Deep Integration of Python with Semantic Web Technologies

Marian Babik, Ladislav Hluchy *

Intelligent and Knowledge-based Technologies Group,
Department of Parallel and Distributed Computing, Institute of Informatics,
Slovak Academy of Sciences
Marian.Babik@saske.sk, Ladislav.Hluchy@savba.sk

Abstract. The Semantic Web is a vision for the future of the Web in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web. Semantic Web will build on the well known language stack, part of which is the Web Ontology Language (OWL). Semantic python (Seth) is a software effort to deeply integrate python and description logic (DL) subset of the OWL, e.g. extend python to seamlessly support logic programming paradigm. The deep integration of both languages introduces the notion of importing the OWL-DL into the programming context so that OWL classes can be used alongside classed defined normally. In this article we present a metaclass-based implementation of the deep integration ideas, which is a promising way of achieving such integration. The implementation is an early Python prototype supporting in-line class and properties declaration, instance creation and simple triple-based queries. The implementation is backed up by a well known OWL-DL reasoner Pellet. The integration of the Python and OWL-DL through meta-class programming provides a unique approach, which can extend the current python-based web frameworks and provide the needed support for the Semantic Web technologies.

1 Introduction

The deep integration of scripting languages and Semantic Web has introduced an idea of importing the ontologies directly into the programming context so that its classes are usable alongside classes defined normally. This can provide a more natural mapping of OWL-DL than classic APIs, reflecting the set-theoretic semantics of OWL-DL, while preserving the access to the classic Python objects. Such integration also encourages separation of concerns among declarative and procedural and encourages a new wave of programming, where problems can be

* Acknowledgments: The research reported in this paper has been partially financed within the Slovak National projects: Tools for acquisition, organization and maintenance of knowledge in an environment of heterogeneous information resources, SPVV 1025/04; Efficient tools and mechanisms for grid computing (2006-2008) VEGA 2/6103/6.

defined by using description logics [21] and manipulated by dynamic scripting languages [1]. The approach represents a unification, that allows both languages to be conveniently used for different subproblems in the software-engineering environment.

In this article we would like to introduce an early prototype, which implements some of the ideas of the deep integration in Python language [19]. It supports in-line declaration of OWL classes and properties, instance creation and simple triple-based queries [2]. We will emphasize the notion of modeling intensional sets (i-sets) through metaclasses. We will also discuss the possible drawbacks of the approach and the current implementation. In the next section we will provide a brief introduction to the Semantic Web and its languages. For a more detailed survey of the Semantic Web technologies and their comparison to the object-oriented systems see [4].

2 Semantic Web

Semantic Web is trying to make the web content machine-readable, so that it can be easily processed by the web agents and shared among the Web services. For that purpose the Semantic Web community has proposed a number of Web-based languages to formalize the web content such as RDF, RDFS and OWL [3, 2]. The key benefits of these languages are:

- Interoperability - models can be shared among applications on the web. Unlike standard XML languages RDFS and OWL have richer expressivity and can establish links among existing models.
- Flexibility - RDF and OWL models can be dynamic, i.e. classes and properties can be defined on-the-fly.
- Consistency and reasoning - it is possible to check if multiple models are consistent and in case of OWL it possible to use automated reasoning tools to infer new facts about the model.

Resource Description Framework (RDF) is a Web-based language, that can be used to describe associations between resources. A resource can be anything identified by the Universal Resource Identifier (URI), e.g. *http://www.w3.org/People/EM/contact#Marian*. In RDF resource can be classes, properties or instances. The low-level building blocks of the RDF are triples, i.e. sets of subject, property and object. Any object from one triple can play the role of a subject in another triple. Furthermore, any RDF statement itself can be subject or object. RDF can then be seen as a graph, where nodes correspond to subject or objects and properties correspond to arcs. RDF has also a serialization format defined in XML allowing users to share the models on the Web. Plain RDF makes no data modeling commitment and has no mechanisms for declaring vocabulary with semantic meaning.

