Hindawi Publishing Corporation
Journal of Engineering
Volume 2013, Article ID 182525, 9 pages
http://dx.doi.org/10.1155/2013/182525

Research Article

Multilanguage Semantic Interoperability in Distributed Applications

Agostino Poggi and Michele Tomaiuolo

Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Parma, Viale U. P. Usberti 181/A, 43100 Parma, Italy

Correspondence should be addressed to Agostino Poggi; agostino.poggi@unipr.it

Received 19 August 2012; Accepted 6 December 2012

Academic Editor: Stavros Koubias

JOSI is a software framework that tries to simplify the development of such kinds of applications both by providing the possibility of working on models for representing such semantic information and by offering some implementations of such models that can be easily used by software developers without any knowledge about semantic models and languages. This software library allows the representation of domain models through Java interfaces and annotations and then to use such a representation for automatically generating an implementation of domain models in different programming languages (currently Java and C++). Moreover, JOSI supports the interoperability with other applications both by automatically mapping the domain model representations into ontologies and by providing an automatic translation of each object obtained from the domain model representations in an OWL string representation.

1. Introduction

Semantic information is assuming more and more importance both for the development of knowledge-based applications and for supporting the interoperability among different applications [1–4]. In particular, ontologies have been gaining interest for the representation of the application domain models and their use has been spreading in different applications fields [5–8].

Domain models are increasingly specified as formal ontologies through the use of a semantic Web language (e.g., OWL [9, 10]), but such models remain difficult to be utilized in applications developed through the used software languages and libraries. In fact, the mapping of such models into the code of a typical application development language often is not possible because of the different expressive power of the modeling and the implementation language. Moreover, when it is possible, the obtained implementation is too complex to be used by the large part of software developers.

However, the development of domain models that represent semantic information is very difficult without the use of a semantic language. To cope with this problem, a possible direction is to integrate usual programming techniques with some meta-programming techniques. In particular, the Java programming language supports meta-programming through annotations and reflection [11]. In fact, while annotations allow the decoration of the Java code with new concepts and idioms, reflection allows the retrieval of the information associated with annotations and then to use them for either modifying the usual execution of the Java code or for building new Java code.

In this paper, we present a software framework, called JOSI (Java and OWL for System Interoperability), whose goal is to simplify the development of the software libraries for managing the data that implement the domain models shared by the systems of a distributed enterprise application. The next section introduces related work on the use of annotations for the development of software and on the mapping between OWL ontologies and Java code. Section 3 describes the JOSI software framework. Section 4 describes how a domain model is represented. Section 5 presents how an implementation of a domain model is built starting from its JOSI representation. Section 6 introduces how domain model implementations are used in a software application. Sections 7 and 8 represent and discuss the experimentation of the JOSI software framework. Finally, Section 9 concludes the paper sketching some future research directions.

2. Related Work

The idea of using Java annotations for extending the Java language is not new and several research teams worked in that direction.

AspectJ [12] is probably the first important work that shows how Java annotation can provide a meta-programming layer on the top of Java programming structures. In particular, AspectJ is an aspect-oriented extension of the Java programming language that uses Java annotations for realizing declaring aspects, point-cuts, and advices.

Andreae et al. [13] proposed a software framework that supports pluggable type systems in the Java programming language by the definition of custom constraints on Java types through Java annotations.

AVal [14] is a software framework for the definition and checking of rules for programs written by using an attribute domain-specific built on the top of Java. This software framework allows the validation of such kinds of program through a set of predefined Java annotations. Moreover, it allows to the users of the framework to add new annotations to provide new kinds of validation.

Bordin and Vardanega [15] used Java annotations to embed in the source code a declarative specification of the required concurrent semantics and then for producing the source code that implements the declared concurrent semantics.

Cimadamore and Viroli [16] proposed a software framework that tried to simplify the seamless integration of Prolog code into Java applications taking advantage of Java annotations to incorporate the declarative features of Prolog into Java programs.

