Ontology Management in a Service-oriented Architecture

Architecture of an Knowledge Base Access Service

Jürgen Moßgraber
Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB
Karlsruhe, Germany
e-mail: juergen.mossgraber@iosb.fraunhofer.de

Marco Rospocher
Fondazione Bruno Kessler - Centro per la Ricerca Scientifica e Tecnologica (FBK- irst)
Trento, Italy
e-mail: rospocher@fbk.eu

***Abstract*—An increasing number of information systems integrate semantic data stores for managing ontologies. To access these knowledge bases most of the available implementations provide application programming interfaces (APIs). The implementations of these APIs normally do not support any kind of network protocol or service interface. This works fine as long as a monolithic system is developed. If the need arises to integrate such a knowledge base into a service-oriented architecture a different approach is needed. In this paper we propose an architecture to address this issue. A first demonstrator was fully implemented in the European project PESCaDO. Several services access and work on a central knowledge base access service which supports multi-threaded access for parallel instantiated ontologies.**

Keywords-service-oriented architecture (SOA); ontology management; Web Ontology Language (OWL); knowledge base; application programming interface (API)

I. INTRODUCTION

The last decade has seen a growing adoption of information systems with a semantic-enabled repository, generally in the form of the so called *triple stores*. Usually, these systems rely on knowledge base access modules tightly integrated into the system. Indeed, these modules usually offer the opportunity to directly access and manipulate their content through application programming interfaces (APIs). However, this solution has some limitations if the information system is based on a service-oriented architecture (SOA) where the knowledge base access module would be a service as well. In fact, the implementations of these APIs normally do not support any kind of network protocol or service interface.

In this paper we propose an architecture supporting access and manipulation of the content of a knowledge base (KB) in service-oriented platforms. The architecture is based on two main pillars:

1. an interface to the KB which combines service interface features with API ones;
2. *session-based ontology instantiation*, i.e. the idea to instantiate a KB for each user session considered at the level of the information system, containing only the instantiated content relevant for the considered session. This allows one to limit the size of the KB on which to perform OWL 2 reasoning, thus enabling more expressive reasoning capabilities (e.g., OWL 2 DL) than those offered by standard triple stores (usually, OWL 2 RL or QL).

A first demonstrator of the proposed architecture was fully implemented in the European project PESCaDO [8]. Several services access and work on a central KB access service which supports multi-threaded access for parallel instantiated ontologies.

The paper is organized as follow: Section II presents some related works. Section III describes the issues behind accessing ontologies in a SOA, while in Section IV we describe the idea of session-based ontology instantiation. Section V presents a general architecture of a KB access service to be used in a SOA, and Section VI describes the deployment of this architecture in the context of a service-oriented system for personalized environmental decision support within the PESCaDO project. We then conclude with some final remarks.

Note: in the following text we use the term *knowledge base* to refer to one or more ontology instantiations persisted in a (triple) store or kept in-memory as well. The KB is intended to be manipulated (i.e. dynamically instantiated) and not static.

II. RELATED WORK

Typically, information systems based on a KB rely on established and efficient triple store repositories. Examples of popular triple stores are: Sesame [2], OWLIM [1], Jena [3], and Virtuoso [4]. Although these stores are able to handle huge amounts of data (e.g., trillion of triples), their reasoning capabilities are usually limited to RDF/RDF(S) or to efficiently tractable variants of OWL 2 (e.g., OWL 2 RL or QL).

Ideas for accessing ontologies in a SOA were designed and implemented in the European projects ORCHESTRA [5] and SANY [6]. In the context of both projects a repository for ontologies was needed. The repository was put in place as a service itself with operations of a broad granularity. This means that the service interface provided operations to read (A-Box and T-Box separately), write, and delete whole ontologies.

If some instances had to be added to the ontology, a rewrite of the whole ontology was necessary. Queries on the content of the ontology were also not supported.

III. Service Interface vs. API

How can several services (distributed in a network) access a KB in parallel as depicted in the following figure 1?

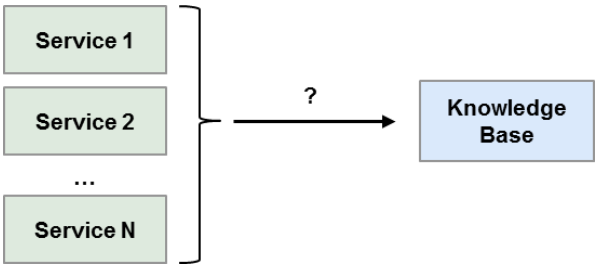

Figure 1. How do services access a knowledge base?

