and property that do not match with the associated Java class or attribute is extended with an equivalent statement, used to hold an alternative URL that references the corresponding Java element. As a result, the extended ontology is able to refer all the OWL classes and properties by means of an alternative URL. This URL follows the following pattern: *<original URL prefix> + "#" + <Java package> + "." (dot) + <OWL class name* **or** *OWL property name>*. This URL contains enough information to link the OWL and Java elements, since both identifiers—fully qualified Java class name and OWL concept name—can be obtained from the URL. This URL is one of the key of the proposed architecture.

The reason for the final ontology containing both the original and extra information lies in the need to ensure backward compatibility: the extended ontology can be still used in legacy systems. In case this feature was not relevant, the original elements together with the added information can be removed to obtain a more reduced, optimized ontology via a compiler option at development stage.

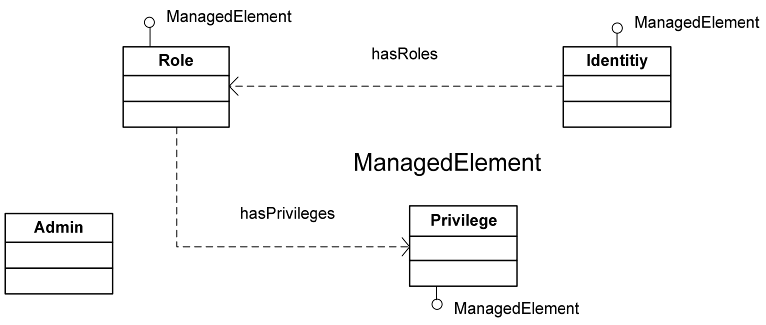

Figure 2 - OWL example diagram.

In order to clarify the concepts presented in this proposal, a working example has been used throughout the paper, illustrating different aspects of the approach. Let us suppose the set of classes and properties depicted in Figure 2. The figure shows a simplified example of a scenario used in our previous research in which the combination of Java and OWL was used to implement an authorization system. OWL was used to implement the authorization theory, whereas Java implemented the final system (the details are available in [9].)

Listing 1 shows these concepts in OWL functional syntax. This syntax intends to provide a human readable format for representing an OWL ontology.

```
URL Prefix <um,"http://um.es">
Class(um:Role); Class(um:Admin)
Class(um:Identity); Class(um:Privilege); Class(um:ManagedElement);

SubClassOf(um:Privilege um:ManagedElement)
SubClassOf(um:Role um:ManagedElement)
SubClassOf(um:Identity um:ManagedElement)

ObjectProperty(um:hasPrivilege); ObjectPropertyDomain(um:hasPrivilege um:Role);
ObjectPropertyRange(um:hasPrivilege um:Privilege)

ObjectProperty(um:hasRole); ObjectPropertyDomain(um:hasRole um:Identity)
ObjectPropertyRange(um:hasRole um:Role)

DataProperty(um:name); DataProperty(um:instanceID); DataProperty(um:authorized)
DataPropertyDomain(um:name um:Role); DataPropertyDomain(um:authorized um:Role)
DataPropertyDomain(um:instanceID um:Identity); DataPropertyRange(um:name xsd:URI)
DataPropertyRange(um:instance xsd:URI); DataPropertyRange(um:auhtorized xsd:boolean)
```

Listing 1 - OWL Example

Listing 2 and Listing 3 show the compiler-generated Java code and the extended ontology generated by the compiler for the original ontology in Listing 1.

```
package exampleA;

@OWLCLass
public class  Role extends  ManagedElement  implements  intRole, intHasRole  {

        @OWLDataProperty
        HashMap<Class, String> name;
        @OWLObjectProperty
        HashMap<Class, Collection<Privilege>> hasPrivileges;

        public Role(){
             this.name = new HashMap<Class, String>();
             this.hasPrivileges = new HashMap<Class, Collection<Privilege>>();
        }
        public Object  getName(Class type){
             return name.get(type);
        }
        public void  setName(Class type, String value){
             name.put(type, value);
        }

        public  Collection<Object> getName(Class type){
             return name.get(type);
        }
        public void  setName(Class type,Collection<Privilege> value){
             name.put(type, value);
        }
}

public interface intRole extends  intManagedElement { }
```

Listing 2 - A fragment of the Java code generated by the compiler for the input shown in Listing 1

Listing 3 shows only the extended information inserted in the final OWL ontology but the reader may note that this the final ontology also contains all the information available in the original ontology (see Listing 1). The Java package specified is *um.authorization*, and the OWL URL prefix is *http://um.es*. It can be observed in the listing how the original concept *http://um.es#ManagedElement* can now also be referenced as *http://um.es#um.auhtorization.ManagedElement* due to the *equivalentClass* assertion between the original and the generated concept in the extended ontology.

```
URL  Prefix <um,"http://um.es">

EquivalentClass(um:um.auhtorization.ManagedElement, um:ManagedElement)
EquivalentClass(um:um.auhtorization.Role,  um:Role)
EquivalentClass(um:um.auhtorization.Identity, um:Identity)
EquivalentClass(um:um.auhtorization.Privilege, um:Privilege)
EquivalentClass(um:um.auhtorization.Admin, um:Admin)

EquivalentProperty(um:um.auhtorization.Role.hasPrivilege,  um:hasPrivilege)
EquivalentProperty(um:um.auhtorization.Identitiy.hasRole, um:hasRole)
EquivalentProperty(um:um.auhtorization.Role.name, um:name)
EquivalentProperty(um:um:um.auhtorization.Role.instanceID,  um:instanceID)
quivalentProperty(um:um:um.auhtorization.Role.authorized,  um:authorized)
```

Listing 3 - OWL generated by the compiler for the input shown in Listing 1.

