Sapphire: Generating Java Runtime Artefacts from OWL Ontologies

Graeme Stevenson and Simon Dobson

School of Computer Science,
University of St Andrews, UK
gs@cs.st-andrews.ac.uk

Abstract. The OWL ontology language is proving increasingly popular as a means of crafting formal, semantically-rich, models of information systems. One application of such models is the direct translation of a conceptual model to a set of executable artefacts. Current tool support for such translations lacks maturity and exhibits several limitations including a lack of support for reification, the open-world assumption, and dynamic classification of individuals as supported by OWL semantics. Building upon the state-of-the-art we present a mapping from OWL to Java that addresses these limitations, and its realisation in the form of a tool, *Sapphire*. We describe Sapphire's design and present a preliminary evaluation that illustrates how Sapphire supports the developer in writing concise, type safe code compared to standard approaches while maintaining competitive runtime performance with standard APIs.

Key words: Information systems, OWL, Java, Ontology, Bytecode generation, Data binding

1 Introduction

A goal of ontology-driven information systems engineering is the construction of real-world domain models with sufficient formality, expressiveness, and semantic-richness to allow information system software components to be automatically generated from the underlying ontological description [1].

Several commentators advocate mapping from ontologies to strongly-typed programming languages in order to eliminate a large class of typographic errors at compile time that might only be discovered at runtime or go undetected if a dynamic approach were taken [2, 3]. Taking OWL as a specific example of an ontology language, tools that provide a mapping to strongly-typed languages exist [4, 5], but exhibit limitations such as a lack of support for reification, the open-world assumption, and the dynamic classification of individuals as supported by OWL semantics. In this paper we address these limitations:

– We propose a novel mapping from OWL to Java that accounts for fundamental differences between ontological and object-oriented modelling. The mapping is realised by part of the *Sapphire* tool, which generates bytecode for a set of Java interfaces corresponding to a set of OWL ontologies (Section 2).

– We describe the design of runtime artefacts called *Sapphires* that conform to an OWL individual's type set and map method invocations to an underlying quad store. Type manipulation operations approximate the dynamic classification of OWL individuals within the statically-typed Java environment (Section 3).
– We present an initial evaluation that shows how Sapphire helps developers to write concise, type safe code while maintaining runtime performance competitive with standard APIs. We also show that our mapping approach improves upon state-of-the-art tool support for OWL's feature set (Section 4).

We conclude with an overview of related work (Section 5) and planned future work (Section 6).

2 OWL to Java: A Mapping Approach

Although ontological and object oriented models exhibit similarities that lend themselves to a mapping, the fit is not exact [2]. Here we explore some of the fundamental challenges that a mapping between OWL and Java presents before discussing our solution.

– Mapping OWL Classes to Java interfaces is an established technique for approximating OWL's multiple inheritance [4, 5]. However, in generating implementation classes, schemes described in the literature provide no support for dynamic classification of OWL individuals.
– OWL properties are first class objects with rich descriptions. This includes domain restrictions identifying which classes may legally use them, range restrictions on the data types and data ranges they may take (e.g., the set of integers between 1 and 31), and relations with other properties (e.g., sub property, transitive, inverse). Java models do not capture such semantics naturally.
– OWL models make the open-world assumption, which states that the truth-value of a statement is independent of whether or not it is known by any single observer to be true. This is at odds with Java's boolean logic.
– In addition to direct naming, OWL classes may be defined as the union (member of any), intersection (member of all), or complement (not member) of other classes. They may also be defined as equivalent to (shared members) or disjoint with (no shared members) other classes, or as an enumerated set of individuals.

2.1 Realising the Mapping

To begin the mapping process we use the OWL API [6] to load a set of ontologies and create a representation of a Java package for each ontology namespace. Within each package we follow the established strategy of mapping OWL classes to Java interfaces. However, we take a novel approach of providing implementation logic via dynamic proxies that conform to an individual's class membership. This allows us to approximate the dynamic classification of objects (see Section 3.1). The root interface of the hierarchy, `Thing`, is constructed and added to the package corresponding to the OWL

namespace, `org.w3.www._2002._07.owl`. Thereafter, each class in the ontology is mapped into the hierarchy according to its sub- and super-class relations.