A lot of work has been done also towards the mapping of OWL ontologies into Java code and vice versa.

The first important work that shows the partial translation of OWL ontologies in Java code is the Protégé Bean Generator [17]. In particular, it transforms Protégé frame-based ontologies into Java source code for developing JADE agents [18, 19].

RDFReactor [20] is a toolkit for dynamically accessing an RDF model through domain-centric methods (getters and setters). In particular, it allows the access to the RDF model through a set of proxy objects that provide the methods for querying and updating the RDF elements.

A more sophisticated approach was presented by Kalyanpur et al. [21]. This approach deals with issues as multiple-inheritance by mapping OWL classes in Java interfaces. However, there is not a software tool which takes advantage of this approach for mapping OWL ontologies into Java code.

SeRiDA [22] is a methodology for enabling a three-tier mapping along ontologies, object-oriented java beans and relational database. In particular, it allows the generation of both an object-oriented and a relational model starting from a domain conceptualization expressed in OWL. This methodology has been experimented by realizing a software tool that generates programming interfaces as enterprise Java beans and Hibernate object-relational mappings from OWL ontologies.

Quasthoff and Meinel [23] presented a mechanism that allows application developers, with limited knowledge about RDF and OWL, to easily map arbitrary Java classes and interfaces to corresponding OWL concepts by using Java annotations. In particular, this mechanism has already been experimented in the development of a social network application testing new access control mechanisms on user-generated content with the help of Semantic Web rules [19].

OWLET [24] is a Java software environment based on an object-oriented model, which allows a simple and complete representation of ontologies defined by using OWL DL profile, and provides a complete set of reasoning functions together with a graphical editor for the creation and modification of ontologies. OWLET supports the development of heterogeneous and distributed semantic systems where nodes differ for their capabilities (i.e., CPU power, memory size, etc.). In fact, it offers a layered reasoning API that allows to deploy a system where high power nodes take advantages of all the OWLET reasoning capabilities, medium power nodes take advantages of a limited set of OWLET reasoning capabilities (e.g., reasoning about individuals) and low power nodes delegate reasoning tasks to the other nodes of the system.

Finally, the OWL API can be considered the reference Java API for managing ontologies [25, 26]. In fact, besides providing the manipulation of ontologies, it offers: a general purpose reasoner interfaces, the validators for the various OWL profiles, and the support for parsing and serializing ontologies in a variety of syntaxes. The API also has a very flexible design that allows third parties to provide alternative implementations for all major components.

Different works cope with the problem of defining models for integrating different data sources in enterprise information systems.

Astrova and Kalja [27] proposed an approach for system interoperability that maps relational database schemas into OWL ontologies and allows an improvement of database schemas by identifying "hidden" (implicit) semantic relationships and bad design solutions.

Lin and Harding [28] proposed a general manufacturing system engineering knowledge representation scheme to facilitate communication and information exchange in inter-enterprise, multidisciplinary engineering design teams. It has been developed and encoded in the standard semantic web language. The proposed approach focuses on how to support information autonomy that allows the individual team members to keep their own preferred languages or information models rather than requiring them all to adopt standardized terminology.

Salguero et al. [29] proposed a framework which encompasses the entire data integration process. The data source schemas as well as the integrated schema are expressed using an OWL extension which allows the incorporation of metadata to support the integration process.

3. Software Framework Overview

JOSI (Java and OWL for System Interoperability) is a software framework that tries to simplify the development of the

```
@Immutable
@Comparator ({"name", "domain"})
public interface Address extends Entity {
        @Getter ("name")
        @Cardinality (1)
        String getName ();

        @Getter ("domain")
        @Cardinality (1)
        Domain getDomain ();
}
@Immutable
@Comparator ({"name"})
public interface Domain extends Entity {
        @Getter ("name")
        @Cardinality (value = 1)
        String getName ();
}
@Name ("naming")
@Version ("1.0")
public interface Naming extends Factory {
        @Binding ({"name", "domain"})
        public Address getAddress (final String a, final Domain b);
        @Binding ({"name"})
        public Domain getDomain (final String a);
}
```

Figure 1: A simple naming domain model.

software libraries for managing the data that implement the domain models shared by the systems of a distributed enterprise application.