To access and manipulate ontologies there are several APIs available. Most of them are for the programming language Java and feature a fine grained and therefore powerful interface. The most popular is still Jena but it currently does not support OWL 2. For full OWL 2 support there is the OWL API [9]. The OWL API is a Java API and reference implementation for creating, manipulating and serializing OWL Ontologies. Starting with version 3 it supports the full OWL 2 standard. It is currently evolving into a kind of *de facto* standard; for example it is used by the most widespread ontology editor Protégé 4 (see http://protege.stanford.edu/).

All of these APIs are designed for direct integration into a system and do not provide a network abstraction.

A brute force approach to design a service interface could be to create a 1:1 mapping from e.g., the OWL API. There are several drawbacks with such an attempt:

- In a SOA the communication should be kept minimal because it is time consuming and slows down the overall system. For example if one would add some instances to the ontology, many operations must be called and therefore much communication takes place.
- The available APIs are mostly designed object-oriented which often means that if you call a function you get back an object on which you call further functions (e.g., get a concept and then get its object properties). In contrast, service interfaces tend to be not object-oriented and have a flat set of operations. However it could be done by using complex data structures which would be complicated to implement and difficult to use (and again causes a lot of communication).

Instead of using an API there is the possibility to use query languages (QLs) for Semantic Web ontologies. An example of such a language is SPARQL [10]. With its extension *SPARQL Update* it can cover most API functionality. A drawback is that modifications to the ontology which are fairly simple via an API tend to get quite complex if formulated in SPARQL. SPARQL acts on the same level as SQL for relational databases and has as well the possibility to be used via a network with the *Remote SPARQL* specification (compares to JDBC or ODBC for SQL).

In our suggested system architecture (presented in section V below) we tried to combine both approaches of API and query language to limit the drawbacks.

IV. Session based Ontology instantiation

A further aspect to be considered when developing an information system exploiting an ontology-based KB is the tradeoff between size and reasoning capabilities supported, especially in real time and interactive systems: reasoning performance falls with the size of the KB. Typically, this aspect is tackled by supporting restricted profiles of the used ontology language (e.g., OWL 2 QL), thus limiting the reasoning capabilities of the system.

However, there are situations in which alternative strategies could also be considered. Indeed, in various applications, it may happen that the inferred content to be produced via reasoning may be limited to a specific user session or to limited content of the KB. A typical example is a decision support system which is regularly (e.g., hourly or every few minutes) fed with new data (e.g., economic data, stock market data, environmental data, news): only a fraction of the huge quantity of data collected is usually relevant for a single user decision support request. In such situations, one can think of extracting from the general data repository only the data which are relevant for the considered user session, to instantiate a user session specific KB with the relevant data, and to perform reasoning on this smaller KB. This approach, which we refer to as *session-based ontology instantiation*, allows an efficient exploitation of the reasoning capabilities offered by more expressive variants of the ontology language used (e.g., OWL 2 DL), due to the limited size of the session specific KB.

A further benefit of the session-based ontology instantiation approach is that, due to its limited size, it enables to work with an in-memory implementation of the session specific KB instead of a persistent one, thus favoring a further improvement of the performance of the system (see Section VI for some details on the performance improvement we observed in a concrete situation).

Another positive side effect of this concept is that we could work around a limitation of most reasoners. The reasoners tested during the implementation of the first non-session-based approach namely HermiT (http://hermit-reasoner.com/) and Pellet (http://clarkparsia.com/pellet) turned out to not (fully) support multi-threaded access. Because of this a synchronized handling of queries was necessary with a huge drawback on performance.

An evident disadvantage of the session-based ontology instantiation approach is that the ontology has to be created and instantiated each time a user session starts, thus producing some computational overhead to be considered.

However, the impact of this can be reduced by setting up an *ontology pooling system*, where multiple ontology pools are in place, each one containing a single ontology. As soon as a new session starts, the first unused ontology is allocated to that session, and disposed at its end, so that the ontology pool becomes available for a new starting session. No synchronization or reconciliation is needed as the session ontology is in place (and used) only for a single user session.

To successfully apply the session-based ontology instantiation approach an important precondition must be met: the data relevant for the session must be known and much less than all available data. In PESCaDO we modeled in the ontology the data types required for a specific request (user profile/type of request), whereby the amount of data is limited by a spatial and temporal selection.

V. The Knowledge Base Access Service

The Knowledge Base Access Service (KBAS) should fulfill the following functional requirements:

- Storing, updating or deleting available ontology modules;
- Retrieving (partially or fully) a stored ontology module;
- Getting high-level information about some ontology modules, such as the list of concepts, or the list of supported properties for a given concept;
- Inserting actual data in the KB;
- Querying one or more ontology modules.