The final manipulation of the interface hierarchy accommodates OWL union and intersection constructs, which map cleanly to Java subclass semantics [4]. The representations for equivalent, complement, and enumerated OWL classes do not affect the structure of the class hierarchy but are handled at runtime as we will later discuss.

In the next step, a representation for each property defined in the ontology set is constructed and mapped to the interface that corresponds to its domain, or `Thing` if its domain is undefined. Cardinality and range value restrictions are extracted from the ontology model. Object property ranges are mapped to the corresponding interface representation in the constructed hierarchy, while for datatype properties a registry provides mappings to Java classes for the range of OWL supported literal datatypes (Literal, XMLLiteral, XML Schema etc.).

Each datatype has a corresponding `DatatypeMapper` that marshals data between its OWL and Java forms, and by implementing and registering a `DatatypeMapper`, the datatype registry is straightforwardly extended. Additionally, developers may also define mappings between OWL classes (i.e., non-literals) and Java objects. This feature is useful in cases where complex ontological descriptions (perhaps spanning multiple individuals) are logically encapsulated by a single Java object. For example, a mapping between concepts in the OWL-Time ontology [7] and the Java Date and Time API. Where such a mapping is defined, the OWL class is omitted from the generated interface hierarchy.

Sapphire adopts closed-world logic by default, but provides an option whereby boolean return values are replaced by a three valued enumerated type to support open-world logic (`LogicValue` {`TRUE`, `FALSE`, `UNKNOWN`}). An `openEquals()` method that forms part of the `Thing` interface allows for the correct interpretation of `owl:sameAs`, `owl:differentFrom`, and `owl:AllDifferent` statements.

Once complete, the internal representation of the annotated Java interface hierarchy is realised using the ASM bytecode manipulation library [8]. Within each generated interface, the parameters and return values of generated *getter* and *setter* methods are typed according to property range and cardinality restrictions. Where appropriate, *add*, *remove*, *removeAll*, *contains*, *has*, *count*, and *reify* methods provide convenient data access. Generated interfaces are annotated with descriptions of their corresponding OWL class; this includes their URI, type, equivalent-, disjoint- and complement-classes, and instance URIs for enumerated types. Methods are similarly annotated with descriptions of their corresponding OWL property; this includes their URI, type, and cardinality and data rage information, which is used for runtime validation.

3 The Sapphire Runtime Library

Dynamically generated software artefacts called *Sapphires* provide the implementation logic for the generated interfaces. Factory methods: `get()`, `create()`, and `getMembers()`, illustrated in Listing 1, provide access individuals in the model. On creation, the Java Dynamic Proxy API is used to construct a Sapphire whose interfaces

Listing 1: Sapphire creation and access via factory methods.

```
// Fetching and Creating individuals
Person bob = Factory.get("urn:bob", Person.class);
Course course = Factory.create("urn:ont101", Course.class);
// To fetch all members of a class.
Collection<Book> books = Factory.getMembers(Book.class);
```

conform to the set of OWL classes to which an individual belongs. A cache maintains a list of those recently accessed.

Figure 1 depicts the internal structure of a Sapphire, with the interfaces it exposes shown on the left hand side of the diagram. Listing 2 illustrates Sapphire's JavaBean style interaction model. When a method is invoked, the `SapphireProxy` dispatches the call to the appropriate handling method, such as those in the `SapphireHandler` and `TypeMechanics` classes shown.

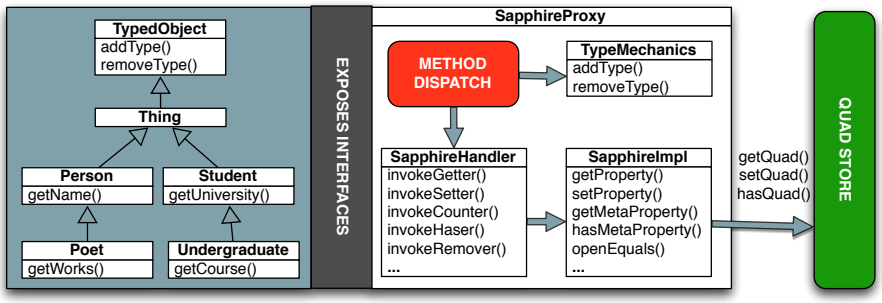

Fig. 1: An illustration of the internal structure of a Sapphire.