The main features of such software library are as follows: (i) a strict separation between the representation of a domain model and its implementation, (ii) an automatic generation of an implementation of the representation of the domain model in different programming languages; (iii) an automatic generation of an OWL ontology from the representation of a domain model and vice versa, and (iv) the possibility of using an OWL string representation of the domain model data to support the interoperability between systems implemented in different programming languages and so the possibility of translating domain model data to OWL string representations and vice versa.

JOSI is implemented in Java and takes advantage of Java interfaces and annotations to build a representation of a domain model and uses Java reflection to drive the processing of the information maintained by such interfaces and annotations for generating the source code of the classes that define the concrete implementation of the domain model.

The following sections will describe how a domain model is represented through Java interfaces and annotations, how the Java classes providing a concrete implementation of such a domain model are generated from such interfaces and annotations, and how such a software framework enables an application to use a concrete implementation of a domain model.

Table 1: Java annotations used in the representation of a domain model.

@Abstract	@Getter	@Name	@Symmetric
@AllValuesFrom	@HasValue	@Ordered	@SomeValuesFrom
@Binding	@Immutable	@Set	@Transitive
@Cardinality	@InverseOf	@Setter	@Version

4. Domain Model Representation

A domain model is represented by a set of Java interfaces. Each domain entity is represented by a Java interface (from here called entity interface) that defines the two methods for reading and modifying its attributes. Moreover, an additional Java interface (from here called factory interface) provides both some general information about the domain model and the factory methods for the creation of the Java classes which implement the different entity interfaces. Figures 1 and 2 show some entities of two domain models represented through the use of Java interfaces and annotations. Table 1 lists the Java annotations used in the representation of a domain model.

To support the creation of the implementation of such entities, each Java interface is enriched by some Java annotations and constant declarations.

The two annotations: *@Getter* and *@Setter* are applicable to the entity interface methods and define the reading and modifying methods of a specific attribute. The type of the

```
@Abstract
public interface Action extends Entity {
        @Getter ("resultTypes")
        @Cardinality (min = 1)
        String [] getResultTypes ();
        @Getter ("errorValues")
        Error [] getErrorValues ();
}
@Immutable
@Comparator ({"name"})
public interface Describe extends Action {
        public static final String DESCRIPTION = Description.class.getName ();
        public static final Error UNKNOWNACTION =
            ((Interaction) DataStore.getModel ("interaction")).getUnknownAction();
        public static final Error UNREACHABLEAGENT =
            ((Interaction) DataStore.getModel ("interaction")).getUnreachableAgent();
        @Getter ("name")
        @Cardinality (1)
        String getName ();
        @Override
        @Getter ("resultTypes")
        @HasValue (values={"DESCRIPTION"})
        @Cardinality (1)
        String [] getResultTypes ();
        @Override
        @Getter ("errorTypes")
        @HasValue (values = {"UNKNOWNACTION", "UNREACHABLEAGENT"})
        @Cardinality (2)
        Error [] getErrorValues ();
}
```

Figure 2: Two entities of a domain model describing the life-cycle of a software agent.

attribute is identified by both the return type of the reading method and the type of the argument of the modifying method (of course they need to identify the same type). In particular, the value of any attribute must be: a Java primitive data, an instance of the String class, an instance of a class implementing an entity interface, or an array of the previous kinds of value.