This resulting compiler output contains the information needed to enable the alignment of concepts and properties between both languages. A Java class can now be identified from the information available in the URL of its corresponding OWL concept. Likewise, the URL to identify an OWL concept or property can also be generated from the annotations available in the Java code. This alignment information is used at run-time to dynamically bind the instances between Java and the ABox component of the ontology.

4.1.2 Class-Driven Approach

A class-driven development strategy is the most common way to tackle software developments. Under this point of view, the software development process starts modelling the application domain by means of a set of classes that drive the rest of the development process.

Developers can then use the architecture presented herein to embed the advantages of OWL and reasoning into their development process. In this case, the aim is to generate an OWL ontology aligned with the existing Java class hierarchy to enable the instance synchronization at run time. Figure 3 graphically shows this process.

The first step is aggregating extra descriptors in the initial Java class hierarchy to help the compiler determine which information is relevant for the OWL ontology. Usually Java developments include data structures, actions, listeners, graphical interfaces, entry points, etc. that are intrinsic of the Java language or library itself, and thus are not part of the data modelling of the application, so are out of the scope of the declarative part represented in OWL. Therefore, those classes and attributes that will conform the declarative part of the new OWL ontology have to be explicitly stated by the developer.

Specifying the classes and attributes relevant to the OWL is performed by adding annotations, so it does not produce any change in the actual Java code. shows the different annotations that can be used by developers. Whereas the *@OWLClass* is used to annotate Java classes, the *@OWLDataProperty* and *@OWLObjectProperty* annotations are used to indicate the Java attributes that are going to be converted to OWL.

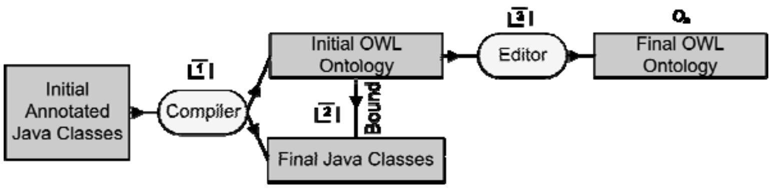

Figure 3 - Java-OWL alignment process in the class-driven approach.

The compiler receives as input this annotated Java classes and produces both an OWL ontology and a new version of the initial Java classes. To properly generate the OWL ontology, the compiler also has to interpret the Java reserved words *extends* and *implements*. Table 3 shows the conversion rules applied. In summary, an *@OWLClass* annotation produces an OWL class, whereas both *@OWL-ObjectProperty* and *@OWLDataProperty* annotations produce OWL properties with its corresponding range and domain. Likewise, the Java reserved words *extends* and *implements* produce a *subClassOf* OWL asserts (inheritance in OWL). Additionally, the OWL URLs generated should follow the same pattern previously described in subsection 4.1.1.

Java code	OWL Results
@OWLClass public class C	C rdf:type owl:Class

@OWLClass public class A implements B	A rdf:type owl:Class A rdfs:subClassOf B

public class A {
@OWLObjectProperty
type P; }
	P rdf:type owl:ObjectProperty
	P rdfs:domain A
	P rdf:range *type*

@OWLClass public class A extends B	A rdf:type owl:Class A rdfs:subClassOf B

public class A {
@OWLDataProperty
type P; }
	P rdf:type owl:DataProperty
	P rdfs:domain
	A Prdf:range *type*

Table 3 - Java-OWL Conversion Rules in the Class-Driven Approach

Either in class-driven or ontology driven development, the new auto-generated OWL ontology can be extended with new information about the application domain using the expressiveness of OWL and SWRL. The expressiveness provided by the OWL 2 RL + SWRL DL-Safe language includes cardinality restrictions, complex class definitions, multiple inheritance, multiple domain and range, semantic rules, etc. Notice that during this extension, the generated base ontology should be preserved in order to keep the mapping with the Java classes. The incorporation of this information can be done by means of OWL and SWRL editors. This step is labeled as #3 in Figure 3. Some examples of well-known OWL editors are *Protégé* [30] and *Swoop* [21] whereas among rule editors *VDR-DEVICE* [5] and *ORE* [29] can be found (*ORE* itself has been implemented in our research group and offers an easy way for developing and debugging SWRL rules.) As a result, an ontology with a full description of the application domain (with constraints, goals, rules, etcetera) is obtained.

The reason why new Java classes have to be generated (not using the current ones) by the compiler is mainly to support the multi-range and multi-domain features of the OWL properties inside the Java code. The new Java classes are essentially the original ones with some more information. The compiler uses the new OWL ontology as a guideline about how the original Java classes should be extended. Table 4 shows the conversion rules used in the compiler. These rules are analogous to those exposed in Table 1. Mainly, each OWL concept produces an empty interface which is implemented in the associated Java class. Each OWL inheritance produces the implementation of an interface in the Java class. Moreover, each OWL property produces the implementation of an interface for each Java class that may contain this property.

OWL Element	**Original Java Element**	**New Java Element**

A rdf:type Class	class A { ... }	class A implements IntA { ... }
		interface IntA { ... }

B rdfs:subClassOf A	class B { ... }	class B implements IntA,IntB { ... }
P owl:ObjectProperty P owl:DataProperty	Java property reference Java primitive type	Java property reference Java primitive type
P cardinality != 1	Java type	Collection < type >

P domain B
P range A
	class B { A p;
	...
	}
		class B implements intP { A p;
		setP(A p){ this.p = p; }
		A getP(){ return this.p; }
		... }

Table 4 - Conversion rules used in the Java to OWL compiler

At the end, both ontology-driven and class-driven approaches achieve the same results. Thus, the class-driven approach takes a Java class hierarchy with some annotations to enable the Java-OWL alignment. On the other hand, the ontology-driven approach takes the OWL ontology containing not only the basic application domain expressed in Java classes, but also the constraints, goals, and policies described in OWL to make the alignment. These two approaches align the Java code and the OWL ontology in order to enable the afterwards instance conversion between Java and OWL. The following section describes how the architecture presented herein deals with these instances at run-time.

