Rethinking Service Oriented Architectures in the Semantic Web

Stefano Bocconi
Cyntelix
Piet Mondriaanplein 13
3812 GZ Amersfoort, The Netherlands
sbocconi@cyntelix.com

Stefano Travelli
Cyntelix
Piet Mondriaanplein 13
3812 GZ Amersfoort, The Netherlands
stefano.travelli@gmail.com

ABSTRACT

By building a Service Oriented system on top of a semantic repository, the expectation is to get the best of both worlds, i.e. strong modelling capabilities exposed by reusable services, which can be exploited as building blocks for third-parties applications. In reality some "mismatches" make the combination not always optimal. Logic languages such as OWL and object-oriented languages such as Java have different paradigms. On the one hand, Software Oriented Architectures tend to favor stable interfaces for Web Clients, and to offer simplified versions of the objects used by the business logic on the server. On the other hand, Semantic Web clients make use of the rich semantics exposed by a semantic repository, which can be highly dynamic. This paper discusses these mismatches and suggests a different role for Software Oriented Architectures when integrated with Semantic Web technologies. The discussion is based on the experience in building a Web user interface using the services provided by a Service Oriented Architecture on top of a semantic repository.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.2.12 [**Software Engineering**]: Interoperability—*Data mapping, Interface definition languages*; H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Data sharing, Web-based services*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Semantic networks*

General Terms

Design, Languages, Theory

1. INTRODUCTION

To foster adoption of the Semantic Web, its community has always tried to reach out to Java programmers, since Java is a widely used language in programming the web. Different tools have therefore emerged that facilitate the use of RDF and OWL with Java. Often these tools attempt to translate RDF and OWL classes to Java classes, since the latter is something programmers are very familiar with. By doing that though, the risk is to overlay the Object Oriented Paradigm (OOP) on the knowledge representation and logic based paradigm of the Semantic Web. Mapping Java classes to OWL classes and vice versa can cause loss of semantics. More importantly, this approach resembles how Java handles persistency of data by mapping objects to database tables. Programmers might be led to believe that the Semantic Web is yet another database technology, only more complicated (and with worse performances). This is even a more serious problem, as developers would only see the burden of this new technology, without exploiting the advantages that the Semantic Web offers, such as rich semantics and support for knowledge discovery.

The latter is particularly true when trying to add semantics to well established software architecture paradigms and patterns, such as Service Oriented Architecture (SOA). A SOA approach exposes services that provide serialized versions of the resources being handled by the server. These serialized versions need to be considerably simpler than the data objects used by the backend, to minimize overhead in data transmission. Stripping down resources risks to remove also semantics that a Web client might have exploited. This is in contrast with the fact that Semantic Web applications make use of the semantics contained in the data, and can be developed by third parties at a later stage, for purposes not yet envisioned when the SOA architecture was designed.

Loss of semantics can also be caused by another "good" programming practice, i.e. fix interfaces between components. According to this practice, interfaces (in terms of data and functions or services) should be stable so that components can be developed in parallel. This mostly well justified practice can be at odd with the dynamicity that the Semantic Web supports: ontologies can evolve while programming an application, since the knowledge of the domain evolves, without having to heavily re-engineer the data structure. Keeping the interface stable means to miss the latest version of the data, which, in case of semantic data, also means to miss possibly useful knowledge about the domain.

This paper discusses these issues in the light of the lessons learnt while implementing an intelligent user interface on top of a SOA with a semantic repository backend. This was done in the scope of Cloud4SOA, a EU FP7 project that address the lack of interoperability between heterogeneous PaaS[1] offerings. Cloud4SOA has been described in [8], and its current release is available at `http://demo.cloud4soa.eu/cloud4soa/`. In this paper we concentrate on the User Interface, which provides functionality such as user profile management, search for PaaS offerings, deployment of applications on PaaS providers and monitoring performances of deployed applications. The interface calls SOA services that expose a semantic representation of the domain of Cloud Computing. Cloud4SOA uses Java and OWL for the semantic model, therefore this paper focuses on these two languages.

The remainder of this paper is organized as follows: section 2 introduces the problem of combining technologies based on different paradigms; section 3 presents existing tools to map Java classes to RDFS/OWL ontologies; the design and architecture of Cloud4SOA is presented in section 4, with particular attention to the interface and the limitations we ran into as a consequences of design choices; to alleviate these limitations, we introduce in section 5 the concept of SOA in semantic times, and we conclude in section 6 with some possible modifications to the SOA paradigm to accommodate the new role of semantics.