The four annotations: *@Abstract*, *@Immutable*, *@OneOf,* and *@Singleton,* are applicable to the entity interfaces. The first annotation identifies an abstract entity, that is, an entity that does not have any direct implementation. The second annotation identifies an entity that has an immutable implementation, that is, the interface cannot define methods that modify the value of its attributes and the implementation of its reading methods will be defined to return either the value of an attribute (if it is an immutable value) or a copy of the value (if it is a mutable value). The third annotation is used for identifying entities that have an extensional description (e.g., that can be defined through an enumeration). Finally, the forth annotation is used for the definition of some special entities that can be represented by a single class object.

Often the use of an implementation of a domain model inside an application needs the availability of operations for the comparison and ordering of their entities. In a Java implementation, such operations can be performed by implementing the *compareTo*, equals, and *hashCode* methods. The annotation *@Comparator* is introduced for this scope. In fact, it identifies the sequence of attributes on which the previous three methods must work.

In a domain model often is necessary both to restrict the value that some attributes can assume and to establish a relationship between the attributes of some entities. It is done by associating some additional annotations to the reading methods of the entity interfaces.

The four annotations: *@AllValueFrom*, *@SomeValuesFrom*, *@Cardinality,* and *@HasValue,* define the most known constraints that OWL applies to the properties of an ontology. In particular, the first annotation constrains the values of an attribute to belong to specific type (of course, an implicit constraint of such a kind, is defined when the reading and modifying methods of an attributed are defined. However, an additional constraint can be added by imposing that the values of an attribute must belong to a subtype of the declared attribute type). The second annotation imposes that some of the values of an attribute must belong to a specific type (of course, such a type must be a subtype of the declared attribute type). The third annotation imposes that an attribute can have either a fixed number of values or a variable number of values defined by a minimum and/or a maximum value. Finally, the forth annotation imposes that an attribute must always contain some values (in this case, for the limited set of value types that can be associated with the attributes of an

```
public final class E1 implements Address {
    private String name;
    private Domain domain;
    // Class constructor.
    public E1(final String a, final Domain b) {
        this.name = a;
        this.domain = b;
    }
    // Checks instance consistency.
    public boolean check () {
        if ((this.name != null) && (this.domain != null)) return true;
        return false;
    }
    @Override
    public String getName () {
        return this.name;
    }
    @Override
    public Domain getDomain () {
        return this.domain;
    }
}
public final class E2 implements Domain {
    private String name;
    // Class constructor.
    public E2 (final String a) {
        this.name = a;
    }
    // Checks instance consistency.
    public boolean check () {
        if (this.name != null) return true;
        return false;
    }
    @Override
    public String getName () {
    return this.name;
    }
}
```

Figure 3: Java implementation of the entities of the naming domain model.

annotation, the values of such constraints are defined through constant variables and the annotations refer to the names of such constant variables).

In some cases it can be necessary to impose that an attribute does not have duplicated values and that its values are maintained ordered: the two annotations: *@Set*, and *@Ordered,* impose the previous two constraints (in particular, the second constraint is implemented either by using the natural ordering between values or the ordering defined by the *compareTo* method built through the *@Comparator* annotation introduced above).

The three annotations: *@InverseOf*, *@Symmetric*, and *@Transitive,* define the most known constraints that OWL applies to the relationship between properties of an ontology. The first annotation defines an inverse relationship between attributes. The second annotation defines a symmetric relationship between the entities that have such kind of attribute. Finally, the third annotation defines a transitive relationship between the entities that have such kind of attribute.

Finally, the two annotations: *@Name* and *@Version,* are applicable to the factory interfaces: the first annotation indicates the name associated with the domain model and the second annotation identifies the version of the model. Lastly, the annotation *@Binding* is associated with a factory method of a model interface. This annotation identifies the attribute that each argument of the factory method will initialize.