4.2 Run-Time Stage

Once the alignment between OWL and Java has been established at development stage, the run-time phase provides a framework where both the Java application and the OWL ontology reasoning can be executed. This stage aims to make the Java instances available to OWL individuals and vice versa. This multi-representation capability supports the bidirectional conversion and, in turn, establishes the real binding between these languages at run-time. Thus, all the OWL reasoning facilities are directly enforced in the Java application and changes in Java instance values are reflected in the OWL. Moreover, the architecture helps the Java developer with the synchronization process between the languages by managing the invocation of the OWL reasoning services and handling the Java and OWL instances synchronization automatically.

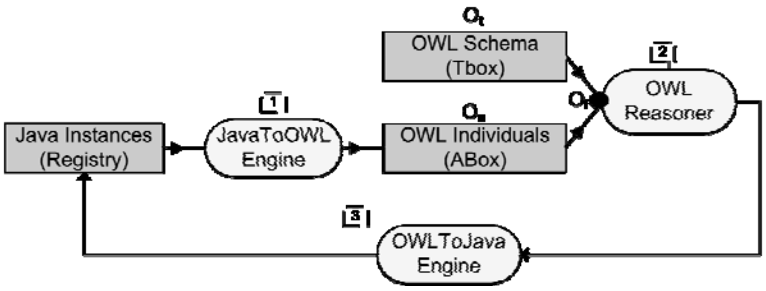

Figure 4 - Process for semantic enrichment at run-time.

Figure 4 shows the different processes defined in the architecture to carry out the run-time stage. This process can be triggered either on demand, when the Java application requires using the OWL reasoning services, or periodically, when the architecture has been programmed beforehand to perform the synchronization between the two languages. Initially, the process starts converting the Java instances into OWL individuals by means of the conversion module (see #1 in Figure 4). After that, these individuals are used in the OWL reasoner to perform the reasoning process (see #2). Among others, the OWL reasoner is in charge of inferring new information related to the application domain and checking the consistency of the information (constraints verification, exceptions, etc). Finally, the new inferred knowledge is re-inserted towards Java instances thanks to the enforcing module (see #3). These three processes are carefully explained in the following subsections.

4.2.1 Conversion Module

As stated before, the conversion process is the first step carried out at the run-time stage in order to achieve the Java-OWL synchronization. This module has the capability to manage the instances available in both languages. While the OWL reasoner uses a knowledge base in which all instances are registered and available to be retrieved, Java requires to develop an additional mechanism to register and manage the instances involved.

This registration process takes into account only those Java instances that are required to be converted between the languages. The main purpose is to keep a registry (implemented by means of a data structure) of the instances related to the application domain information.

The registry is defined as a `Map<Class, Collection<? >>`, a map structure where the elements are indexed by the fully-qualified name of the Java classes which are going to be managed. Besides, these classes are used to retrieve the collection of instances belonging to such classes. Notice that the collection of instances is defined as an abstract collection that can be refined as arrays, lists, linked lists, etc. Moreover, this collection has been defined as unknown type (genericity) due to the fact that the type of these instances is unknown until run-time.

As a result, the registry component is able to register all the Java instances related to the definition of the application domain. This data structure has been optimized to perform instance searches as well as to register large amounts of instances efficiently. Additionally, the registry exposes methods to manage the set of instances of a Java class. An example of the registration process of our running example can be shown in the code comments as "‘instance registration"’ in Listing 4 which represents a complete example of the framework proposed.

Once the Java instances are registered, the synchronization can be launched. The conversion module retrieves all the information available in the registry to generate the corresponding OWL individuals. The instances registered are converted using the conversion rules described in Table 5. Each Java instance generates an OWL individual belonging to such a class. Likewise, each Java attribute with *OWLObjectProperty* or *OWLDataProperty* annotations generates an OWL individual containing the value of the property.

Java Code	OWL Description
ManagedElement example = new ManagedElement();	prefix:example rdf:type prefix:ManagedElement
example.setID(value);	prefix:example prefix:ID value

Table 5 - Conversion rules between Java and OWL instances at run-time.

The conversion process uses a bidirectional algorithm that permits aligning OWL individuals and Java instances by means of uniquely identifying each instance. Since OWL uses URLs to refer individuals and Java uses object identifiers (OID), an equivalence has to be devised. The algorithm unifies both approaches by generating OWL URIs containing information about both the Java OID and the name of the Java class of such instance. As a result, the generated URL has enough information to uniquely identify a Java instance in the registry. This fact enables the architecture to retrieve any Java instance from a given OWL individual and vice versa. The algorithm in charge of building the URLs uses the following pattern: <*OWL Prefix*> + ’#’ + <*statical type of Java instance*> + ’$’ + <*OID associated to such instance*>. The static type of the Java instance used in the pattern, i.e. the Java class, can be used later as the index of the map structure to select the right collection of instances (`Collection<? >`) belonging to such Java class. Likewise, the OID part in the URL can be used to retrieve/insert the instance from/in this structure.

