Developing Distributed Reasoning-Based Applications for the Semantic Web

Dimitrios Koutsomitropoulos   Georgia Solomou   Tzanetos Pomonis
Panagiotis Aggelopoulos   Theodore Papatheodorou, *member IEEE*
High Performance Information Systems Laboratory
Computer Engineering and Informatics Dpt., University of Patras
Patras-Rio, Greece
{kotsomit, solomou, aggelopp, pomonis, ptheodor}@ceid.upatras.gr

Abstract—**In order for Semantic Web applications to be successful a key component should be their ability to take advantage of rich content descriptions in meaningful ways. Reasoning consists a key part in this process and it consequently appears at the core of the Semantic Web architecture stack. From a practical point of view however, it is not always clear how applications may take advantage of the knowledge discovery capabilities that reasoning can offer to the Semantic Web. In this paper we present and survey current methods that can be used to integrate inference-based services with such applications. We argue that an important decision is to have reasoning tasks logically and physically distributed. To this end, we discuss relevant protocols and languages such as DIG and SPARQL and give an overview of our Knowledge Discovery Interface. Further, we describe the lessons-learned from remotely invoking reasoning services through the OWL API.**

Reasoning; Semantic Web; SPARQL; DIG; OWL

I. INTRODUCTION

Understanding the merit that the Semantic Web contributes to the current Web turns out to be not always self-evident. The proliferation of applications that claim to be conformant to Semantic Web standards may reveal that these technologies may have become popular in web-literate circles; still Tim Berners-Lee's vision has not yet proven itself to provide tangible added-value to the everyday browsing experience. What seems to be missing for Semantic Web to be more than a set of well-specified logical formalisms is to share the understanding that automated reasoning is the key to unlock and exploit its information.

The late success of Description Logics (DL) seems to owe much to their close relationship with ontology languages such as OWL and OWL 2 [5]. Relevant DL-based reasoning systems, such as RACER, FaCT++ and Pellet have become popular accordingly. Nevertheless, it is the use of these systems within applications that can add an "AI flavor" to the current Web and make the Semantic Web deliver.

Even as these tools exist and evolve, we have not yet experienced common-scale usage of the Semantic Web and the inference capabilities it inherently bears with. A possible reason is that there is not a standardized way to take advantage of these capabilities. The dominance of the relational data model in data base systems and warehouses is tightly related to the fact that it comes coupled with a simple, clutter-free and standardized querying protocol, SQL. Despite relevant efforts, (we will see that) such a facility has yet to come for ontology models.

Additionally, the usage of reasoning services seems to imply applications distributed in nature. Reasoning problems in OWL DL are solved optimally in 2-NEXP time [18], whilst for OWL 2 it is 3-NEXP [10]. One cannot expect from the occasional user to bear with such long time intervals during his browsing and querying, nor is there any room for improvement, at least as far as DLs are concerned. Thus, a way must be found to alleviate the end-user machine from this task and delegate it to a more powerful, back-end system (usually a server), thus creating 3-tier, distributed applications.

The purpose of this paper is exactly to show how one can commence on developing applications of this type for the Semantic Web. First, we review the ways in which existing reasoning software may be utilized in the framework of such applications and how these reasoners can be interfaced. At the core of developing distributed, reasoning-based applications, there lie:

- The existence of protocols that would allow efficient communication/interchange with reasoners and

- The specification of corresponding querying languages, allowing also for *conjunctive queries*, resembling the convenience of relational DB systems.

Therefore we survey options in these two categories, putting emphasis on DIG and SPARQL. Then, we discuss the value of a decentralized architecture for Semantic Web applications as the scene is currently set and present the *Knowledge Discovery Interface* (KDI) as one of the early exemplars for such applications. Finally, we refer to our experience in developing a middleware that allows for remote utilization of reasoning mechanisms on top of the OWL API and outline our results and conclusions.

II. Querying Languages for the Semantic Web

Knowledge discovery does not coincide with traditional data retrieval, at least not in the way the latter is standardized by query languages and protocols – e.g. in a relational database. However, it too, assumes the formulation and composition of some kind of intelligent queries, the answers to which will compose the requested knowledge. For example a typical query in SQL or XQuery is not suitable for retrieving answers from an inference engine. Neither however do the latest proposals on Semantic Web query languages, like SPARQL, seem to be adequate for modeling such declarative queries since they cannot capture the full expressivity of OWL.