The difficult part was to find a good balance between fine grained API functionality and a very broad one as in the SANY project which allowed only the storage and retrieval of a full ontology. Furthermore, a query language like SPARQL should be supported as well because many users of the service are already familiar with it.

Therefore we decided to start with SPARQL (and SPARQL Update as well). After that we added operations which were either hard to formulate in a SPARQL query or took a lot of processing time. All operations can work on the main (persisted) ontology or one of the session-based ontology instances kept in memory.

The most relevant query operations of the KBAS interface are:

- *queryOntology*: Submits a query to the ontologies stored in the KB (can work on the main ontology and session-based ontologies). The query has to be formulated in a query language (e.g., SPARQL) compatible with the knowledge representation model used by the KB (e.g., RDF/OWL).
- *getRestrictionsFor*: Get all restrictions between two classes and a specific property. In PESCaDO these restrictions were used to describe the input parameters required for specific user queries and allowed to dynamically generate the user interface. This gives us the opportunity to add new query types without having to adjust the user interface at a later stage. Retrieving this information via SPARQL requires a lot of SPARQL calls and additional post-processing of the responses.

For manipulating the KB there are the operations:

- *setOntology*: Creates a new session-ontology instance based on a configurable base ontology. It returns an identifier for further operation calls.
- *deleteOntology*: Removes an existing session-ontology instance from the KB.
- *addABoxStatements*: This function allows data to be inserted into the KB. It can work on the main ontology and session-ontology instances. It is easier to use than SPARQL Update.
- *removeABoxStatements*: This function allows factual knowledge (data) to be deleted from the KB (it can also work on the main ontology and session-ontology instances). It is easier to use than SPARQL Update.

In PESCaDO we noticed that the expressive power of OWL 2 and DL reasoning was not enough to implement the Decision Support required in the project. Indeed, one of the requirements of the Decision Support module was to be able to instantiate new individuals in the ontology as a direct result of reasoning, something unfeasible with OWL DL reasoning. To deal with this we added support to use rules which is handled by the following operation:

- *applyRules*: Applies an arbitrary set of rules to a session-ontology instance. The newly reasoned statements are automatically added to the ontology.

VI. Information Technology Implemented for the PESCaDO Demonstrator

A. The PESCaDO project

Citizens are increasingly aware of the influence of environmental and meteorological conditions on the quality of their life. The consequence of this awareness is the demand for personalized environmental information, i.e., information that is tailored to citizens' specific context and background. In PESCaDO (Personalized Environmental Service Configuration and Delivery Orchestration) an environmental information system that addresses this demand in its full complexity is being developed. More precisely, it aims to develop a system supporting the user in questions related to environmental conditions by searching for reliable data on the web. These data are processed and converted into knowledge stored in an ontology-based KB, from which information relevant to the specific user is deduced and communicated in the language of his / her preference.

B. *Architecture of the KBAS*

The internal architecture of the KBAS is depicted in Figure 2. The interface of the KBAS is defined in the Web Services Description Language (WSDL) [11]. The WSDL file is available from the public PESCaDO web server (http://pescado-project.eu/). We preferred WSDL over REST [13] because the strict interface typing makes integration in a distributed project easier.

As Java was set as the programming language we chose the JAX-WS framework (which is the reference implementation for building web services in Java; see http://jax-ws.java.net/) as the basis of the implementation.

To implement the *queryOntology* operation we chose SPARQL. A publicly available processor for SPARQL is ARQ which is a part of Jena.