RDF Schema (RDFS) extends RDF by defining simple hierarchy of concepts and properties, i.e. type system. In schemas, new resources can be defined as specialization of old ones, thus allowing to infer implicit triples. Schemas also

constrain the context in which defined resources may be used (i.e. validity). Based on the first-order logic those two notions can be seen as one, since they all can be expressed by rules allowing to infer new facts (i.e. new triples). Apart from RDF and RDFS there also other languages that can have better readability and can combine the expressivity of the RDFS with rules. Examples of such languages are Notation3 (N3)[1], NTriples[2] and Turtle[3].

Web Ontology Language (OWL) extends RDF Schema and adds ability to express more information about characteristics of properties and classes, e.g. define classes by grouping instances that meet certain characteristics, define value restriction, existential quantification, etc. A decidable sublanguage of the OWL, called OWL-DL, is based on the description logic (DL). This means, that OWL-DL can use the automated reasoning techniques of the DL to reveal subclass/superclass relationships among classes, determine the most specific types of individuals, detect inconsistent class definitions, etc.

There are many existing RDF/OWL APIs, that offer generic methods such as getStatement, getTriples, getSubject, etc. Such APIs provide all the necessary functionality, however their usage assumes a very detailed knowledge of the semantic technologies and languages. A more usable API has been advocated [1], which should integrate OWL/RDF and scripting languages. However, designing an object-oriented API for the RDF/OWL data is quite complex due to the following issues:

- The semantics of the RDF/OWL are very different from semantics of the object-oriented schemas. For a detailed comparison, see [4].
- RDF/OWL can be untyped, i.e. type of an object can be inferred during the runtime.
- Multiple inheritance of both classes and properties is allowed. Furthermore properties are standalone entities, that can exists without any classes.
- RDF/OWL is designed to integrate heterogeneous data with varying structure. The schemas used (RDFS/OWL) can evolve over time, which requires a flexible API.

Dynamic scripting languages, such as Python, can address these issues more transparently than statically-typed languages. Scripting languages have dynamic type system and types can be defined during the runtime. It is possible to support multiple inheritance and with meta-programming capabilities one can generate objects on the fly and override internal type system. Interpretability is also an advantage, since one can generate APIs on the fly and change them to reflect the set-theoretic semantics of the OWL. We have chosen Python due to its simplicity, numerous powerful libraries and its nature to support different programming paradigms.

[1] http://www.w3.org/DesignIssues/Notation3.html
[2] http://www.w3.org/2001/sw/RDFCore/ntriples/
[3] http://www.dajobe.org/2004/01/turtle/

3 Intensional Sets and Metaclasses

Intensional sets as introduced in [1] are sets that are described with OWL DL's construct and according to this description, encompass all fitting instances. A sample intensional set can be defined by using Notation3 (N3) as, e.g. ":Person a owl:Class; rdfs:subClassOf :Mortal". This simply states that Person is also a Mortal. Assuming we introduce two instances, e.g. ":John a :Person and :Jane a :Mortal", the instances of *Mortal* are both John and Jane. Please note, that N3 is used only for demonstration purposes, since it is more readable than XML serializations.

A metaclass-based implementation of the intensional sets is based on the core metaclass Thing, whose constructor accepts two main attributes, i.e. default namespace and N3 description of the intensional set. The instance of the metaclass is then a mapping of the OWL class to the intensional set. Following the above example class Person can be created with a Python construct:

```
Person = Thing('Person', (),
{defined_by: 'a owl:Class; rdfs:subClassOf :Mortal',\
 namespace: 'http://samplens.org/test#'})
```

This creates a Python class representing the intensional set for Person and its namespace. In the background it also updates the knowledge base with the new assertion. The individual John can then be instantiated simply by calling $John = Person('John')$. This statement calls the default constructor of the class Person, which provides support for asserting new OWL individual into the knowledge base. A similar metaclass is used for the OWL property except that it can not be instantiated. The constructor is used here for different purpose, i.e. to create a relation between classes or individuals. The notion of importing the ontology into the Python's namespace is then a matter of decomposing the ontology into the groups of intensional sets, generating Python classes for these sets and creating the instances.