The conversion module uses the algorithm described above in order to (re-)create the ABox component (the set of individuals representing current system state): it uses the URL pattern to create as many OWL individuals as Java instances are hold in the registry. This ABox component is used afterwards by the reasoning module as explained in the following subsection.

4.2.2 Reasoning Module

The OWL reasoning processes need two different sources of information in order to carry out the inference process. On one hand, the TBox component (labelled as O_t in Figure 4) is needed. This component contains the schema of the declarative part of the application domain, and is generated taking into account the OWL ontology described at the development stage. On the other hand, the ABox component (labelled as O_a in Figure 4) is provided by the conversion module at run time. Thus, this generated ontology contains the instances hold in both languages when the synchronization is performed.

The ontology of the TBox component can contain any constructor among the ones provided under the OWL 2 RL and SWRL DL-Safe languages. There is no limitation in the usage of the provided constructors. For example, the application domain contained in the ontology can use: i) complex class definitions—unionOf, complementOf, intersectionOf, disjointWith—; ii) cardinal restrictions—maxCardinatiligy, minCardinality, exactCardinality—, enumerations—oneOf—; iii) class descriptions —equivalentClasses, disjointClasses—; iv) property descriptions —transitiveProperty, inveseProperty, functionalProperty—; v) semantic rules—SWRL rules. To sum up, all the expressiveness supported in OWL 2 RL + SWRL DL-Safe.

The reasoning process receives as input these two sources of information (TBox and ABox) and, relying on the OWL reasoner, carries out the inference process. The complete process which includes the conversion from Java to OWL, the reasoning process and the enforcing of the inferred facts again into the Java instances is completely hidden for the developer and it is done internally. Thus, Listing 4 only summarizes this complete process invoking the method "‘owl.syncronizeLanguages()"’.

An OWL reasoner can perform reasoning processes over both TBox and ABox components. For example, with regards to the TBox component, the reasoner can classify the ontology concepts in order to discover initial hidden relationships in the schema. Moreover, the reasoner can also ensure that there is no inconsistencies in the definition of the domain schema. Regarding the ABox component, the reasoner can infer new information related to individuals using the information contained in the TBox. Moreover, it can ensure that all the constraints, goals and rules are fulfilled and the knowledge base is consistent.

OWL comply with an open world assumption by which further knowledge about the application domain can be inserted in the ontology when needed. This knowledge can provide new information related to both ABox and TBox components. New knowledge inferred about the TBox component may imply changes on the definition of the Java classes, which is not an easy task to accomplish in a Java application at run-time. On the other hand, changes in the ABox component affect only to instances, which is easy to manage in Java. Since changes on instances are more common at run-time and changes on classes are very unlikely to happen at this stage, the reasoning process exposed herein is focused only on the ABox. Future work could consider the management of TBox changes.

As stated in Section 2, the OWL 2 RL profile together with the SWRL DL-Safe extension has been selected to perform the reasoning process. This OWL 2 profile is suitable to perform reasoning processes over the ABox component and the reasoners for this profile can be implemented by means of any common rule engine.

The reasoner generates two kinds of outcomes after performing the reasoning process. On one hand, the reasoner infers new information about the ABox component. This information is the result of applying the semantics available in the application domain schema (TBox component) over the instances present in the ABox component. For example, new individuals can be inferred, as well as new property values, new class memberships, etc. On the other hand, the reasoner also checks that the individuals fulfill all the constraints about the application domain. In case that any constraint is violated, an inconsistency arises, notifying the Java program. In fact, when an inconsistency is detected, a Java exception is thrown in order to notify to the Java application about the reasons that have led to this inconsistency. This enables developers to react when any constraint is violated in OWL. (see the code comment "‘inconsistent management in Listing 4). As a result, the reasoner module provides the enforcing module with the newly inferred information.

4.2.3 Enforcing Module

The enforcing module is in charge of inserting the inferred information provided by the OWL reasoner into the Java instances at run-time (this module is labeled #3 in Figure 4). That information represents changes that should be applied in the Java application. In particular, due to the limitations in the reasoning services previously described in Section 4.2.2, for the time being, the inferred information is related only to OWL individuals. Thus, a new individual belonging to a concept is enforced by creating a new Java instance belonging to the corresponding class. The new instance is also properly stored in the registry.

In turn, a new attribute value is enforced by establishing its value in the associated instance. In case this value is a reference to another object (*ObjectProterty*), this reference can also be retrieved from the registry.

Some of these conversion rules may entail retrieving the Java instances in order to update its state. Thereby, when a new attribute has to be enforced, the instance retrieval can be done using the URL associated to the inferred individual. This can be done since the OWL URLs contain both the name of the Java classes and the OID reference.

In order to clarify this enforcing process, let’s suppose the scenario shown in Listing 1. Listing 4 shows a Java main class that makes use of the original classes generated at development stage. This scenario has been slightly extended inserting a SWRL rule labelled as 1 like *Role(?role) ∧ name(?role, ladminl) → authorized(?role, true)*, which establishes that all the instances belonging to the role *admin* will be set *true* in the property *authorized*. Another SWRL rule, labelled as 2, can be stated to enforce that identities belonging to the role admin are also instances of a special class called *Admin* defined as *Role(?role) ∧ name(?role, ladminl) ∧ Identity(?i) ∧ hasRole(?i, ?r) → Admin(?i)*. The fact of including these rules does not entail the modification in the Java classes generated at development stage, which are shown in Listing 2.

```
public static void  main()  {
     Collection<Role> roles  = new ArrayList<Role>;
     Collection<Identity> identities = new ArrayList<Identity>;
     Collection<Privilege> privileges  = new ArrayList<Privilege>;
     Collection<Admin> admins = new Collection<Admin>();

     Identity john  = new Identity();
     Role admin = new Role();
     identities.add(john);
     identities.hasRole(admin);
     roles.add(admin);

     // instance Registration
     OWLBinding  owl  = new OWLBinding("TBox.owl","http://um.es");

     owl.register(Role.class,  roles);
     owl.register(Identity.class,  identities);
     owl.register(Privileges.class,  privileges);
     owl.register(Admin.class,  admins);

     // language  synchronization
     try{
                  owl.synchronizeLanguages();
     }catch(InconsistencyException  ie){
            // Inconsistency Management
     }

     System.out.println(roles.getByName("admin").getAuthorized());
     // The system should  print ’true’
}
```