2. IMPEDANCE MISMATCH

In electric engineering, impedance mismatch indicates the situation when two electric circuits of different impedance are coupled, resulting in a not optimal transfer of energy due to the difference in impedance. Software engineering has borrowed this term to describe the case when two technologies based on different paradigms need to talk to each other, but concepts of each technology do not have an exact match to concepts of the other technology.

Originally the term was used when trying to couple Object Oriented Paradigm (OOP) languages with Relational Database technology, and to map objects to relations. In this case the mismatch is considerable, since relational databases have no such concepts as classes, inheritance, interfaces and data encapsulation. More recently, the term has also been used when trying to use OOP with Semantic Web technology[2]. In this case the match is closer since OWL has classes and inheritance. Nevertheless, Java does not capture OWL semantics naturally. Characteristics that do not find an exact match are [5]:

- multiple inheritance
- reification of statements
- class definition as the union, intersection, or complement of other classes. Also as the equivalent to or disjoint with other classes, or as an enumerated set of individuals
- properties as first class objects, with restrictions on domain and range in terms of data types and data ranges (cardinality), and properties such as *rdfs:subPropertyOf*, *owl:TransitiveProperty*, *owl:inverseOf*
- dynamically inferring the type of an individual (i.e. determining the class of which an individual is an instance)
- open-world assumption, which states that statements not explicitly entailed by the ontology might be true or false

[1]PaaS stands for Platform as a Service, and it is one of the Cloud service models. A PaaS Cloud provider offers users managed services such as Application Servers and databases.

[2]In this case the term Impedance mismatch 2.0 is used [1].

On the other hand, directs translation of conceptual models to sets of executable artifacts yields many advantages. A mapping between Java and OWL ontologies is beneficial to Java developers, for several reasons. Firstly, developers can handle Java objects instead of OWL statements, and read and write data using familiar syntax, such as *project.setName("Cloud4SOA")* instead of *addTriple(projectURI, setNamePropertyURI, "Cloud4SOA")*[3]. Secondly, typographic errors (such as typos of name of classes or properties) can be detected at compile time. The generation of Java code from OWL ontologies can also be motivated on a more abstract level, by considering ontologies to play a similar role as UML models in model-driven architectures [6].

In the next section we discuss some existing tools that support the mapping Java-OWL.

3. EXISTING APPROACHES AND TOOLS

RDFReactor [7] generates Java classes from an RDF Schema using Velocity templates[4]. RDFReactor stores the state of the objects directly in the triple store using RDF2Go[5], an abstraction layer over several triple stores such as Sesame[6].

In Cloud4SOA RDFReactor has been used initially to generate the classes from the model. Subsequently, while the model was evolving, the mapping has been performed manually using RDFBeans, in order to have better control over it and maintain a more succinct code base. RDFBeans[7] is an object to RDF mapping library for Java, which provides a framework to map an RDF model to an object-oriented model. Using annotations in the Java code, RDF classes and properties are mapped to Java classes and properties. Both approaches support (a subset of) RDFS semantics only, but no OWL semantics (with the exception of cardinality constraints in RDFReactor). This was not a problem for Cloud4SOA as RDFS semantics was mostly sufficient to model the Cloud computing domain.

Other approaches exist that address the OWL semantics, such as HarmonIA [3] and Sapphire [5]. HarmonIA approximates multiple inheritance, and class intersection, union and disjoint, using Java interfaces. Sapphire addresses most of the Java-OWL semantics mismatches presented in the previous section, by generating Java interfaces that behave as proxies for the real instances. Calls to those instances are marshaled through handlers that perform OWL semantics checking and, if no inconsistency is detected, forward the call to the semantic repository. Not all OWL characteristics are covered: Sapphire does not handle OWL entailment rules in Java, and relies on a OWL reasoner to do so.

4. CLOUD4SOA ARCHITECTURE

[3]Example derived from `http://semanticweb.org/wiki/RDFReactor`

[4]`http://velocity.apache.org/`

[5]`http://semanticweb.org/wiki/RDF2Go`

[6]`http://www.openrdf.org/`

[7]`http://rdfbeans.sourceforge.net/index.html`

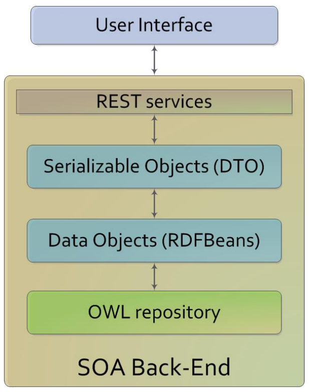

Figure 1: Cloud4SOA architecture from a data point of view.

