SemSon
Connecting Ontologies and Web Applications

Geert Vanderhulst, Kris Luyten and Karin Coninx
Hasselt-University – transnationale Universiteit Limburg
Expertise Centre for Digital Media, Wetenschapspark 2, 3590 Diepenbeek, Belgium

Keywords: Semantic web, Web-based applications, Ontologies.

Abstract: The emerge of semantic data on the web puts the development of dynamic web applications to the test. On the one hand, we witness more and more semantic information becoming available on the web. On the other hand, we can see web-based applications evolve from server-side applications to responsive client-side applications running in a web browser. In this paper we present a framework that bridges the gap between semantic data defined in OWL ontologies and client-side web applications implemented in JavaScript.

1 INTRODUCTION

The shift to dynamic Web applications running in a Web browser has brought up the need to interact with remote data from client-side scripts. At the same time, an increasing amount of meaningful information is published on the Web, making it possible for Web applications to better understand and satisfy requests between users and machines as envisioned by the semantic Web (Berners-Lee et al., 2001). Since most Web-based applications still rely on a fixed database schema, the semantics of the application data are often known in advance and interaction with a Web server's database can be abstracted using data objects. However, when developers start to adopt ontologies as a means to structure, query and share information, the full semantics of available data are not always known beforehand. For example, one ontology can extend another ontology with extra concepts and relations and hence give rise to a richer knowledge base. In this case, data objects should adapt to match the concepts in the aggregated knowledge base, in order to fully exploit the available information.

In this paper we present a framework, called SemSon, that dynamically maps information defined in an ontology on scripted data objects and vice versa. Even though there are fundamental differences between object-oriented programming (OOP) languages and knowledge representation frameworks such as OWL (Koide and Takeda, 2006), we show that a dynamic programming language such as JavaScript (JS) is very suitable for semantic data binding using OOP. We use SPARQL (Prud'hommeaux and Seaborne, 2008) and JSON[1] as glue to connect semantic data and JS objects on the fly. SemSon contributes to an improved scalability of Web applications that leverage semantic data by providing constructs to 1) map OWL instances on JS objects and 2) create new JS objects that adhere to an OWL class definition.

2 RELATED WORK

In (Koide and Takeda, 2006), the semantic gap between OOP languages and OWL is explained: various discrepancies between static OOP languages such as Java and OWL/RDF are due to a mismatch between OWL classes/individuals and OOP classes/instances. The authors show that these mismatches can be addressed by using a more dynamic and reflective OOP language such as Common Lisp Object System (CLOS) and present an OWL processor built on top of CLOS that allows programmers to develop application domain models using OOP. A more static approach for mapping an OWL ontology on Java classes and instances is presented in (Kalyanpur et al., 2004). In this work a solution is proposed to create a set of Java interfaces and classes from an OWL ontology whilst minimizing the impact of fundamental semantic differences. An instance of a Java class represents an instance of a single class of the ontology with most of its properties, class relationships and restric-

[1] http://json.org/

Vanderhulst G., Luyten K. and Coninx K.
SemSon - Connecting Ontologies and Web Applications.
DOI: 10.5220/0002770201630166
In *Proceedings of the 6th International Conference on Web Information Systems and Technology (WEBIST 2010)*, page
ISBN: 978-989-674-025-2

tion definitions maintained.

Our work focuses on the dynamic use of OWL constructs in JS to make meaningful information directly available to client-side Web applications. JavaScript, the default programming language to develop this type of applications, is just like CLOS very flexible in the way objects are composed: objects can be redefined and extended with new properties and methods at runtime. A growing interest of JSON in combination with semantic Web technologies such as SPARQL also motivates our research. For instance, SPARQL query results can be serialized into JSON and thus processed efficiently in JS using OOP (Clark et al., 2007).

3 OWL NOTATION IN JSON

Ontology experts think in terms of concepts and relations while many software developers think object-oriented. We connect both worlds by representing information defined in an OWL ontology using *class* objects and *individual* objects, expressed in JSON. Class and individual objects correspond to OWL class and individual descriptions respectively as depicted in figure 1. In both representations, URIs (namespaces and identifiers) are used to refer to objects (JSON) and resources (OWL). Note however that SemSon is *not* an API for OWL, but rather provides a means to generate JS objects from OWL DL semantics. We rely on the DL subset of the complete OWL vocabulary because it enforces a strict separation of classes, properties, individuals and data values which is also found in OOP languages.

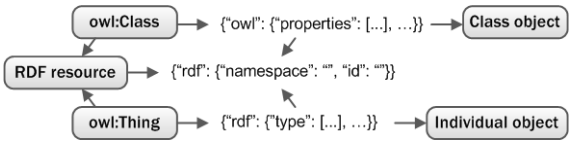

Figure 1: Mapping OWL to JSON.