Listing 4 - Program example to clarify the complete binding process at run-time stage

Listing 4 shows how two different classes *Role* and *Identity* are being instantiated as well as how the registration and the synchronization process is performed after that. Therefore, these instances are converted to OWL individuals and the reasoner infers the results. A particular piece of results are shown in Listing 5.

```
admin rdf:type ManagedElement    // Role extends  ManagedElement,  john  instance of  Role (1)
admin rdf:type Admin             // SWRL  rule 2 (2)
admin um:authorized ’true’       // admin instance of  Role and SWRL  rule 1 (3)
```

Listing 5 - Inferred information

Together with the inferred knowledge, a brief explanation of how this knowledge has been inferred is included. (1) states that, since class *Role* extends class *ManagedElement* and *admin* is an instance of *Role*, then *admin* is also an instance of *ManagedElement*. (2) is inferred when applying the aforementioned SWRL rule 1, stating that instances of *admin* should set *authorized* to *true*.

Taking as input the inferred information detailed above, the enforcing process behaves for this example as follows. The first inferred fact, shown as (1) in Listing 5, states that the instance *admin* belongs to the class *ManagedElement*. This is already implemented in the Java semantics by polymorphism, so no action is performed. (2) states that the instance *admin* should belong to class *Admin*. To enforce this information, a new Java instance *i* is created and inserted in the registry (this instance will be available in the `Collection<Admin>`) and *i* and *admin* are grouped in the registry, so they can be converted into the same OWL individual in further synchronizations. (3) states that instance *admin* should set up the value ’true’ in the *authorized* property. This is enforced by retrieving the *admin* instance from the registry and setting up the value. As a result, all the Java instances have received all the information provided by the OWL ontology and vice versa, achieving a real binding between Java and OWL at run-time.

5 Implementation

The architecture described in this proposal has been implemented as a prototypical framework. This framework has been released to the community as an open-source and free project under the acronym JASB (*Java Architecture for Semantic Binding*) at SourceForge[1]. The architecture provides a binding between Java and OWL. The framework is basically composed of two different tools. On the one hand, a cross-compiler obtains OWL representations from Java classes and vice versa. On the other hand, an API manages the binding between the two languages at run-time.

The cross-compiler tool is able to interpret a set of Java annotations, which are then used to perform the conversion between the annotated Java classes and the corresponding OWL representation. The annotations are shown in Table 6. Additionally, some annotations have also been implemented in order to enable Java developers not only to decide the classes relevant in the binding process, but also to annotate some OWL constructors directly in the Java code. The idea is to provide a basic OWL authoring feature in the Java code. Then, in case the developers use these new annotations in the Java code, the compiler will generate the corresponding OWL constructors. Table 6 shows the additional implemented annotations with its associated OWL constructors.

Annotation	Scope	Binding
@OWLEquivalentClass	Class	Equivalent classes
@OWLDisjointWith	Class	Disjoint classes
@OWLComplementOf	Class	Complementary classes
@OWLTransitive	Attribute	Transitive property
@OWLSymmetric	Attribute	Symmetric property
@OWLReflexive	Attribute	Reflexive property
@OWLSubPropertyOf	Attribute	Subproperty
@OWLKey	Attribute	Key property
@OWLIrreflexive	Attribute	Irreflexive property
@OWLFunctional	Attribute	Functional property
@OWLAntiSymmetric	Attribute	Antisymmetric property

Table 6- Additional Annotations used as built-in OWL editor

A Java API has been implemented to enable Java developers to use the binding process between the languages at run-time. This API provides methods for registering Java instances in the registry—conversion module—, specifying OWL ontologies containing the schema of the application domain (TBox)—reasoning module—, and carrying out the synchronization process between languages. The API has been carefully designed to help developers in these tasks, hiding the details about the usage of the OWL reasoner. Thus, developers just have to invoke basic methods to schedule the synchronization process. Moreover, the Java API also provides some Java exceptions to notify inconsistencies during the OWL reasoning process.

Regarding the OWL reasoner used in the implementation, three different engines, *Pellet* [35], *Jena* [10], and *Jess* [16] have been taken into account during the JASB framework development, in order to provide compatibility support for them. All of them support OWL 2 DL and SWRL expressiveness.

The architecture has been fully developed in Java, i.e., the compiler, the API used by application at run-time, and even the external services provided by the OWL reasoners. This allows the architecture to inherit the Java platform-independence nature.

[1] JASB (*Java Architecture for Semantic Binding*) is available at http://sourceforge.net/projects/jasb

6 Performance Analysis

An analysis of performance and scalability of the JASB architecture is provided in this section. The aim it is to demonstrate the usability of this architecture when it is applied to realistic scenarios. The proposed testbed has been executed in a computer with the following hardware and software features: Intel Core 2 DUO T7500 2.2Ghz, 4Gb RAM, Linux Fedora 15 and Java JRE v1.7.0.