In this section we therefore discuss some early protocols for querying web ontology documents. Then we focus on SPARQL, as the *de facto* querying language for the Semantic Web and look in to the efforts for extending this language to query OWL-specific documents.

A. Early Efforts

Along with the first efforts for standardizing ontology languages for the Web, like DAML+OIL and then OWL, there came proposals for taking advantage of these languages, through querying mechanisms. This led to the proposal in the literature of the DAML Query Language (DQL) and OWL-QL [3], [4] as protocols responsible for exchanging information and queries/answers against ontology documents. However, these very promising specifications did not seem to find their way in any substantial implementation, let alone production, possibly for the reasons identified in [11].

On the other hand, it took some time for standardizing a query language for the Semantic Web – based however upon RDF in general and not OWL: SPARQL [15].

B. SPARQL and SPARQL/OWL

SPARQL came out of a scientific competition between various candidates for standardization (see for example the comparison between SRARQL and RDQL [9]) and emerged to a definite W3C standard through a process not totally smooth (see the rollback of the specification from candidate recommendation status to working draft[1]).

SPARQL's semantics are based on the notion of RDF graphs and thus they do not correspond directly to OWL. Therefore, although SPARQL achieves to define a standard way for composing and submitting conjunctive queries to RDF documents, it lets out of scope the inference capabilities, taxonomic queries and the particulars of the essential ontology languages OWL and OWL 2.

SPARQL-DL has been introduced in [17] as a subset of SPARQL that allows, under certain conditions, the expression of conjunctive queries and their combination with taxonomic ones, for the ontology language OWL DL (for example, see Table I). SPARQL-DL is implemented and supported, with various optimizations, by the recent versions of the Pellet reasoner. Also, it can be evenly extended towards OWL 2, by adding corresponding query atoms like `Reflexive(p)`, for

---

TABLE I.   Tim Query Examples in SPARQL-DL

Query Type	Example
ABox/TBox	`Type(?x,ex:Student),Type(?x,?c)` `SubClassOf(?c,ex:Employee)`
ABox/RBox	`ObjectProperty(?p),` `PropertyValue(ex:John,?p,?u)`

checking the reflexivity of a property p. In fact, the efforts for standardizing a query language for OWL and OWL 2 are now grouped under the name *SPARQL/OWL*[2]. Finally, nRQL is a fairly expressive query language supported by RACER [6] tailored specifically for conjunctive queries that pertain to the ABox part of an ontology.

III. The DIG Interface

DIG is a common standard interface for DL reasoners[3]. Through the provision of a basic API, DIG allows the utilization of DL systems by a variety of tools in a standardised way. DIG's underlying protocol, responsible for accomplishing communication with a reasoner, is HTTP. Request messages are composed using a proprietary XML syntax. The messages that server and client exchange are divided in *Tells* and *Asks* and allow the client for the following actions: query for the server's reasoning capabilities, transfer an ontology to the server, perform certain ontology manipulations and ask standard DL queries about the given ontology, e.g., concept subsumption, satisfiability, equivalence, etc.

Although DIG is biased towards DL systems only, it consists in fact a robust protocol for communicating with reasoners and thus can be used for the remote delivery of reasoning-based services.

A. DIG 1.x and DIG 2.0

The introduction of the DIG Interface, DIG 1.0, took place in October 2002 and its updated version, DIG 1.1, came soon after, on February 2003. DIG 1.1 was quickly adopted by reasoning engines, ontology editors and many other Semantic Web applications. But despite its popularity, it was confronted with a number of problems, such as limitations in the supported language fragment (e.g. no support for datatypes, poor fit between DIG's notion of relations and OWL properties), a lack of elementary queries, and the absence of a mechanism to extend the protocol (a short report on these issues is made in [2]).

Next version, DIG 1.2, was introduced as an extension to the original DIG 1.1 interface regarding the provision of an ABox query language. In essence, version 1.2 provided the ability to pose unions of conjunctive queries to reasoners. DIG 1.2 didn't fix semantics for the queries, which allowed implementors to define their own. Version 1.2 was only implemented by the reasoners RacerPro and QuOnto.

Some of the DIG's 1.1 flaws were addressed in the subsequent draft specification of DIG 2.0 [19]. As DIG 1.1 was

---

unable to capture general OWL DL ontologies, another concern of this update was to upgrade the concept language up to OWL DL level. In addition, support for alternative DL dialects (e.g. EL++) became possible. With an ultimate goal to support the then-upcoming OWL 2, DIG 2.0 provided for some extended functionality such as support for qualified cardinality restrictions and role forming operators.