Since metaclasses act like regular classes it is possible to extend their functionality by inheriting from the base metaclass. It is also simple to hide the complex tasks needed for accessing the knowledge base, reasoner and processing the mappings between OWL-DL's concepts and their respective Python counterparts.

4 Sample session

A sample session shows an in-line declaration of a class *Person*. This is implemented by calling a static method *new* of the metaclass *Thing* (the static method *new* is used to hide the complex call introduced in Sec. 3). An instance is created by calling the *Person's* constructor, which in fact creates an in-line declaration of the OWL individual (*John*). The print statements show the Python's view of the respective objects, i.e. *Person* as a class and *John* as an instance of the class *Person*. We also show a more complex definition of the *PersonWithSingleSon*, which we will later use to demonstrate the reasoning about property assertions.

```
>>> from Thing import Thing
>>> Person = Thing.new('Person a owl:Class .')
>>> John = Person('John')
>>> print Person
<class 'Thing.Person'>
>>> print John
<Thing.Person object at 0xb7d0b50c>
>>> PersonWithSingleSon = Thing.new("""PersonWithSingleSon \
      a  owl:Class ; rdfs:subClassOf   [ a  owl:Restriction ;
     owl:cardinality "1"^^<http://www.w3.org/2001/XMLSchema#int> ;
     owl:onProperty :hasSon
                                        ] ;
     rdfs:subClassOf  [ a  owl:Restriction ;
     owl:cardinality "1"^^<http://www.w3.org/2001/XMLSchema#int> ;
     owl:onProperty :hasChild ] .""")
```

A similar way can be used for in-line declarations of OWL properties. Compared to a class declaration the returned Python class can not be instantiated (i.e. returns *None*).

```
>>> from PropertyThing import Property
>>> hasChild = Property.new('hasChild a owl:ObjectProperty .')
>>> print hasChild
<class 'PropertyThing.hasChild'>
>>> hasSon = Property.new('hasSon a owl:ObjectProperty ;
                           rdfs:subPropertyOf :hasChild .')
```

Properties are naturally used to assign relationships between OWL classes or individuals, which can be as simple as calling:

```
>>> Bob = PersonWithSingleSon('Bob')
>>> hasChild(Bob, John)
```

Assuming we have declared several instances of the class *Person* we can find them by iterating over the class list. It is also possible to ask any triple like queries (the query shown also demonstrates reasoning about the property assertions).

```
>>> for individual in Person.findInstances():
...     print individual, individual.name
<Thing.Man object at 0xb7d0b64c> Peter
<Thing.Person object at 0xb7d0b50c> John
<Thing.Person object at 0xb7d0b6ec> Jane

>>> for who in hasSon.query(Bob):
...     who.name
'John'
>>> print hasSon.query(Bob, John)
1
```

5 Implementation and Drawbacks

Apart from the metaclass-based implementation of the i-sets, it is necessary to support query answering, i.e. provide an interface to the OWL-DL reasoner. There are several choices for the OWL-DL reasoners including Racer, Pellet and Kaon2 [15, 14, 29]. Although these reasoners provide sufficient support for OWL-DL their integration with Python is not trivial since they are mostly based on Java (except Racer) and although they can work in server-like mode Python's support for standard protocols (like DIG) is missing. Other possibilities are to use Python-based inference engines like CWM, Pychinko or Euler [23–25]. However, due to the performance reasons, lack of documentation or too early prototypes we have decided to use Java-based reasoners. We have managed to successfully use JPype [26], which interfaces Java and Python at native level of virtual machines (using JNI). This enables the possibility to access Java libraries from within CPython. Having the ability to call Java and use all the capabilities of the current version of CPython (e.g. metaclasses) is a big advantage over the other approaches such as Jython or JPE [22, 20]. We have developed a wrapper class, which can call Jena and Pellet APIs and perform the needed reasoning [16, 14]. The wrapper class is implemented as a singleton and interfaces the Python calls to the reasoner and RDF/OWL API and forwards it to the JVM with the help of the JPype.