5. Domain Model Implementation

A domain model representation, defined as described in the previous sections, contains all the information for building an implementation of such a domain model. This implementation is realized by an annotation processor that builds a Java class for each Java interface of the model. Figures 3 and 4 show the source code of the Java classes obtained through the naming domain model introduced in the previous section.

```
public final class F1 implements Naming {
    private static final String NAME = "naming";
    private static final String VERSION = "1.0";
    private static final String [] ENTITIES = {
        Address.class.getName (), Domain.class.getName ()};
    @Override
    public String getModelName () {
        return NAME;
    }
    @Override
    public String getModelVersion () {
        return VERSION;
    }
    @Override
    public String [] list () {
        return ENTITIES;
    }
    @Override
    public E1 getAddress (final String a, final Domain b) {
        E1 i = new E1 (a, b);
        if (i.check ()) return i;
        return null;
    }
    @Override
    public E2 getDomain (final String a) {
        E2 i = new E2 (a);
        if (i.check ()) return i;
        return null;
    }
}
```

Figure 4: Java implementation of the model of the naming domain model.

The result of such an annotation processor is a set of Java files. Each Java file contains the source code of a class that implement an interface of the domain model representation. Moreover, each class that implements an entity interface provides a method for building an OWL string representation of an entity class instance, and each class that implements a model interface provides a method for building an entity class instance from its OWL string representation.

The annotation processor used for generating the domain model implementation is composed by two software modules. The first module, called processing module, extracts the information from the domain model representation, generates an intermediate representation and then calls the second module. Then the second module, called generation module, builds the domain model implementation from the intermediate representation.

The intermediate representation is based on a two level tree where the root object maintains the information about the model interface and each leaf object maintains the information about an entity interface.

The processing module is independent from the implementation of the generation module because it calls a generation module by a Java interface and the generation module implementation is a parameter of the processing module constructor.

Therefore, it is very easy to provide different implementations of some domain model representations by defining new generation modules able to process in different ways the intermediate representation built by the processing module. In particular, the current version of the software framework provides another generation module which builds OWL ontologies from the domain model representations and stores them in RDF format [30].

6. Domain Model Application

After the creation of an implementation of a domain model, its use inside an application is very simple. In fact, the JOSI software framework provides a class, called *DataStore*, which has the duty of both maintaining the information about the different domain models available for the current application and providing the access to their implementation through the creation of an instance of the class that implements their domain interface. In particular, the *Datastore* instance can access to the list of the domain models used by the application through a property file.

Therefore, after the creation of an instance of the *DataStore* class, the code of the application can create instances of any class implementing the factory interface of a domain model and then use it for creating instances implementing

```
DataStore dt = Datastore.getInstance ();
Namning n = (Naming) dt.getFactory ("naming");
Domain d = n.getDomain ("localhost");
Address a1 = n.getAddress ("agent1", d);
Address a2 = n.getAddress ("agent2", d);
Lifecycle lc = (Lifecycle) dt.getFactory ("lifecycle");
Describe d1 = lc.getDescribe ("agent1");
```

Figure 5: Java code for creating instances of the entities of two domain models.

any entity interface of such a domain model. Figure 5 shows a sample of Java code performing the operations described above.

7. Experimentation

We are using the JOSI software framework for the development of the models and then the implementations of the data necessary for supporting the basic interactions among the components of a distributed system realized through the HDS software framework. Moreover, JOSI was experimented for defining the domain models of some applications in the fields of distributed information sharing and social networks.

HDS (Heterogeneous Distributed System) is a software framework that tries to simplify the realization of pervasive applications by merging the client-server and the peer-to-peer paradigms and by implementing all the interactions among the processes of a system through the exchange of typed messages and the use of composition filters for driving and dynamically adapting the behavior of the system [31].

Typed messages are one of the elements that mainly characterize such a software framework. In fact, typed messages can be considered an object-oriented "implementation" of the types of message defined by an agent communication language and so they are means that make HDS a suitable software framework both for the realization of multiagent systems and for the reuse of multiagent model and techniques in nonagent based systems.