The `SapphireHandler` class inspects interface annotations to check for potential cardinality or data range violations that operations may cause. If these checks pass, corresponding calls to the `SapphireImpl` class are made, which in turn map calls to primitive operations on the quad store[1]. As described in Section 2, data access and modification operations use a `DatatypeMapper` to marshall data between its OWL and Java forms.

Rather than translate OWL entailment rules to production code, we apply a reasoner to the underlying quad store. While this impacts performance, a hybrid approach yields several benefits: classifications that do not cleanly map to Java are handled by the reasoner, applications can be integrated seamlessly with legacy code operating over the

[1] A standard triple store could be used in place of the quad store.

Listing 2: Sapphires support JavaBean style method invocation.

```
// Set Bob's name.
bob.setName("Bob Jones");
// Get Bob's date of birth
DateTime age = bob.getDateOfBirth();
// Get Bob's publications.
Collection<Publication> publications = bob.getPublications();
// Use reification to mark assertion time.
bob.reifyTeachingAssistantOf(course, Temporal.class).
    setAssertedAt(new DateTime());
// Clear Bob's teaching responsibilities.
bob.removeAllTeachingAssistantOf();
```

same data, and certain types of schema evolution are supported without requiring that interfaces be regenerated.

3.1 Type Mechanics

We approximate the OWL type system, supporting the addition and removal of types to and from an object, through the `addType()` and `removeType()` methods of the `TypedObject` interface, which map to the `TypeManipulator` component of the `SapphireProxy`.

When adding a type, the new type set supported by the object is calculated and used to generate a new proxy object that rewraps the internal components of the old proxy, updating the quad store in the background. The type removal process does likewise.

Casting of an object to its equivalent classes is supported by inspecting annotations describing class equivalence during the proxy construction process[2]. Similarly, the annotations describing disjoint classes are used to prevent violations in the type addition and removal processes.

The processes of adding and removing types may result in the implicit addition or removal of union, intersection, or complement class to an object's type set. This is handled automatically when the object's updated type set is calculated. Listing 3 illustrates this using an instance of the `Cat` class as defined by the Mindswap Pet Ontology [9]. To this instance, the `Pet` type is added, after which the object may be cast to the `PetCat` class — the intersection of the two. Removing the `Pet` type from the instance removes the intersection class from its type set.

There are several advantages of our approach to approximating OWL type semantics. In particular, we need not generate intersection classes to accommodate possible class combinations (which in the worst case would require the powerset $\mathbb{P}(n)$ generated classes for n classes in an ontology); Sapphires may assume multiple, unrelated types; and we preserve the semantics of the Java *equals*, *instanceof*, and *casting* operators.

[2] This results in a cleaner API than those produced by tools that create a composite interface for all equivalent classes.

Listing 3: Adding types to and removing types from a Sapphire

```
// Create a Cat and make it a Pet.
Cat tom = Factory.create("urn:ex:tom", Cat.class);
Pet tomAsPet = tom.addType(Pet.class);
// We may now cast between Cat, Pet, and PetCat classes.
PetCat tomAsPetCat = (PetCat) tomAsPet;
// Remove the Pet type from Tom (identifying return type).
tom = tomAsPetCat.removeType(Pet.class, Cat.class)
// New proxy interface prohibits access to old types, e.g.,
tomAsPetCat = (PetCat) tom; // ClassCastException thrown.
```

One disadvantage of this approach is that old references to modified objects become out of date. As a result, problems might arise if a method is invoked on a type that an object no longer carries. To mitigate this, method calls are checked against a Sapphire's *graveyard type* set, to prevent such modifications.

4 Evaluation

We have undertaken a preliminary qualitative and quantitative evaluation of Sapphire. First, we illustrate the core benefits of Sapphire — brevity and safety — through a typical query example. We then compare Sapphire's capabilities to similar tools, and examine its impact on performance by comparing Sapphire with standard Java-based querying mechanisms.

This evaluation uses the Leigh University Benchmark (LUBM) [10], an ontology and data generator based on a university domain ontology.

4.1 Code Size and Error Prevention

Based on the example presented by Goldman [2], we construct a query using the LUBM ontology to find and print the name of the person whose email address is *FullProfessor7@Department0.University0.edu*. Listing 4 shows how this query is expressed using Sapphire, while Listing 5 expresses the same query using the Jena ontology API [11].