B. Moving from DIG to OWLlink

DIG's current version is called OWLlink [13]. Its basic feature is that it is based on the most recent OWL 2 specification for the primitives of the modeling language. So OWLlink is regarded as DIG's successor – upgraded in many levels – that provides an extensible protocol for communication with OWL reasoning systems. It is actually a more refined version of DIG – as far as query and language expressivity is concerned – but it is not backward compatible with it. OWLlink supports many features of OWL 2, like for example the notions of punning and structural equivalence. However, it does not support any part of OWL 2 beyond the level of axioms, like for example, OWL imports.

The communication between OWLlink-compliant clients and servers is implemented through the exchange of request and response messages. By this process client applications become able to configure reasoners, to access reasoning services and to transmit ontologies. A basic functionality provided by OWLlink is the ability to either add axioms to a KB (*Tell* requests) or to retrieve information about a KB (*Ask* requests) (Table II). Ask requests only cover very common queries with respect to the given and inferred axioms of the KB. More complex queries, notably including conjunctive ones, remain to be specified by the *OWLlink Query Extension*. The latter, however, is still in draft stage and a bit outdated, since it refers to the preceding DIG 2.0 proposal[4].

C. DIG Compliant Tools and Applications

Apart from those DL reasoners that support the DIG interface (e.g. CEL, FaCT++, Racer, Pellet and KAON2) ontology editors, as well as many other applications and middleware, make use of DIG for reasoning over OWL

ontologies (e.g. Instance Store, OntoXpl and Jena framework). As far as OWLlink is concerned, although still in draft stage, it is supported or it is about to be done so by widely used reasoning systems like RacerPro, Pellet and FaCT++. RacerPro had initially implemented OWLlink as part of its 1.9.3 version and kept supporting it in its current version,

TABLE II.   EXAMPLE OF OWLlink *Ask* REQUEST AND RESPONSE

REQUEST	RESPONSE

`<GetSubClasses ol:kb="`*aKB*`"`
`ol:direct="false">`
`<ox:OWLClass`
`ox:URI="`*aClass*`"/>`
`</GetSubClasses>`
	`<SetOfClassSynsets>`
	`<ClassSynset>`
	`<ox:OWLClass`
	`ox:URI="&owl;Nothing"/>`
	`<ox:OWLClass`
	`ox:URI="`*aClass*`"/>`
	`</ClassSynset>`
	`</SetOfClassSynsets>`

---

[4] http://www.sts.tu-harburg.de/~al.kaplunova/dig-query-
interface.html

RacerPro 2.0. QuOnto is another reasoner which is being aligned with OWLlink, through its OBDA extension [16]. What is more, Protégé and OntoTrack, as stated in [13], have already committed to implement OWLlink, a fact that also becomes true for various other developers of ontology editing and browsing tools.

IV.   THE VALUE OF A 3-TIER ARCHITECTURE: THE KDI

In this section we see why it is important to stick to a distributed architecture for Semantic Web applications, even though DIG could not always be used towards this goal. First we explain that we cannot always count on DIG as a reasoner's interface and this comes at a price. Then we briefly present a concrete reasoning-based application that has been built upon these ideas.

A. DIG vs Direct In-Memory Deployment

FaCT++ and Pellet are currently the only two DL-based engines that appear to fully support the decidable subset of OWL and OWL 2 [12]. However they only support DIG 1.1, which is insufficient for full OWL DL support (see section 3.1), a fact that has mostly driven the upcoming OWLlink specification.

DIG 1.1 communication takes place over HTTP and there is no other TCP/IP-like connectivity support; rather, for these reasoners to be utilized by a tool or application, one may use a programmatic API (e.g. Jena or the OWL API) that interfaces these reasoners as direct in-memory implementations [8]. This approach may have the advantage of reducing the message-passing load of the DIG protocol, but surely it is insufficient for developing truly decentralized Web applications and services for the Semantic Web. As DIG 2.0 specification that would solve the aforementioned problems has been for a long time in flux, these reasoners cannot be used in developing a distributed web service for Semantic Web knowledge discovery that would fully support OWL DL or OWL 2.

Even from a performance point of view, the physical distribution to different computational systems would rather accelerate the application as a whole, especially in the case of "hard" (very expressive) ontologies: there would be a point when reasoning times outmatch the communication overhead (recall the multiple exponential complexities, both in time and space) and the execution of the reasoning algorithms to a powerful back-end system appears tempting (see also section 5).