One of the main drawbacks of the current implementation is the fact, that it doesn't support open world semantics (OWA). Although the reasoner in the current implementation can perform OWA reasoning and thus it is possible to correctly answer queries, the Python's semantics are based on the boolean values. One of the possibilities is to use epistemic operator as suggested in [1], however this is yet to be implemented. Another problem when dealing with ontologies are namespaces. In the current prototype we have added a set of namespaces that constitute the corresponding OWL class or property description as an attribute of the Python's class. This attribute can then be used to generate the headers for the N3 description. This approach needs further extension to support management of different ontologies. One of the possibilities would be to re-use Python's module namespace by introducing a core ontology class. This ontology class would serve as a default namespace handler as well as a common importing point for the ontology classes.

The other drawback of the approach is the performance of the reasoner, which is due to the nature of the JPype implementation (the conversions between virtual machines imposes rather large performance bottlenecks). This can be solved by extending the support for other Python to Java APIs and possible implementation of specialized client-server protocols.

6 Related Work

The approach of mapping RDF objects to the object-oriented programming language is not novel and is similar to the existing Python projects such as Tramp

[17] and Sparta [30]. The projects are concerned mainly with RDF and doesn't address OWL. There are also standard Python APIs for RDF and OWL including rdflib [5], CWM[23], Pychinko[24] and 4Suite [6]. These libraries are usually based on the standard statement-centric API paradigm and although they provide excellent support for the basic manipulation with RDF and OWL, their support for the OWL-DL reasoning is limited. There are similar projects to Seth written in other scripting languages such as RDFReactor written in Ruby [13]; RDFHomepage and RDF World written in PHP [8, 9].

The most popular existing OWL APIs, that provide programmatic access to the OWL structures are based on Java [16, 10, 18]. Since Java is a frame language its notion of polymorphism is very different than in RDF/OWL. This is usually solved by incorporating design patterns, which make the APIs quite complex and sometimes difficult to use. The dynamic nature of the scripting languages can support OWL/RDF level of polymorphism and thus it is possible to directly expose the OWL structures as Python classes without any API interfaces. Implementation of the deep integration ideas with statically-typed languages has been attempted in projects such as RdfReactor [13], Elmo[11], Kazuki[12] and Jastor[27]. These projects however assume existence of a stable schema and assume conformance to such schema, which in dynamic scripting language is not necessary. There are also initiatives for mapping relational databases to objects, as done with Enterprise Java Beans, or in the project SQLObject[4].

7 Conclusion

We have described a metaclass-based prototype implementation[5] of the deep integration ideas, which is available under MIT license[6]. We have discussed the advantages and shortcomings of the current implementation. We would like to note, that this a work in progress, which is constantly changing and this is just a report of the current status. There are many other open questions, that we haven't covered here including integration of query languages (possibility to reuse ideas from native queries [28]); serialization of the ontologies; representation of rules, concrete domains, etc. We hope that having an initial implementation is a good start and that its continuation will contribute to the success of the deep integration of the scripting and Semantic Web.