Listing 4: Sapphire query for a person with a given email address's name.

```
Collection<Person> people = a_factory.getMembers(Person.class
    );
for (Person person : people) {
   for (String emailAddress : person.getEmailAddress()) {
      if ("FullProfessor7@Department0.University0.edu".equals
          (emailAddress)) {
         System.out.printf("Name: %s", person.getName());
  } } }
```

Listing 5: Jena query for a person with a given email address's name.

```
static final String U_URI = "http://www.lehigh.edu/~zhp2
    /2004/0401/univ-bench.owl#";
Literal emailAddress = ResourceFactory.createPlainLiteral("
    FullProfessor7@Department0.University0.edu");
OntClass personClass = model.getOntClass(U_URI + "Person");
OntProperty nameProperty = model.getOntProperty(U_URI + "name
    ");
OntProperty emailProperty = model.getOntProperty(U_URI + "
    emailAddress");
ExtendedIterator<Individual> instances = model.
    listIndividuals(personClass);
while (instances.hasNext()) {
   Individual resource = instances.next();
   if (resource.hasProperty(emailProperty, emailAddress)) {
      RDFNode nameValue = resource.getPropertyValue(
          nameProperty);
      Literal nameLiteral = (Literal) nameValue.as(Literal.
          class);
      String name = nameLiteral.getLexicalForm();
      System.out.printf("Name: %s", name);
 } }
```

It can be seen that the Sapphire representation of the query is as least half the size of the Jena representation. Representations of the same query using NG4J [12] and the Jena SPARQL API are similarly verbose. While the primary contributor to this brevity is the use of domain-typed objects and properties, this example illustrates how Sapphire prevents typographical errors affecting class and property names and does not require developers to cast values obtained from the model to an appropriate Java type.

4.2 Feature Comparison Matrix

Figure 2 provides a feature by feature comparison of Sapphire against four similar tools: RDFReactor [13], OntoJava [14], the work of Kalayanpur et al. [4], and Owl2Java [5], against the core features of the OWL specification. Two of these tools target RDF Schema and comparisons have therefore been drawn where appropriate.

Of the projects surveyed, Sapphire is the most feature complete. In particular, Sapphire is the only tool to support reification, open-world modelling, extension of OWL's supported datatypes, and dynamic re-classification of individuals. The hybrid schemes employed by both Owl2Java and Sapphire take advantage of reasoners to handle tasks that would otherwise be less straightforward for a complete Java conversion (e.g., computing the membership of complement classes). OWL versioning properties have no equivalent Java feature.

	Sapphire	RDFReactor	OntoJava	Kalyanpur et al.	OWL2Java
RDF/OWL	OWL	RDF	RDF	OWL	OWL
Approach	Hybrid	Hybrid	Standalone	Standalone	Hybrid
Mapping Technique	Interfaces + DynamicProxy	Implementation Classes	Interfaces + Implementation Classes	Interfaces + Implementation Classes	Interfaces + Implementation Classes
Preserves class hierarchy	Yes	No	Yes	Yes	Yes
Open-world assumption	Yes	No	No	No	No
Reification	Yes	No	No	No	No
Named Classes	Yes	Yes	Yes	Yes	Yes
Enumerated Classes	Yes	N/A	N/A	Yes	No
Property Restriction Classes	Yes	N/A	N/A	No	Yes
Intersection Classes	Yes	N/A	N/A	Yes	Yes
Union Classes	Yes	N/A	N/A	Yes	Yes
Complement Classes	Yes	N/A	N/A	No	Yes
Equivalent Classes	Yes	N/A	N/A	Yes	Yes
Disjoint Classes	Yes	N/A	N/A	Yes	Yes
Property Cardinality	Yes	Yes (OWL)	N/A	Yes	Yes
Intra-Property Relations (sub, super, inverse, equiv.)	Yes	Yes	Yes	Yes	Yes
Property Logic Characteristics (functional, transitive, etc.)	Yes	N/A	N/A	Yes	Yes
Individual Identity Relationships	Yes	N/A	N/A	No	No
Dynamic Individual Classification	Yes	No	No	No	No
Full Datatype Support	Yes	No	No	No	Yes
Extensible Datatype Support	Yes	No	No	No	No
Enumerated Data Ranges	Yes	N/A	N/A	Yes	No
RDF Container/Collections	Yes	List only	No	No	No
(Annotation) Imports	Yes	N/A	N/A	Yes	Yes
(Annotation) Versioning/ Compatibility	No	N/A	N/A	No	No
(Annotation) Deprecated Properties and Classes	Yes	N/A	N/A	No	No

Fig. 2: A comparison of support for features of the OWL specification within Sapphire and similar tools.