The performed test compares the time spent on the reasoning task with the time required by the whole architecture during the run-time stage. The idea is to provide the overhead imposed by the usage of the architecture proposed in comparison with an isolated OWL reasoner. This can provide an idea about timing related to the synchronization between Java and OWL instances at run-time stage.

Times have been taken considering the whole process described in Section 4.2, i.e. they include the time spent by the JavaToOWL Engine, the time spent by the OWL reasoner and the time spent by the OWLToJava Engine. It should be noticed that the reasoning process is performed by an OWL reasoner and takes a fair amount of time. Thus, comparing this time with the global time spent by the whole architecture will provide a way to weight up the overhead introduced by the architecture and therefore a mechanism to analyze the performance and scalability of the whole proposal.

In this sense, we first carried several performance test with different combinations of OWL/SWRL reasoners including Jess, Jena, Pellet, and combination of them. As a result, we chose the combination Pellet-Jess because it achieved the best performance for this testbed. Using the faster reasoner leads to lower reasoning times during the tests, thus highlighting the architecture overhead when compared to this time. The best reasoner is then the worst case of the scalability study of the proposed architecture, since the time associated to the architecture is higher.

The tests are performed using Java classes generated in an ontology-driven development approach, as described in Section 4.1.1. We chose the GOM ontology [24] to generate the Java classes and to perform the evaluation. This ontology is based on the Common Information Model (CIM), which is a broadly adopted standard that enables the representation of information systems. It requires SHOIN [20] expressiveness and it has around 1,600 concepts, 4,600 object properties and 21,000 data properties. The populations are generated using an ad-hoc guided-random algorithm to generate the group of individuals that conform to each population. The purpose of this algorithm is to generate a distribution of instances of OWL elements which make up the populations, at the same time it is covered the whole SHOIN expressiveness. The testbed is composed of eight incremental populations, where the highest one has 100,000 individuals. It should be noted that the reasoner requires a high amount of facts to represent a population of individuals, being this size quite higher than the number of individuals. For instance, a population of 40,000 OWL individuals randomly generated produce around 125,000 facts. The idea is to show how the architecture behaves when increasing of instances managed at run-time.

Figure 5 shows the results of the test. The numbers labelled in the graph represent the overhead in the performance for the usage of the architecture. The architecture does not require a significant increase in the execution time when it is compared to the time spent on the reasoning task. This overhead introduced by the architecture ranges from 6.88% for the first population to 10.58% for the last one. Moreover, this time follows a constant linear trend across the different analyzed executions. This fact remarks a good ratio of performance, scalability and usability in the implemented framework.

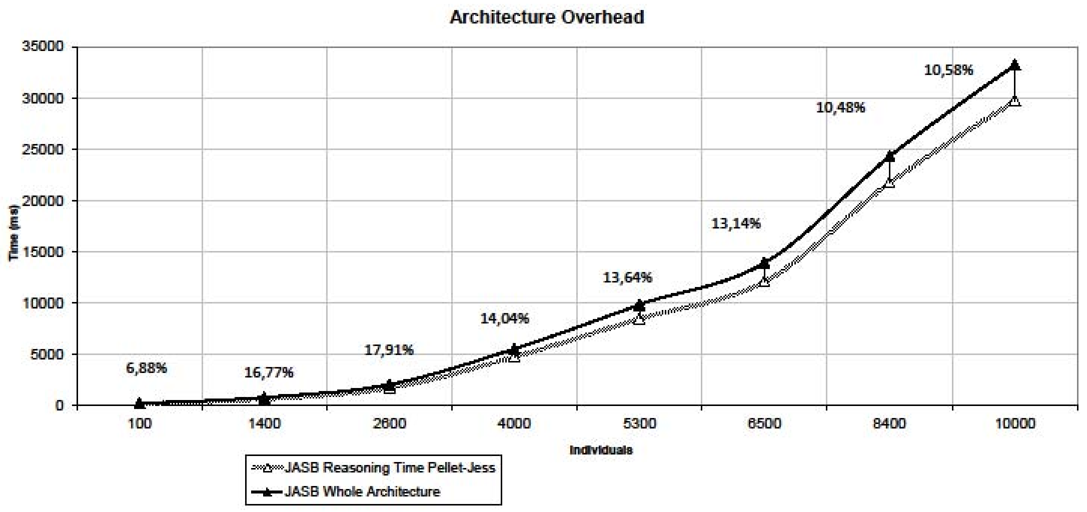

Figure 5 - Difference between the whole architecture and the reasoning times.

The evaluation performed so far deals with the performance achieved by our proposal. However, with the aim of evaluating this approach with others architectures, Figure 6 shows a comparison between *JASB* and *Jastor* overheads. This comparison can help to weigh up the previous achieved results. To carry out this analysis an scenario has been designed where both architectures make use of the same populations of individuals, the same OWL ontology as well as the same OWL reasoner. The overheads shown in the Figure 6 represent the overhead of the language binding architectures with respect to the simple reasoning time without any architecture. This metric enables us to analyze the computational complexity introduced by such architectures.