In the remainder of this section we give a brief overview of the representation of OWL class descriptions in JSON and section 4 elaborates on the creation, use and validation of individual objects.

Class Descriptions. OWL classes can be described using a simple class identifier (URI reference) or built using the following constructs for which we provide a JSON equivalent:

- *Enumeration*: an OWL class can be described by exhaustively enumerating its individuals.

```
{"owl": {"oneOf": ["A", "B"]}}
```

- *Intersection, union, complement*: AND, OR and NOT operators can be applied to OWL class descriptions.

```
{"owl": {"intersectionOf": ["A", "B"]}}
{"owl":{"complementOf":"A"}}
```

- *Properties*: properties are linked with class descriptions through domain and range axioms. Restrictions on properties put additional constraints on the range of an OWL property when applied to a particular class description (value constraints) or on the number of values a property can take (cardinality constraints).

```
{"owl": {"properties": [{"uri": "P",
"range":["A","B"],"cardinality":
"1"}, {"uri": "Q", "hasValue": ["C"]}]}}
```

Class Axioms. OWL contains three language constructs for combining class descriptions into class axioms. These axioms describe inheritance, equivalence and disjointness relations that exist between classes.

```
{"owl": {"subClassOf": ["A"]},
        {"equivalentClass": ["B"]},
   {"disjointWith": ["C"]}}
```

This information enables a developer to reason about the class hierarchy and semantically compare classes.

4 SEMSON FRAMEWORK

The SemSon framework consists of a JavaScript library (semson.js) and a collection of Java-based servlets (semson.war) which connect scripts running in a Web browser with ontologies published on the Web, as shown in figure 2. We use Jena[2] as semantic data store (DS), Pellet (Sirin et al., 2007) as reasoner and SPARQL as query language. Table 1 lists the main operations supported by SemSon.

4.1 Import and Query Ontologies

Data is made available to Web applications through a semantic DS which acts as an application's database. Instead of using a fixed database schema, SemSon allows programmers to specify the ontologies they want to use at runtime and dynamically loads them into a semantic DS. Once in the DS, information from the ontology can be derived by constructing JS objects that correspond to a class or individual whose related data is transparently retrieved from the DS or by using SPARQL queries when more specific semantic

[2]http://jena.sourceforge.net/

data selection is needed. A preprocessing step generates native JS object wrappers for each OWL class description in an ontology. These wrappers support the creation of objects in OOP-style: new myclass() is equivalent with semson.i('myclass'). When preprocessing is omitted,e.g. when dealing with large ontologies with many class descriptions, class objects are resolved on an as-needed basis.

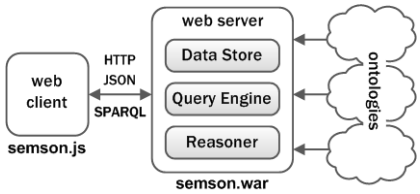

Figure 2: SemSon architecture.

Table 1: Main SemSon operations.

`import(location)`	Import an ontology into a semantic DS.
`c(c_uri)`	Create a class object from an OWL class description.
`i(c_uri,i_uri)`	Generate an individual object from an OWL class description.
`validate(i,c)`	Validate an individual object against a class object.
`update()`	Receive class or individual data from a semantic DS.
`commit()`	Send individual data to a semantic DS.

4.2 Create Individual Objects

In SemSon, objects are automatically generated from an OWL class description at runtime. Software developers only need a minimal understanding of the ontologies designed by domain experts such as available classes and their properties. Objects abstract underlying (instances of) ontologies and thus simplify working with semantic data. Furthermore, when an ontology is modified, data objects are updated accordingly: the application developer gets an updated data interface for free. Consider for instance an object that represents a person described in an ontology with namespace ns. This object is created as follows:

```
var p = new ns.Person();
// eq. semson.i('ns:Person');
```

To construct an instance of a class, we analyze the class description fetched from the ontology as described in section 3 and dynamically generate an object whose member variables match the OWL properties and their restrictions. For example, if a restriction on the ns:Person class defines that a person has zero or more hobbies, we generate a hobbies member variable that corresponds to an array:

```
p.firstName = 'John';
p.hobbies.push('Swimming');
```

Moreover, we have to take into account class descriptions based on intersection, union and complement as well as axioms such as rdfs:subClassOf when generating objects. These constructs specify the individuals that are supported by a class and thus also indicate valid properties for instances of classes. We select a superset of properties which allow to create valid class instances and map these on object member variables. Whether an instance is compliant to a class or not can be validated at runtime (see further). To avoid clashes with member variables, we use namespaces to differentiate between similar properties, e.g. obj.property versus obj.ns.property.