4.3 Initial Performance Analysis

To evaluate Sapphire's overhead, we compare its performance with three alternative Java-based query models: the NG4J API, the Jena ontology model API, and the Jena SPARQL API.

We generated two data sets using LUMB: a small data set consisting of 8832 statements, and a larger data set consisting of 103,048 statements. Next we formulated queries to return the name of the professor identified by a specific URI (*Q1*), and to search for the name of the professor with the email address *FullProfessor7@Department0.University0.edu* (*Q2*) and ran them on each data set. The tests ran on an Intel Core i7 (2.66GHz) Macbook Pro with 4GB RAM. Each query was repeated 200,000 times and the mean duration taken to reduce the effect of external factors on the performance measurement. Figure 3 shows the results of this comparison. Note that execution times are shown on a logarithmic scale.

Evaluation of Sapphire's performance is best done in comparison to NG4J, upon which it is built. The data show that Sapphire introduces a small overhead to the baseline NG4J performance. For example, for Q2 on the large data set, we see that the primary cost comes from operations delegated to the NG4J library, while the remaining cost is

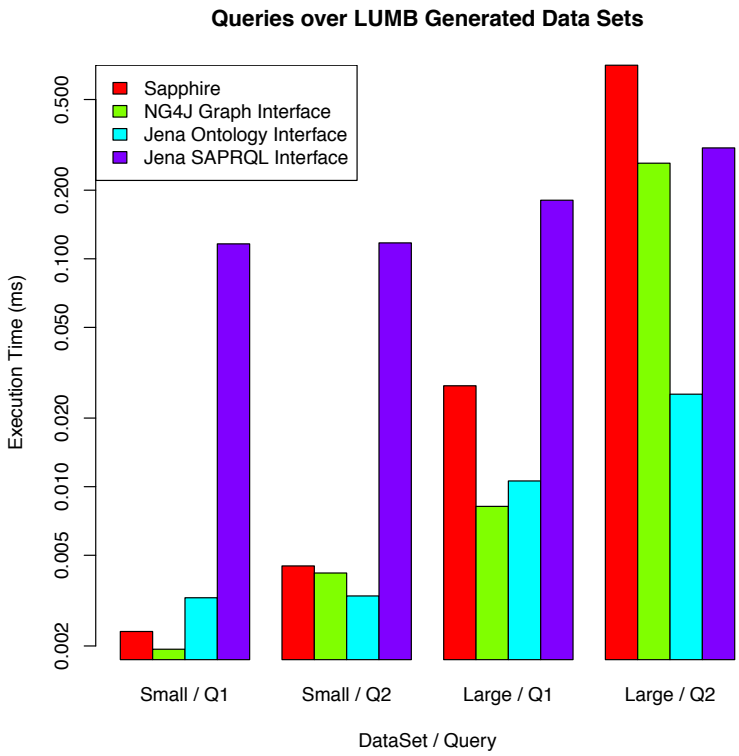

Fig. 3: Initial performance evaluation of Sapphire against traditional methods of querying OWL data using two queries over two data sets generated by LUMB.

due to the invocation of the `getName()` method on each `FullProfessor` instance in the data set. Examining the relative performance for both queries on each data set, we see that compared with NG4J, the Jena ontology API exhibits a superior performance ratio between queries that select single and multiple values from the model.

While further optimisations may reduce the overhead of the Sapphire library, we note that the execution time for Q2 on the larger data set remains under 1ms. This, we posit, is an acceptable overhead for the code brevity and type safety Sapphire provides. A more detailed evaluation of all of Sapphire's capabilities against data sets with different characteristics is planned for the future.