B. The KDI

The KDI [11] is a web application, providing intelligent query submission services on Web ontology documents. The interface design follows the traditional 3-tier model, with an important variation: Where a database server would be typically used, we now use a knowledge base management system, since a traditional DBMS lacks the necessary features and functions for managing and utilizing ontological knowledge (Fig. 1). Note that each of the three levels may be physically located on different computer systems.
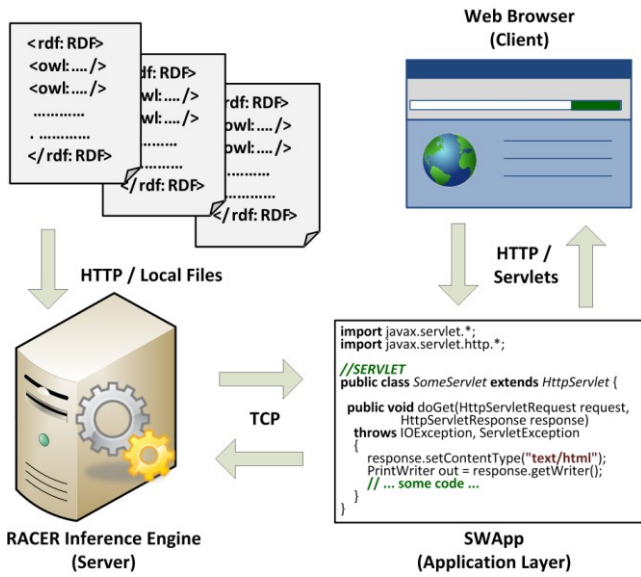
Figure 1. The architecture of the KDI.

KDI interacts with RACER to conduct inferences: despite its limited expressivity, RACER constitutes a system that decentralized applications can be built around. Indeed, from a design point of view, KDI aims to be decentralized in at least two ways:

i. The Semantic Web knowledge bases that data is extracted from, can be both logically and physically distributed (in the case of OWL, this can be accommodated by the `owl:import` directive).

ii. The tiers of the application can also be distributed both at the logical and at the physical level: the front-end layer (browser), the application logic layer (servlets, javabeans, tomcat) and the knowledge-base management layer (inference engine).

Such a truly decentralized architecture, in accordance to the traditional 3-tier paradigm, is not yet possible with the majority of the current state-of-the-art and highly expressive inference engines, due to limitations of their interface capabilities, as described earlier. On the other hand, such an approach can have a more substantial contribution to the utilization of semantic information by users and applications by eliciting more obvious value from ontological data [7]. In addition, it can be extended to support next generation Web 3.0 applications, as shown in [14].

V. A MIDDLEWARE FOR DECENTRALIZED REASONING

Although the KDI may serve as a useful paradigm for distributed, reasoning-based applications, it is proprietarily designed to interact only with RACER through the custom JRacer API. OWL API[5] on the other hand is a well-known Java programmatic abstraction for manipulating ontologies. It is widely used for developing ontology-based applications and allows communication with various inference engines,

deployed however only in-memory (i.e. there is no remote communication).

In order not to restrain an application to the particulars and/or limitations of any single inference engine and to allow for decentralized architecture at the same time, we have designed and developed a variant of OWL API that, in addition to its original advantages, can also support remote communication. In the following we briefly introduce this effort and check how it may contribute to the decentralized reasoning problem.

A. Description

Our goal is to have one main server that hosts the inference engine, which would be able to provide reasoning services to a number of client applications over the TCP/IP protocol. Client applications will be able to make queries and receive responses from the knowledge base, as by using the original OWL API.

To this extent, we developed a client-server interface in Java, using "remote objects". The reasoner is hosted in a server along with the OWL API. Our "server" application listens to a specific port, waiting for connections. Our "client" application, a variant of OWL API, is physically located on a different computer system, and in order to use the reasoner, it has to connect to the server. Upon connection, the server application initializes a reasoner object, reads the parameters from the client application, calls the corresponding reasoner method, and sends back the results. The whole client-server communication and message transaction, takes place over the TCP/IP protocol.

On the client side, the idea is to override the OWL API's `Reasoner` interface with a new class (`myReasoner`) that would expose all the reasoner-related methods defined in the original OWL API interfaces. These methods are replicate copies from the originals, aiming to provide an interface that can be used instead of the original OWL API, with minimal code modifications. Thus, an original OWL API application just needs to import our `myReasoner` class, instead of the original one.