In particular, the type of a message is defined by its content and its content is defined by an entity of a specific domain model defined with the JOSI software framework. Therefore, we used JOSI foe the definition of the domain models that support the basic interaction among HDS processes, that is, the managing of the processes themselves and of the resources that can they used in a distributed application. Moreover, we used JOSI for defining the domain models used for realizing the typical coordination algorithms of intelligent distributed systems.

RAIS (Remote Assistant for Information Sharing) is a peer-to-peer multiagent system supporting the sharing of information among a community of users connected through the Internet [32]. RAIS offers search facility similar to Web search engines, but it avoids the burden of publishing the information on the Web and it guaranties a controlled and dynamic access to information through the use of agents.

The use of agents in such a system is very important because it simplifies the realization of the three main services: (i) the filtering of the information coming from different users on the basis of the previous experience of the local user; (ii) the pushing of the new information that can be of possible interest for a user; and (iii) the delegation of access capabilities on the basis of a network of reputation built by the agents on the community of users.

RAIS is composed of a dynamic set of agent platforms connected through the Internet. In this case, JOSI has been used for the definition of the domain models supporting the definition of the interaction of agents for the retrieval and pushing of the information and for the management of the user profiles.

About the applications in the field of the social networks, we are starting the development a system for the study of the most known social networks and, in particular, of the social networks that provide semantic support for the management of both the profiles and the information published by the users [33].

In particular, we built a system that can simulate the behavior of some of the most known social networks and can compare them with some enhanced versions of such networks that provide semantic support through the use of JOSI domain models. In particular, we defined some domain models for representing the user profiles of different social networks and some domain models for supporting users in the publishing and retrieval of information related to some sample topics (e.g., computer science and music).

8. Experimental Results

The results of the experimentation of the software framework showed that the definition of a domain model can be done by any programmer with knowledge about the Java programming language, but does not require any knowledge about any knowledge engineering and semantic Web techniques and technologies. Moreover, if the entities of a domain model are defined as immutable objects, then the performance of managing such entities is similar to the one of managing JavaBean objects.

Other important results come from some tests that compared the result of the work of groups of students, which developed domain models using JOSI, with the work of other groups of students, which developed domain models without using it. In fact, while the first set of

groups developed the domain model in few time spending a very limited part of it for code correction, the second set of groups developed the domain model in a very long time spending its large part for code correction. Moreover, the performance measures of the tests showed that the implementations of the domain model based on the JOSI framework provided better measures or at least similar to the ones provided by the "custom" implementations. Of course, while the use of JOSI guaranteed implementations in different programming languages (currently Java and C++) without additional costs, it was not true for "custom" implementations.

9. Conclusion

This paper presented a software framework, called JOSI (Java and OWL for System Interoperability), that has the goal of simplifying the development of the software libraries for managing the data that implement the domain models shared by the systems of a distributed enterprise application.

This software framework allows to represent a domain model through Java interfaces and annotations and then to use such a representation for automatically generating a Java implementation of the domain model. Moreover, it provides the interoperability with other kinds of systems both automatically mapping the Java domain representation in an OWL ontology and providing an automatic translation of each object defined by the domain model representation in an OWL string representation.

JOSI derived from O3L (Object-Oriented Ontology Library), a software library that provides a complete representation of ontologies compliant with OWL 2 W3C [34]. O3L has not the goal to be used for the creation and manipulation of ontologies, but provides a simplified and efficient API for the realization of applications, that interoperate through the use of shared ontologies, and allows: (i) the use of OWL individuals as data of the applications, (ii) the exchange of OWL individuals between applications, (iii) the reasoning about OWL individuals, and (iv) the classification of OWL classes and properties. The experimentation of O3L showed that it is a powerful means for developing applications but with two main limits: developers must have a good knowledge of semantic techniques and technologies and often applications cannot provide the required performances.