5 Related Work

Several approaches for working with RDF-based data models, that sit in a spectrum ranging from the ontology agnostic to the domain-specific, have been proposed. The

most generic APIs (e.g., the model APIs of *Jena* [11] and *NG4J* [12]), provide operations that support manipulation of the data-graph (e.g., the addition and removal of triples or quads). *Tramp* [15] binds RDF nodes to Python dictionary structures, allowing indexing by property names, while *SETH* [16], supports dynamic schema extension through the writing of inline RDF/OWL snippets in Python code. Increasing the level of abstraction, *The OWL API* [6], and *O3L* [17] are libraries based on OWL concepts and semantics that provide reflection-like capabilities for indexing into a data model using classes such as `OWLClass` and `OWLProperty` and `OWLRestriction`.

Towards the domain-specific end of the spectrum sit tools that map from object-oriented representation to ontology-oriented representation and vice versa. The former category of tools, to which *Jenabean* [18], *Sommer* [19], *Spira* [20], and *Empire* [21] belong, operate through manual annotation of the fields of Java Bean style objects, from which ontological descriptions are then extracted or persisted to a back-end. While this approach removes the need for developers to construct ontologies natively, these tools realise only a small subset of the RDFS and OWL specifications. Of the latter category, *Sparta* [22] binds RDF nodes to domain-typed Python objects and RDF arcs to attributes of those objects, while *ActiveRDF* [23] provides RDF data inspection capabilities via a Ruby-based virtual API that intercepts method calls and dynamically maps them onto SPARQL queries against a data model. ActiveRDF does not use schema information as part of its mapping process, and while this allows for great flexibility in terms of the ability to easily manipulate schema and structure of individuals at runtime), this comes as a necessary trade off against the ability to verify application code both in terms of type correctness and typographical errors against an ontology.

The tools closest in spirit to Sapphire generate domain-specific APIs for strongly-typed programming languages directly from an ontology. *RDFReactor* [13] generates Java libraries for RDFS models that act as a proxy to a triple store back-end at runtime, while *Owl2Java* [5] gives OWL ontologies a similar treatment. *OntoJava* [14] and the work of Kalyanpur et al. [4] (on which Jastor [24] is based), perform the conversion of both ontology axioms and rules into executable code. As we presented in Section 4.2, Sapphire builds upon the strong foundational work of these tools to provide improved support for OWL's feature set. Sapphire also offers novel features in the form of an extensible type mapping system, support for reification, support for the open-world assumption, and support for the dynamics of multiply-classified individuals.

Puleston et al. [25] present a case study supporting an approach that conflates both domain-specific and ontology-agnostic aspects within a single API. Applications constructed using this technique are assumed to be structured around a static core that is accessed in a domain-specific manner, with highly dynamic concepts at its edges that are accessed via a generic API. The authors advocate this approach for application domains where parts of an ontology are frequently evolved, or a high number of concepts (e.g., tens of thousands) makes code generation impractical. We note that one limitation the authors found with existing domain-specific APIs was the inability to evolve the type of existing instances, an issue that Sapphire addresses.

Finally, Parreiras et al. [26] introduce a Domain Specific Language, *agogo*, that allows developers to describe a mapping between OWL and a conceptual API. Independently specified transformations between the DSL and programming languages allow

the API to be realised in the developer's programming language of choice. The core novelty of this approach is support for *patterns* as first class objects in the DSL. This feature allows single calls in the generated API to be mapped to multiple operations on the underlying data store. Although Sapphire's extensible type registry can be thought of as providing similar functionality, the approach of Parreiras et al. lends itself to the generation of highly-customised APIs, something that we seek to investigate further.

6 Conclusion and Future Work

One goal of ontology-driven information systems engineering is to automatically generate software components from ontological descriptions. To this end, we presented a mapping from OWL to Java, and its realisation in the form of a tool, *Sapphire*. Building upon the state-of-the-art, our approach overcomes several limitations of existing tools by providing an extensible type mapping registry, supporting reification, accounting for the open-world assumption, and supporting the dynamic classification of OWL individuals in the Java model through a novel, dynamic-proxy-based solution. We demonstrated the primary benefits of Sapphire's API — code brevity and type safety — and quantified improvements to the state-of-the-art in the set of OWL's features supported. Through an initial performance evaluation, we showed that although Sapphire introduces an overhead compared with standard APIs, its performance remains competitive.

By profiling the codebase, we have observed that the main performance bottleneck comes from searches performed on the quad store. To mitigate this, we will devise data caching strategies to reduce such calls. Additionally, we are working towards fully supporting the feature set of OWL 2, and will investigate approaches to more closely integrate application logic with the generated artefacts.