In designing Cloud4SOA, we started from established software engineering principles. Many best practices of software engineering emphasize the separation of concerns between business logic, data objects and presentation layer. According to this separation, the presentation layer has a very limited knowledge of the business logic. This avoids the risk of breaking the code in case the logic of the application changes.

When designing an application for different clients, possibly developed by third-parties, a Service Oriented Architecture is often adopted. The functionality is broken down and provided with services API that client applications can "repackage" to offer, for example, task-based interfaces to users, as in Cloud4SOA's case. Services offered are often of the REST type, i.e. they offer a representation of a given resource, without keeping the state the requesting client might be in. In order to minimize breaking the code of the clients, the interface should be as stable as possible, since interfaces constitute the contract that both parties need to respect (contract-driven approach). Even if the data objects change on the server side, REST offers a representation of the resources, which means that the representation can be kept stable by mapping the new data objects to the same representation, even when these data objects change. The Java language, and OOP language in general, have a design pattern that is used to decouple the objects stored in a database back-end with the objects that are presented to, for example, an user interface: the Data Access Object (DAO)/ Data Transfer Object (DTO) paradigm. DAOs encapsulate the logic to retrieve data model instances from the underlying DBMS, and offer via the public interface the required objects to the application logic. DTOs, on the other hand, provide a view on these objects for remote components (such as a client UI interface). This separation is needed for mainly these reasons:

- Data object instances might not be easily serializable to be transmitted over the wire. For example, objects might be densely interconnected via object attributes, so that serializing an object would require to include many other objects
- The client should not know about the internal details of the model
- The clients need a stable view on the data, which does not change when the model needs to change.

All these factors led to the design of the platform schematically depicted in Figure 1. The domain of Cloud Computing was modeled using OWL, by modeling Users, Software Applications and Cloud Platforms[8]. The ontology was then translated to Java classes with RDFReactor and kept aligned with the Java code using RDFBeans annotations.

For the considerations discussed previously, two distinct layers of Java classes were implemented: the first layer consists of classes with a one to one correspondence to the ontology concepts, via RDFBeans; the second layer consists of classes that wrap the first level, in order to offer a more stable interface, with serializable objects (DTOs) that can be exposed via the service interface. In this way, while the ontology and consequently the first layer of objects were evolving, the functionality of the REST interface based on the second layer was constant.

4.1 User Interface

The interface supports software developers in searching for suitable Cloud offerings where to deploy their application. Even though the architecture allowed to rely on stable services, it was soon evident that our design had some drawbacks. Firstly, there was a delay between the moment the ontology would evolve and the moment the services would reflect the changes. Services would evolve when enough important changes in the ontology and the mapping layer would make it worth to modify the DTO layer. Secondly, even though not so often, the interface code had to change when the services would change. This was due to the fact that relying on a fixed structure of the REST resources would tend to produce code that was not so resilient to data structure changes. The layout was mostly affected, since the interface was used to let the user fill in the data structures. Having to fix the layout while the project was still maturing the semantic data structures was not optimal.

These problems led to the idea of dynamically generating the interface layout based on the data structure, to avoid being forced to redesign the layout when fields were added, modified or removed from the data schemas. Furthermore, two additional tasks required to be able to inspect a semantically rich model instead of a simplified and possibly outdated version as offered by the DTO layer. The first one was to allow navigation of the PaaS offerings in the repository, for which a facet browsing solution was very appropriate. The second one was to provide translations between the terminologies used by different providers, by exploiting the term equivalences contained in the ontology.

Consequently, part of the interface (facet browsing) was modified to access directly the ontology using SPARQL [2], so that the interface at runtime could query for the model objects and for their properties to display. Other functionality was not migrated to use SPARQL, since this would have required a considerable overhead. To increase flexibility though, an internal domain specific language[9] (DSL) was defined to specify how data needed to be represented in visual form, including field validation and decoration (labels, tooltips and so on). Such specifications could ideally be generated by rules based on the semantic model. In the next

[8]The model is described in [8].

[9]`http://www.martinfowler.com/bliki/DomainSpecificLanguage.html`

section we will see another useful application of DSL.

5. SOA IN SEMANTIC TIMES

The experience built in the project led to reconsider some assumptions that are generally made in SOA. In line with the semantic web, interfaces might need to be intelligent, and not thin. This might require to think whether the business logic needs necessarily to reside mostly on the server side, and whether the advantages to have stable interfaces are enough to "serialize away" or freeze the semantics in the data.