Current and future research activities are dedicated, besides to continue the experimentation of the current implementation of JOSI, to: (i) the development of a software generation module that allows the automatic generation of a C++ and Python implementation from a JOSI model representation, (ii) the generation of a JOSI model representation from an OWL ontology compliant with the JOSI domain model representation, (iii) the generation of OWL ontologies compliant with such a representation from OWL ontologies that contain classes and properties that cannot be defined through the annotations defined in the JOSI software framework, (iv) the introduction of new annotations for increasing the expressive power of the JOSI model representation.

References

[1] P. A. Bernstein and L. M. Haas, "Information integration in the enterprise," *Communications of the ACM*, vol. 51, no. 9, pp. 72–79, 2008.

[2] M. Ciocoiu, D. S. Nau, and M. Gruninger, "Ontologies for integrating engineering applications," *Journal of Computing and Information Science in Engineering*, vol. 1, no. 1, pp. 12–22, 2001.

[3] R. García-Castro and A. Gómez-Pérez, "Interoperability results for semantic web technologies using OWL as the interchange language," *Web Semantics*, vol. 8, no. 4, pp. 278–291, 2010.

[4] S. Heiler, "Semantic interoperability," *ACM Computing Surveys*, vol. 27, no. 2, pp. 271–273, 1995.

[5] D. Oberle, S. Staab, R. Studer, and R. Volz, "Supporting application development in the semantic web," *ACM Transactions on Internet Technology*, vol. 5, no. 2, pp. 328–358, 2005.

[6] M. Quasthoff, H. Sack, and C. Meinel, "Who reads and writes the social web? A security architecture for Web 2.0 applications," in *Proceedings of the 3rd International Conference on Internet and Web Applications and Services (ICIW '08)*, pp. 576–582, Athens, Greece, June 2008.

[7] M. Uschold, "Ontology-driven information systems: past, present and future," in *Proceedings of the 5th International Conference on Formal Ontology in Information Systems (FOIS '08)*, pp. 3–18, Amsterdam, The Netherlands, 2008.

[8] N. F. Noy, "Semantic integration: a survey of ontology-based approaches," *SIGMOD Record*, vol. 33, no. 4, pp. 65–70, 2004.

[9] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. Patel-Schneider, and U. Sattler, "OWL 2: the next step for OWL," *Web Semantics*, vol. 6, no. 4, pp. 309–322, 2008.

[10] D. L. McGuinness and F. van Harmelen, OWL web ontology language overview, W3C Recommendation, 2004, http://www.w3.org/TR/owl-features/.

[11] B. Joy, J. Gosling, G. Steele, and G. Bracha, *The Java Language Specification*, Addison-Wesley, New York, NY, USA, 3rd edition, 2005.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, vol. 2072 of *Lecture Notes in Computer Science*, pp. 327–354, Springer, Berlin, Germany, 2001.

[13] C. Andreae, J. Noble, S. Markstrum, and T. Millstein, "A framework for implementing pluggable type systems," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, pp. 57–74, New York, NY, USA, October 2006.

[14] C. Noguera and R. Pawlak, "AVal: an extensible attribute-oriented programming validator for Java," *Journal of Software Maintenance and Evolution*, vol. 19, no. 4, pp. 253–275, 2007.

[15] M. Bordin and T. Vardanega, "Real-time Java from an automated code generation perspective," in *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES '07)*, pp. 63–72, New York, NY, USA, September 2007.

[16] M. Cimadamore and M. Viroli, "A Prolog-oriented extension of Java programming based on generics and annotations," in *Proceedings of the 5th International Symposium on the Principles and Practice of Programming in Java (PPPJ '07)*, pp. 197–202, New York, NY, USA, September 2007.

[17] C. van Aart, R. Pels, G. Caire, and F. Bergenti, "Creating and using ontologies in agent communication," in *Proceedings of the*

Workshop on Ontologies in Agent Systems, Bologna, Italy, July 2002.

[18] F. Bellifemine, A. Poggi, and G. Rimassa, "Developing multi agent systems with a FIPA-compliant agent framework," *Software Practice & Experience*, vol. 31, no. 2, pp. 103–128, 2001.