Acknowledgements

This work has been supported by the EU-FP7-FET Proactive project SAPERE – Self-aware Pervasive Service Ecosystems, under contract no. 256873.

References

1. M. Uschold. Ontology-driven information systems: Past, present and future. In *Proceedings of Formal Ontology in Information Systems*, pages 3–18, Amsterdam, The Netherlands, 2008. IOS Press.
2. N. M. Goldman. Ontology-oriented programming: Static typing for the inconsistent programmer. In *Proceedings of the 2nd International Semantic Web Conference*, Sanibel Island, FL, USA, 2003.
3. M. Vanden Bossche, P. Ross, I. MacLarty, B. Van Nuffelen, and N. Pelov. Ontology driven software engineering for real life applications. In *Proceedings of the 3rd International Workshop on Semantic Web Enabled Software Engineering*, 2007.
4. A. Kalyanpur and D. Jimenez. Automatic mapping of OWL ontologies into Java. In *Proceedings of Software Engineering and Knowledge Engineering*, 2004.

5. OWL2Java: A Java code generator for OWL. Website, 2011. http://www.incunabulum.de/projects/it/owl2java.
6. The OWL API. Website, 2011. http://owlapi.sourceforge.net/.
7. J. R. Hobbs and F. Pan. An ontology of time for the semantic web. 3:66–85, March 2004.
8. E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of Adaptable and Extensible Component Systems*, 2002.
9. The Mindswap Pet Ontology. Website, 2011. http://www.mindswap.org/2003/owl/pet.
10. Y. Guo, Z. Pan, and J. Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal of Web Semantics*, 3:158–182, October 2005.
11. Website of the Jena Semantic Web Framework. Website, 2011. http://jena.sourceforge.net/.
12. C. Bizer, R. Cyganiak, and E. R. Watkins. NG4J Named Graphs API for Jena. In *Proceedings of the 2nd European Semantic Web Conference*, Heraklion, Greece, 2005.
13. M. Vökel. RDFReactor – from ontologies to programmatic data access. In *The Jena Developer Conference*, Bristol, UK, 2006.
14. A. Eberhart. Automatic generation of Java/SQL based inference engines from RDF Schema and RuleML. In *Proceedings of the First International Semantic Web Conference, Chia, Sardinia, Italy*, pages 102–116. Springer, 2002.
15. TRAMP: Makes RDF look like Python data structures. Website, 2011. http://www.aaronsw.com/2002/tramp/.
16. M. Babik and L. Hluchy. Deep integration of Python with web ontology language. In *Proceedings of the 2nd Workshop on Scripting for the Semantic Web*, Budva, Montenegro, 2006.
17. A. Poggi. Developing ontology based applications with O3L. *WSEAS Transactions on Computers*, 8:1286–1295, August 2009.
18. Jenabean: A library for persisting Java Beans to RDF. Website, 2011. http://code.google.com/p/jenabean/.
19. Sommer: Semantic object (medata) mapper. Website, 2011. http://sommer.dev.java.net/.
20. Spira: A linked data ORM for Ruby. Website, 2011. https://github.com/datagraph/spira.
21. Empire: A JPA implementation for RDF. Website, 2011. https://github.com/clarkparsia/Empire.
22. Sparta: a simple API for RDF. Website, 2011. https://github.com/mnot/sparta/.
23. E. Oren, R. Delbru, S. Gerke, A. Haller, and S. Decker. ActiveRDF: object-oriented semantic web programming. In *Proceedings of the 16th international conference on World Wide Web*, pages 817–824, New York, NY, USA, 2007. ACM.
24. Jastor: Typesafe, ontology driven RDF access from Java. Website, 2011. http://jastor.sourceforge.net/.
25. C. Puleston, B. Parsia, J. Cunningham, and A. Rector. Integrating object-oriented and ontological representations: A case study in Java and OWL. In *Proceedings of the 7th International Conference on The Semantic Web*, ISWC '08, pages 130–145, Berlin, Heidelberg, 2008. Springer-Verlag.
26. F. S. Parreiras, C. Saathoff, T. Walter, T. Franz, and S. Staab. APIs a gogo: Automatic generation of ontology APIs. *International Conference on Semantic Computing*, 0:342–348, 2009.