A strong reason in support of dealing with (simplified) objects is that consistency can be enforced at compile time. A wrong SPARQL query can be detected only at runtime, while a field in an object is checked by the compiler. Recently though there are more technologies that rely on test checking more than on compile checking, such as dynamic languages like Ruby, where fields can be dynamically added to objects and the code needs to be checked at runtime. Setting up an efficient framework for testing might compensate for the lack of compile time checking. Moreover, a clear disadvantage of mapping data structure to objects is then that the code needs to be recompiled when the data changes. Although this is alleviated by tools such as the ones we discussed in section 3, this is not always an ideal solution.

6. CONCLUSIONS

The impedance mismatch problem we discussed in section 2 is a reason against mapping Java objects to OWL classes, although in our view not a strong one. Currently most applications are still not fully exploiting the rich semantics OWL offers, and that was true for Cloud4SOA as well. In our opinion, the strongest motivation to adopt a mapping Java-OWL is that developers cannot be forced to deal with OWL on the level of statements. Developers are used to have abstractions that allow them to deal with objects, which are transparently populated from the data storage with standards such as Java Persistence API[10]. Having to create objects and populate them with several SPARQL queries is not a realistic alternative because of the overhead it implies, even if it leads to more semantics available for the applications.

Moreover, in case developers would deal directly with the repository with SPARQL, a SOA architecture would not make sense anymore, since the SPARQL endpoint would provide most (if not all) of the services needed by the client applications. Considering the overhead that this approach would cause to the developers, as we discussed above, we might rethink the role of the SOA architecture in a semantic setting.

We think that a "Semantic" SOA could take an approach like Agogo [4], which adopts a DSL to define patterns that describe complex mapping between objects and OWL class and properties. These patterns are first class objects and can map several OWL classes to a single object, and multiple operations on the underlying data to a single API call. Moreover, the DSL can be checked at compile time for validity and can be used to generate automatically software using independently developed transformations.

A semantic SOA could therefore specify *views* on the underlying data, using a declarative language that developers should be able to view, reuse and modify. Such views would also encode procedural knowledge, for example by assembling in the same delete API call an operation on several OWL statements, in this way retaining the procedural nature of services. Being declarative would allow for modifications to include more semantics from the repository if needed. Features such as compile-time checking and code generation would also be highly desirable, although checking could also be performed by running them automatically against the semantic repository in a sort of unit testing.

We believe that such an approach would represent a middle ground between the classic SOA approach (stable interface but with *frozen* semantics) and leaving developers to deal with a SPARQL end-point (free access to semantics but lack of guidance in an unfamiliar environment). In the future we plan to experiment with this approach to evaluate whether it can alleviate the shortcomings presented in this paper.

[10]`http://jcp.org/en/jsr/detail?id=317`

7. ACKNOWLEDGMENTS

This work is partially funded in the context of Cloud4SOA, under the contract No. 257953 within the Seventh Framework Programme of the European Community. The authors wish to thank Filippo Bosi, Vincenzo Laudizio, Franco Ravagli and Yosu Gorroñogoitia who helped stimulate the thoughts contained in this paper.

8. REFERENCES

[1] M. L. Garshol. Impedence mismatch 2.0, 2010.

[2] S. Harris and A. Seaborne. SPARQL 1.1 Query Language, 2013.

[3] A. Kalyanpur, D. J. Pastor, S. Battle, and J. Padget. Automatic Mapping of OWL Ontologies into Java. In F. Maurer and G. Ruhe, editors, *Proceedings of Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2004.

[4] F. Silva Parreiras, T. Walter, S. Staab, C. Saathoff, and T. Franz. APIs agogo: Automatic Generation of Ontology APIs. In *Proceedings of the 3rd IEEE International Conference on Semantic Computing (ICSC 2009)*, pages 342 – 348. IEEE Computer Society, 2009.

[5] G. Stevenson and S. Dobson. Sapphire: Generating Java Runtime Artefacts from OWL Ontologies. In C. Salinesi and O. Pastor, editors, *Advanced Information Systems Engineering Workshops*, volume 83, pages 425–436. Springer Berlin Heidelberg, 2011.

[6] M. Uschold. Ontology-Driven Information Systems: Past, Present and Future. In *Proceedings of the Fifth International Conference on Formal Ontology in Information Systems (FOIS 2008)*, pages 3 – 18, Amsterdam, The Netherlands, 2008. IOS Press.

[7] M. Völkel. Writing the Semantic Web with Java. Technical report, DERI Galway, Galway, 2005.

[8] D. Zeginis, F. D'Andria, S. Bocconi, J. Gorronogoitia Cruz, O. Collell Martin, P. Gouvas, G. Ledakis, and K. a. Tarabanis. A user-centric multi-PaaS application management solution for hybrid multi-Cloud scenarios. *Scalable Computing: Practice and Experience*, 14(1):17–32, Apr. 2013.