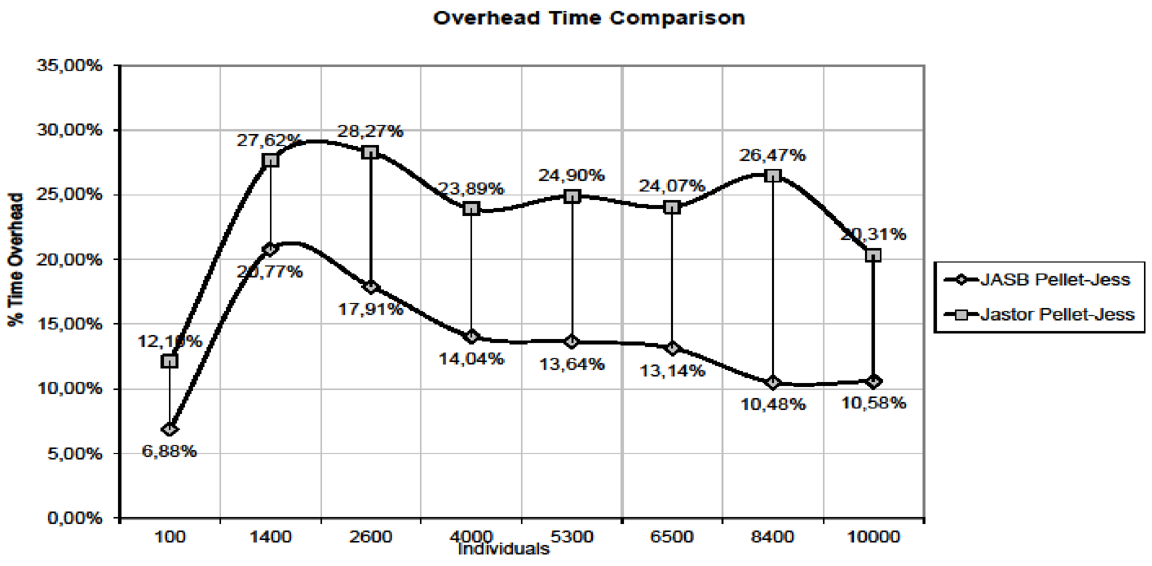

Figure 6 - Comparison between JASB and Jastor architectures.

As can be seen, *Jastor* is a bit faster than our proposal (less overhead percentage). This is due to the overhead included by the additional processes required by our approach to convert the instances between both languages. Since *Jastor* uses directly as active data model the OWL knowledge base, it does not require any conversion process. However, this fact causes that *Jastor* can not work properly with some scenarios at run time, for example, to register new individuals from the Java-side at run-time. Additionally, our architecture provides some features not available in *Jastor*, such as higher level of abstraction for developers when dealing with the OWL reasoner or backward compatibility with legacy systems. The comparison between the architectures have settled that *JASB* needs just a 11.5% more time to perform the binding than *Jastor*. The comparison also shows a constant trend in the increase of the time throughout the different executions. Therefore, it can be said that not only *Jastor* but also *JASB* are scalable and usable in real scenarios. Notice that in general terms as far as the population is being increased, the overhead is being reduced. The reason of this time decreasing is the time increasing of the time spent by the reasoning process. The bigger the population is, the more time is required by the reasoner. This reasoning time causes that the overhead of the language binding architectures is being reduced perceptually with respect to the reasoning time

7 Conclusions and Future Work

An architecture to establish a bridge between the Java and OWL languages has been provided. This architecture enables to program in both languages sharing programming facilities between them. The approach provides a bidirectional binding and it is able to deal with both the development and run-time stages. Different software development strategies have been successfully supported by the architecture. As a result, an speed up in the software developments can be reached combining the facilities provided by both languages. For example, developers can define semantics, rules and constraints related to the application domain in OWL whereas graphical interfaces, I/O functions and actions are implemented in Java.

The overhead introduced by the proposed architecture has been successfully tested, showing that this overhead is minimal when compared to the time required for the reasoning process. Support for legacy Java applications is also provided, making these applications able to extend their functionality by including OWL features. Additionally, an efficient way to detect and manage inconsistencies in the application domain and to notify it to the Java application has been provided by means of the usage of exceptions.

Regarding future works, an expected step is to extend the provided binding at real-time to be able to lead to Java code the changes that affects to the TBox. We are also currently working on integrating this architecture in a visual development tool to combine model-driven developments with the programming facilities provided in both the Java and OWL languages. Another future work is the inclusion of new languages in the architecture in order to enable a multi-language binding.

Funding

Thanks to the Fundación Seneca for sponsoring Jose M. Alcaraz Calero under their post-doctoral grants. Thanks as well to the Funding Program for Research Groups of Excellence by Fundación Seneca 04552/GERM/06, the Spanish Ministerio de Educación y Ciencia under the research project TIN2008-06441-C02-02, and also thanks for the support of the Spanish MICINN, Consolider Programme and Plan E funds, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04-02/01.

References

[1] Jastor. Typesafe, Ontology Driven RDF Access from Java. http://jastor.sourceforge.net.

[2] The Kazuki Project. http://projects.semwebcentral.org/projects/kazuki.

[3] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding Type Parameterization to the Java Language. *ACM SIGPLAN Notices*, 32:49 – 65, 1997.

[4] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[5] Nick Bassiliades, Efstratios Kontopoulos, and Grigoris Antoniou. A Visual Environment for Developing Defeasible Rule Bases for the Semantic Web. In *Rules and Rule Markup Languages for the Semantic Web*, volume 3791 of *Lecture Notes in Computer Science*, pages 172–186. Springer Berlin / Heidelberg, 2005.

[6] Jesús Bermejo, Frank Golatowski, Francesco Furfari, and Elmar Zeeb. Open Source and Product Lines Architectures. In *11th International Software Product Line Conference (SPLC2007)*, 2007.