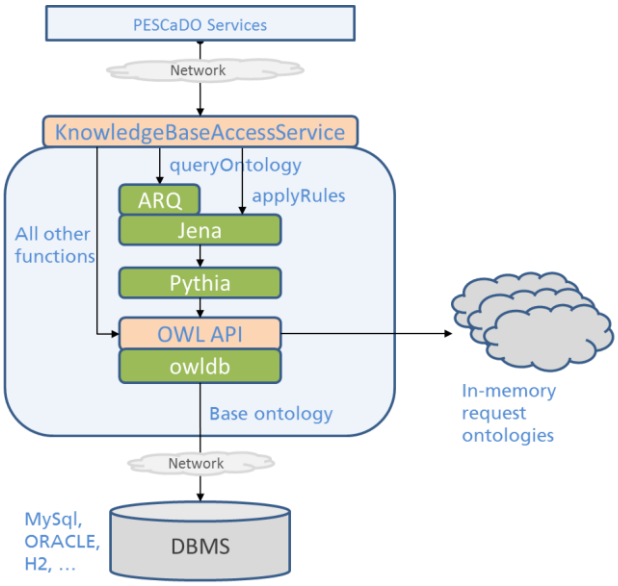

Figure 2. Architecture of the Knowledge Base Access Service

As we wanted to use the OWL API for full OWL 2 support (we recall that the standard Jena Ontology API supports OWL 1.0/1.1), there was the need for an adapter which could be plugged into the Jena framework. Fraunhofer IOSB has already implemented such an adaptor called *Pythia* (http://theseus-programm.de). Pythia provides a Jena *ontology model* by extending a Jena *GraphBase* class. The calls to the *graphBaseFind* function of the class are converted into the corresponding calls of the OWL API.

All other operations of the KBAS are directly realized with the functionality of the OWL API. Indeed, this was also needed to access the ontology of the KB at a conceptual level, such as for instance to dynamically access restrictions imposed on classes. This is problematic (and in some cases even impossible) with SPARQL, as a *static* graph pattern to be matched has to be encoded in the query.

The reference implementation delivered with the OWL API is an in-memory only implementation, which means that the data is lost after a restart of the system. Also, the ontology size is limited to the main memory of the computer. To overcome this problem, a persistence solution called *owldb* [7] was built in parallel to Pythia (available under the LGPL from http://owldb.sourceforge.net/).

For the persistence of OWL ontologies, owldb uses a relational database. To store the OWL API entities and axioms, an object relational approach is used. This component guarantees that only the entities currently in use are kept in memory. This facilitates the use of even large ontologies. Once stored, the axioms and entities of an ontology can be retrieved directly from the database and the ontology can be directly manipulated within the database. Furthermore, it is possible to perform format conversions from all ontology formats currently supported by the OWL API to the new database format and vice versa.

C. *Support for Rules in the KBAS*

We researched how to use rule sets on top of the OWL ontology in order to achieve a richer expressiveness. Although current state-of-the-art OWL DL reasoners support the definition and firing of rules represented in SWRL [12], its expressive power and its DL-safeness was not sufficient to encode the rules envisaged in PESCaDO. For instance, PESCaDO rules required the ability to instantiate new individuals in the ontology as a direct result of the inference phase, something not supported in SWRL and by DL reasoners (would break decidability). Therefore, for the realization of the operation *applyRules* (and not to start from scratch) we looked into extending Pythia to additionally adapt Jena rules to the OWL API.

Jena already includes a general purpose rule-based reasoner which is used to implement both the RDFS and OWL reasoners, but is also available for general use. This reasoner supports rule-based inference over RDF graphs and provides forward chaining, backward chaining and a hybrid execution model.

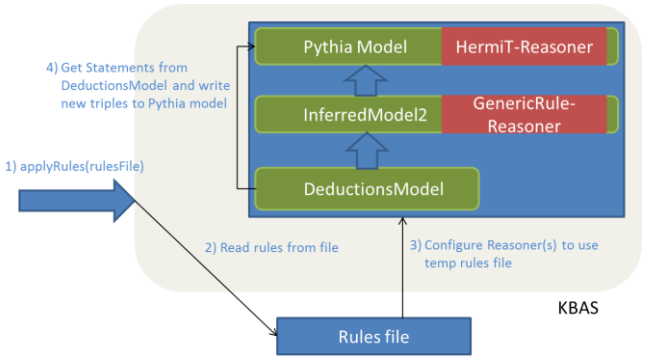

Figure 3. Architecture of the Rules implementation of the KBAS

The goal was reached by wrapping the Jena Model provided by Pythia into a *InferredModel* which uses the *GenericRuleReasoner* and a *DeductionsModel* (see figure 3). We conclude this section by reporting on an example of a rule defined in PESCaDO, according to the Jena rule syntax:

```
[ruleAbundantPollen:
    (?request rdf:type AnyHealthIssue)
    (?request hasUser ?user)
```

```
      (?user isSensitiveTo birchPollen)
      (?pollen rdf:type PollenDataType)
      (?request hasGeoArea ?geoArea)
      (?geoArea hasAggregatedData ?dataAggregated)
      (?dataAggregated hasEnvDataType birchPollen)
      (?dataAggregated hasAggregationType max)
      (?dataAggregated hasRating ?rating)
      (?rating hasRatingValue abundantPollen)
  makeTemp(?rec)
->
  (?rec rdf:type Recommendation)
  (?rec hasRecommendationType:rec_abundantPollen)]
```

This rule is used to generate a recommendation due to abundant birch pollen concentration in case the user who asks for decision support is sensitive to this pollen.