[19] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa, "JADE: a software framework for developing multi-agent applications. Lessons learned," *Information and Software Technology*, vol. 50, no. 1-2, pp. 10–21, 2008.

[20] M. Volkel, "RDFReactor: from ontologies to programmatic data access," in *Proceedings of the Jena User Conference*, Bristol, UK, 2006.

[21] A. Kalyanpur, D. J. Pastor, S. Battle, and J. Padget, "Automatic mapping of OWL ontologies into Java," in *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering*, pp. 98–103, Banff, Canada, 2004.

[22] I. N. Athanasiadis, F. Villa, and A. Rizzoli, "Enabling knowledge-based software engineering through semantic-object-relational mappings," in *Proceedings of the 3rd International Workshop on Semantic Web-Enabled Software Engineering, 4th European Semantic Web Conference*, pp. 16–30, Innsbruck, Austria, 2007.

[23] M. Quasthoff and C. Meinel, "Semantic web admission free—obtaining RDF and OWL data from application source code," in *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering*, pp. 17–25, Karlsruhe, Germany, 2008.

[24] A. Poggi, "OWLET: an object-oriented environment for OWL ontology," in *Proceedings of the 11th WSEAS International Conference on Computers (ICCOMP '07)*, pp. 44–49, Stevens Point, Wis, USA, 2007.

[25] S. Bechhofer, R. Volz, and P. Lord, "Cooking the semantic web with the OWL API," in *Proceedings of the 2nd International Semantic Web Conference (ISWC '03)*, D. Fensel, K. Sycara, and J. Mylopoulos, Eds., vol. 2870 of *Lecture Notes in Computer Science*, pp. 659–675, Springer, Berlin, Germany, 2003.

[26] M. Horridge and S. Bechhofer, "The OWL API: a Java API for working with OWL 2 ontologies," in *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED '09)*, Chantilly, Va, USA, 2009.

[27] I. Astrova and A. Kalja, "Mapping of SQL relational schemata to OWL ontologies," in *Proceedings of the 6th WSEAS International Conference on Applied Informatics and Communications (AIC '06)*, pp. 376–380, Stevens Point, Wis, USA, August 2008.

[28] H. K. Lin and J. A. Harding, "A manufacturing system engineering ontology model on the semantic web for inter-enterprise collaboration," *Computers in Industry*, vol. 58, no. 5, pp. 428–437, 2007.

[29] A. Salguero, F. Araque, and C. Delgado, "Ontology based framework for data integration," *WSEAS Transactions on Information Science and Applications*, vol. 5, no. 6, pp. 953–962, 2008.

[30] J. Z. Pan, "Resource description framework," in *International Handbook on Ontologies*, S. Staab and R. Studer, Eds., Handbooks on Information Systems, Part 1, pp. 71–90, Springer, Berlin, Germany, 2009.

[31] A. Poggi, "HDS: a software framework for the realization of pervasive applications," *WSEAS Transactions on Computers*, vol. 9, no. 10, pp. 1149–1159, 2010.

[32] F. Bergenti and A. Poggi, "Building distributed and pervasive information management systems with HDS," in *Advances in Distributed Agent-Based Retrieval Tools*, V. Pallotta, A. Soro, and E. Vargiu, Eds., vol. 361 of *Studies in Computational Intelligence*, pp. 129–142, Springer, Berlin, Germany, 2011.

[33] F. Bergenti, E. Franchi, and A. Poggi, "Selected models for agent-based simulation of social networks," in *Proceedings of the Social Networks and MultiAgent Systems Symposium (SNAMAS '11)*, pp. 27–32, York, UK, 2011.

[34] A. Poggi, "Developing ontology based applications with O3L," *WSEAS Transactions on Computers*, vol. 8, no. 8, pp. 1286–1295, 2009.