[7] Jorge Bernal Bernabe, Juan Manuel Marin Perez, Jose M. Alcaraz Calero, Gregorio Martinez Perez, Felix J. Garcia Clemente, and Antonio F. Gomez Skarmeta. Towards an authorization system for cloud infrastructure providers. In IEEE, editor, *International Conference on Security and Cryptography (SECRYPT)*, 2011.
[8] Gilad Bracha. Generics in the Java Programming Language. Technical report, Sun Microsystems, 2004.
[9] Jose M. Alcaraz Calero, Gregorio Martinez Perez, and Antonio F. Gomez Skarmeta. Towards an authorization model for distributed systems based on the semantic web. *IET Information Security*, 4(4):411–421, 2010.
[10] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: Implementing the Semantic Web Recommendations. In *Proceedings of the 13th international World Wide Web conference*, pages 74–83. ACM Press, 2004.
[11] Taylor G. Cowan. Jenabean: Easily bind JavaBeans to RDF. IBM DeveloperWorks, 2008.
[12] Mike Dean, Dan Connoll, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Web Ontology Language (OWL). Technical report, W3C, 2004.
[13] Jens Dietrich. An Ontological Model for Component Collaboration. *Journal of Research and Practice in Information Technology (JPRIT)*, 43(1):25–39, February 2011.
[14] Joerg Evermann and Yair Wand. Ontology Based Object Oriented Domain Modelling: Fundamental Concepts. In *Requirements Eng*, volume 10, pages 146–160. Springer-Verlag, 2005.
[15] Christoph Frenzel, Bijan Parsia, Ulrike Sattler, and Bernhard Bauer. Mooop – a hybrid integration of owl and java. In Springer, editor, *Advanced Information Systems Engineering Workshops*, volume 83 of *LNCS*, pages 437–447. Springer, 2011.
[16] Ernest Friedman. *Jess in Action: Rule-Based Systems in Java*. Manning Publications Co., Greenwich, CT, USA, 2003.
[17] Iván García-Magariño, Celia Gutiérrez, and Rubén Fuentes-Fernández. The INGENIAS Development Kit: a Practical Application for Crisis-management. In *International Work conference on Artificial Neuronal Networks*, 2009.
[18] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. In *OWLED 2009, 6th OWL Experienced and Directions Workshop*, Chantilly, VA, October 2009.
[19] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RULEML. Technical report, W3C, 2004.
[20] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1:7–26, 2003.
[21] Aditya Kalyanpur, Bijan Parsiaa, Evren Sirina, Bernardo Cuenca Graua, and James Hendlera. Swoop: A Web Ontology Editing Browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):144–153, 2006.
[22] Aditya Kalyanpur, Daniel Pastor, Steve Battle, and Julian Padget. Automatic Mapping of OWL Ontologies into Java. In *Proceedings of Sixteenth International Conference on Software Engineering and Knowledge Engineering*, 2004.
[23] Holger Knublauch. Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protégé/OWL. In *International Workshop on the Model-Driven Semantic*, 2004.
[24] Marta Majewska, Bartosz Kryza, and Jacek Kitowski. Translation of Common Information Model to Web Ontology Language. *LNCS Computational Science ICCS 2007*, 4487:414–417, 2007.
[25] Frank Manola and Eric Miller. RDF Primer. W3C recommendation, W3C, 2004.
[26] Brian McBride. Rdf Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C, 2004.
[27] Boris Motik, Bernardo Cuenca Grau, Ian Horrocks, Zhe Wu, Achille Fokoue, and Carsten Lutz. OWL 2 Web Ontology Language Profiles. W3c recommendation, W3C, http://www.w3.org/TR/owl2-profiles/, 2009.
[28] Boris Motik, Peter F. Patel-Schneider, and Ian Horrocks. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. W3c working draft, W3C, http://www.w3.org/TR/owl2-syntax/, 2009.
[29] Andres Muñoz, Antonia Vera, Juan A. Botía, and Antonio F. Gomez Skarmeta. Defining Basic Behaviours in Ambient Intelligence Environments by means of Rule-Based Programming with Visual Tools. In J. C. Augusto, editor, *1st Workshp of Artificial Intelligence Techniques for Ambient Intelligence. ECAI*, 2006.
[30] N.F. Noy, M. Sintek, S. Decker, M. Crubezy, R.W. Fergerson, and M.A. Musen. Creating Semantic Web contents with Protégé-2000. *IEEE Intelligent Systems*, 16(2):60– 71, 2001.
[31] Eyal Oren and Renaud Delbru. ActiveRDF: Object Oriented Semantic Web Programming. In *Proceeding at 16th International World Wide Web Conference*, pages 817–823, 2007.
[32] Eyal Oren, Benjamin Heitmann, and Stefan Decker. ActiveRDF: Embedding Semantic Web Data into Object-Oriented Languages. *Journal of Web Semantics*, 6:191–202, 2008.

[33] Juan M. Marín Pérez, Jorge Bernal Bernabé, Jose M. Alcaraz Calero, Félix J. García Clemente, Gregorio Martinez Pérez, and Antonio F. Gómez Skarmeta. Semantic-aware authorization architecture for grid security. *Future Generation Computer Systems*, 27(1):40–55, 2011.

[34] Colin Puleston, Bijan Parsia, James Cunningham, and Alan Rector. Integrating Object Oriented and Ontological Representations: A Case Study in Java and OWL. 2008.

[35] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics*, 5(2):10, 2007.

[36] Quatrani Terry. *Visual Modeling with Rational Rose and UML*. Addison-Wesley, 1998.

[37] Max Volkel. RDFReactor: From Ontologies to Programmatic Data Access. In *4th International Semantic Web Conference ISWC*, 2005.

[38] M. Zimmermann. Owl2Java - A Java Code Generator for OWL http://www.incunabulum.de/projects/it/owl2java, 2009.