4.3 Validate Individual Objects

Most restrictions on properties and classes are hard to enforce while generating an individual object. An individual object can be manipulated at runtime and become incompatible with its class object. To detect inconsistencies, SemSon includes a validation algorithm implemented in JavaScript – validate(i,c), with i an individual object and c a class object – which compares the data of an individual object with its corresponding class object. The algorithm currently consists of three steps:

1. *Enumeration Checking*: check whether i equals one of the allowed individuals for c.
2. *Property Checking*: verify whether properties declared in an object are allowed, have valid values and a correct number of elements. This also involves class and datatype checking, for instance to assert that a property value indeed matches a class or value within a specified range.
3. *Intersection, Union, Complement Checking*: assert that an object passes the following tests: `validate(this,A) && validate(this,B)`, `validate(this,A) || validate(this,B)`, `!validate(this,A)`

The validation algorithm runs in a Web browser and is fast, but it does not carry out advanced reasoning techniques or complex datatype checking (e.g. user defined schema types). Its main purpose is to provide an efficient means to debug client-side Web applications by asserting that data objects are valid at strategic places in the program. Besides, the algorithm can be used to quickly validate user input by encapsulating input data in an individual object and

performing validation. However, to make sure a semantic DS remains consistent, a more powerful approach towards consistency checking is needed. This is achieved by installing an OWL reasoner such as Pellet (Sirin et al., 2007) between query engine and DS which analyzes data before making it persistent.

4.4 Semantic Data Binding

A class or individual object is filled with data from the DS by invoking its update function and data is sent back to the DS by executing the commit function. For performance reasons, we implemented these methods in SemSon servlets deployed on a Web server which have direct access to the DS. An update extracts data from the DS using the resource's URI as a reference and maps this information on JSON as depicted in figure 1. At the client-side script, the JSON data is converted into an object and convenience methods are added. A commit sends an object serialized into JSON to the Web server where it is checked by a reasoner for consistency and finally added to the DS. We also support an atomic commit operation in SemSon using transactions. In a transaction, multiple individual objects can be grouped together and committed to the DS in a single operation.

5 DISCUSSION AND CONCLUSIONS

As a proof of concept, we have developed a prototype Web application in which we use an OWL DL version of the Friend of a Friend (FOAF) vocabulary to create user profiles (see listing 1). The FOAF ontology is imported into the DS and is preprocessed so that OWL classes and individuals can be directly instantiated using OOP. In this example, we use a transaction to send the new and updated user profile to the DS and verify the information was stored correctly using a SPARQL query. When the ontology is extended with constraints on properties which e.g. state that each person must have a valid name, we can validate individual objects at runtime. Furthermore, the Pellet reasoner can be used to automatically infer missing information when adding data to a DS. For example, if FOAF is extended with family relations which typically include inverse relations such as A parentOf B $\Leftrightarrow$ B childOf A, the reasoner can automatically infer missing facts and add them to the DS to keep ontologies and their instances consistent at all times. We have presented a framework[3] for developing dynamic Web applications that can create and/or use semantic data from within a Web browser. SemSon is targeted toward programmers who are familiar with JavaScript, but only have a basic understanding of OWL and ontologies in general. Besides the FOAF example, we have used SemSon to create a Web interface for controlling a pervasive environment described using different domain-specific ontologies. For now, the DS is disadvantageous for the scalability of our approach since it is a centralized component. When facing large data sets (and associated complex ontologies) it needs to be replaced by a distributed storage and querying solution.

```
var foafns =
'http://www.mindswap.org/2003/owl/foaf';
var friend = 'http://myns/myfriend';
semson.registerNS('foaf', foafns);
semson.import(foafns, true);
// register class objects
var p1 = new foaf.Person();
// create individuals
p1.firstName = 'first'; p1.surname = 'last';
var p2 = new foaf.Person(friend);
p1.knows.push(p2.uri);
p2.knows.push(p1.uri);
semson.commit(p1, p2); // transaction
var q = 'SELECT ?p
WHERE {?p foaf:knows <'+ friend +'>}';
var qr = semson.select(q); // querying
for (b in qr.results.bindings)
    if (qr.results.bindings[b].p.value
    == p1.uri)
        alert('found!');
```

Listing 1: Creating a FOAF user profile using SemSon.

[3] http://research.edm.uhasselt.be/semson/

REFERENCES

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The Semantic Web. *Scientific American*, pages 34–43.

Clark, K. G., Feigenbaum, L., and Torres, E. (2007). Serializing SPARQL Query Results in JSON. http://www.w3.org/TR/rdf-sparql-json-res/.

Kalyanpur, A., Pastor, D. J., Battle, S., and Padget, J. A. (2004). Automatic Mapping of OWL Ontologies into Java. In *SEKE'04*, pages 98–103.

Koide, S. and Takeda, H. (2006). OWL-Full Reasoning from an Object Oriented Perspective. In *ASWC'06*, pages 263–277.

Prud'hommeaux, E. and Seaborne, A. (2008). SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., and Katz, Y. (2007). Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53.