References

1. Vrandecic, D., Deep Integration of Scripting Languages and Semantic Web Technologies, In Proc. of 1st International Workshop on Scripting for the Semantic Web SFSW 2005 , volume 135 of CEUR Workshop Proceedings. CEUR-WS.org, Greece, 2005, ISSN: 1613-0073

[4] http://www.sqlob ject.org
[5] http://seth-scripting.sourceforge.net
[6] http://seth-scripting.sourceforge.net/license

2. Web Ontology Language (OWL), see http://www.w3.org/TR/owl-features/
3. Resource Description Framework (RDF), see http://www.w3.org/RDF/
4. A Semantic Web Primer for Object-Oriented Software Developers, http://www.w3.org/TR/sw-oosd-primer/
5. RDFLib, http://rdflib.net/
6. 4Suite, http://4suite.org/
7. Oren, E., Delbru, R., ActiveRDF: object-oriented RDF in Ruby In Proc. of Scripting for the Semantic Web Workshop at the ESWC, Budva, Montenegro, June 12, 2006, CEUR Workshop Proceedings, ISSN 1613-0073, online CEUR-WS.org/Vol-181/
8. Grimnes, G.A., Schwarz, S., Sauermann, L., RDFHomepage or "Finally, a use for your FOAF file" In Proc. of Scripting for the Semantic Web Workshop at the ESWC, Budva, Montenegro, June 12, 2006, CEUR Workshop Proceedings, ISSN 1613-0073, online CEUR-WS.org/Vol-181/
9. Snyder, C.: (Rdfworld.php), http://chxo.com/rdfworld/index.htm
10. Sesame, http://www.openrdf.org/
11. Elmo, http://www.openrdf.org/doc/elmo/users/index.html
12. Kazuki, http://projects.semwebcentral.org/projects/kazuki/
13. Volkel, M., Sure, Y.: RDFReactor From Ontologies to Programmatic Data Access, In: Poster Proceedings of the Fourth International Semantic Web Conference. (2005) http://rdfreactor.ontoware.org/.
14. Pellet OWL Reasoner, see http://www.mindswap.org/2003/pellet/index.shtml
15. RacerPro Reasoner, see http://www.racer-systems.com
16. Jena: A Semantic Web Framework for Java, see http://www.hpl.hp.com/semweb/jena2.htm.
17. TRAMP: Makes RDF look like Python data structures http://www.aaronsw.com/2002/tramp, http://www.amk.ca/conceit/rdf-interface.html
18. Bechhofer, S., Lord, P., Volz,R.:Cooking the Semantic Web with the OWL API. 2nd International Semantic Web Conference, ISWC, Sanibel Island, Florida, October 2003
19. G. van Rossum, Computer programming for everybody. Technical report, Corporation for National Research Initiatives, 1999
20. Java-Python Extension, http://sourceforge.net/projects/jpe
21. Baader, F., Calvanese, D., McGuinness, D.,L., Nardi, D. and Patel-Schneider,P., F. editors. The description logic handbook: theory, implementation, and applications. Cambridge University Press, New York, NY, USA, 2003.
22. Jython, Java implementation of the Python, http://www.jython.org/
23. Closed World Machine, see http://www.w3.org/2000/10/swap/doc/cwm.html
24. Katz, Y., Clark, K. and Parsia, B., Pychinko: A native python rule engine. In International Python Conference 05, 2005.
25. Euler proof mechanism, see http://www.agfa.com/w3c/euler/
26. JPype, Java to Python integration, see http://jpype.sourceforge.net/
27. Kalyanpur, A., Pastor, D., Battle, S. and Padget, J., Automatic mapping of owl ontologies into java. In Proceedings of Software Engg. - Knowledge Engg. (SEKE) 2004, Banff, Canada, June 2004.
28. Cook, W. R. and Rosenberger, C., Native Queries for Persistent Objects, Dr. Dobb's Journal, February 2006
29. Hustadt, U., Motik, B., Sattler, U.. Reducing SHIQ Description Logic to Disjunctive Datalog Programs. Proc. of the 9th International Conference on Knowledge Representation and Reasoning (KR2004), June 2004, Whistler, Canada, pp. 152-16
30. Sparta, Python API for RDF, see http://www.mnot.net/sw/sparta/