D. Facilitation in PESCaDO

The main ontology instance of the system contains the basic ontology schema and data which need to be persisted such as for example user profiles. This main ontology instance is stored in a database via owldb. Based on this ontology the KBAS manages specific ontology instances for each user session of the PESCaDO system. Such a session-ontology instance is created as a clone of the base ontology and kept in memory. All data required to answer a user's query is added to this instance (e.g., weather, pollen and air quality information in the region and time of the user's interest). This data is only required for a single query and there is no need to make it persistent. Therefore we do not have the need to synchronize changes made in the session-ontology back to the base ontology.

In the first version of our demonstrator all queries were working on the main ontology instance which led to a processing time in the range of 2 to 4 minutes which is absolutely inacceptable for a real world information system. By switching to the per session-ontology instance approach the time was reduced to the range of 10 to 30 seconds (on a current computer with Intel i5 Processor and 4GB of main memory). More detailed measurements will be taken in the next step of the project. About 95% of that remaining time is spent in data retrieval, data fusion, and the final text generation of the answer for the user and is out of the scope of the KBAS.

VII. SUMMARY AND FUTURE WORK

In this paper we discussed the differences between service interfaces and APIs for accessing an ontological KB. After that we presented the KBAS interface which tries to combine the advantages and limit the drawbacks of fine vs. broad grained interfaces.

The ideas explained could be evaluated in the PESCaDO project and shown to function. Nevertheless, more work needs to be done to fine tune the concepts. The *ontology pooling system* mentioned in section IV is not yet in place. In the final version of the KBAS implementation we want to improve the handling of session-ontology instances by implementing this idea which should work similarly to a database connection pool. This will completely remove the time for providing a newly initialized session-ontology instance. After that task is finished we want to conduct some final performance measurements.

Another drawback is that the in-memory implementation combining Jena, Pythia and the OWL API shows a slower performance than the Jena only in-memory implementation. We need to investigate if the way how Pythia plugs into Jena is not optimal and/or if the OWL API in-memory implementation is weaker than the Jena one. If this concept cannot be improved, alternatives such as OWLIM will be considered as well (with the disadvantage of losing some OWL 2 features).

ACKNOWLEDGMENT

Running from January 2010 to December 2012, PESCaDO is partially funded by the European Commission in its 7th Framework Programme under the contract number ICT-259486.

Pythia and owldb were developed in THESEUS, a research program initiated by the German Federal Ministry of Economy and Technology (BMWi).

REFERENCES

[1] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, R. Velkov, "OWLIM: A family of scalable semantic repositories," Semantic Web, 2011

[2] J. Broekstra, A. Kampman, F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema," International Semantic Web Conference, 2002, pp. 54-68.

[3] J. J. Caroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, K. Wilkinson, "Jena: implementing the semantic web recommendations," Digital Media, 2004, pp. 74-83.

[4] O. Erling, I. Mikhailov, "Virtuoso: RDF Support in a Native RDBMS," Semantic Web Information Management, 2009, pp. 501-519.

[5] T. Usländer (ed.), "Reference Model for the ORCHESTRA Architecture (RM-OA) V2 (Rev 2.1)," Open Geospatial Consortium Inc., 2007.

[6] T. Usländer, "Specification of the sensor service architecture Version 3.0 (Rev. 3.1)," OGC Discussion Paper 09-132r1, Deliverable D2.3.4 of the European Integrated Project SANY, FP6-IST-033564, 2009.

[7] J. Henss, J. Kleb and S. Grimm, "A database backend for OWL," Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED), 2009.

[8] L. Wanner, et al., "Building an Environmental Information System for Personalized Content Delivery," Environmental Software Systems. Frameworks of eEnvironment. Volume 359/2011, 2011, pp. 169-176.

[9] M. Horridge, S. Bechhofer, "The OWL API: A Java API for working with OWL 2 ontologies," OWLED, volume 529, Proceedings of CEUR Workshop, 2008.

[10] E. Prud'hommeaux, A. Seaborne, "Sparql query language for rdf," W3C Working Draft, http://www.w3.org/TR/rdf-sparql-query/, 2006.

[11] E. Christensen, F. Curbera, G. Meredith and S. Weerawarana, "Web Services Description Language (WSDL) 1.1," World Wide Web Consortium (W3C) note, 2001.

[12] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML," W3C Member Submission, 2004.

[13] S. Vinoski, "REST Eye for the SOA Guy," IEEE Internet Computing, vol. 11, no. 1, pp. 82-84, Jan.-Feb. 2007, doi:10.1109/MIC.2007.