On the server side, any OWL API compliant reasoner can be interfaced, through a `myServer` class which, upon initialization, creates a TCP/IP socket on a desired port, and waits for client connections. Remote objects are then being exchanged using Java Serialization.

As a next step, in order to further extend the distributed characteristics of our middleware, we are currently developing another version of our "server" application, which makes use of Java threads. In this multi-threaded version, each client connected to our server is virtually served by a different instance of the `Reasoner` class, resulting in parallel processing of the clients requests, instead of a serial client-after-client way. The only catch in the multi-threaded approach is that it cannot function with the FaCT++ reasoner, because of the JNI which does not support multi-threading.

The application has been tested with both inference engines known to support the OWL API, i.e. FaCT++ and Pellet.

---

[5] http://owlapi.sourceforge.net/

TABLE III.   TIME MEASUREMENTS FOR VARIOUS REASONING TASKS WITH AND WITHOUT REMOTE REASONER COMMUNICATION

	local, pizza	client-server, pizza	local, Finance	client-server, Finance
Total reasoning time	*239*	*1866*	*2470*	*7460*
loadOntologies	0	636	0	5760
classify	219	195	2417	613
isConsistent	0	45	0	85
getInconsistentClasses	0	300	0	204
getDescedantClasses	0	193	0	242

B. Results

In order to figure out our implementation's potential, we made some tests and time measurements based on the well known Example 8 of OWL API's documentation, using either the original pizza.owl ontology[6] or the Finance.owl one[7]. Both ontologies fall into the OWL DL dialect. We used two different computer systems: a client based on an Intel Atom 1.6 GHz processor with 2 GB of RAM, and a server based on an Intel Core 2 Duo 2.66 GHz with 2 GB of RAM.

During an indicative experiment using FaCT++, we measued the time (in *ms*) consumed by some methods as well as the total reasoning time in this example. We tested all the possible cases: *local installation* on the client computer (no remote communication) and *client-server installation*, against each ontology. These results are summarized in Table III.

Total reasoning time refers to the time the whole example program would take to terminate. Other metrics refer to various reasoner tasks and contribute to the total reasoning time, with classify() to be the hardest from a complexity point of view.

We observe that there is a significant time overhead on almost all occasions because of the remote communication over TCP/IP protocol and Java Serialization, as expected. On the other hand, having a "fast" server system dedicated for reasoning processes, results greatly in boosting processor-demanding methods (such as classify()), decreasing the effect of network delay. For example, the Finance ontology is classified almost 75% faster when delegating this task to a more powerful system, through our middleware.

An immediate conclusion is that the harder the ontology the better our middleware performs, as the relevant overhead lessens. And in real-life scenarios, ontologies tend also to evolve over time, thus classification results cannot always be cached. Therefore, the gains of using a dedicated reasoning server will ultimately overcome any communication overhead, making possible the use of "slow" computer systems for hosting the non-reasoner portions of any ontology-based semantic application.

In addition, the Java Serialization protocol seems to contribute greatly to this communication load blow-up, due to the complicate structure of OWL API objects that have to be transferred. To this end, we have also experimented with our implementation so as to communicate only the objects' URIs, instead of sending them selfsame through Java Serialization. In this way the client-server communication load can be minimized in many occasions.

VI.   CONCLUSIONS

The realization of the Semantic Web imperatively calls upon ways to transfer its benefits to the end users of the World Wide Web. For this to happen, the facility for reasoning-based, conjunctive queries should be made available by applications that would transparently integrate into the everyday browsing experience, in accordance to the Semantic Web vision [1].

However, we have shown that a serious gap still exists in the way the end-users or automated agents may interact with and take advantage of ontologies by composing queries and receiving results in a specific form. The standardization of a protocol for this process is currently a major research issue and is desirable in any production system. Both SPARQL/OWL and OWLlink specifications are considered to be two very promising steps towards this direction.

No matter how reasoning results may be wrapped and delivered, it has also become evident that applications must follow a decentralized scheme, in order to be both reliable and flexible at the same time. We have indicated exactly how this can become feasible and also efficient, in the long run.

REFERENCES

[1] T. Berners-Lee, J. Hendler and O. Lassila, "The Semantic Web", *Scientific American*, vol. 279, pp. 34-43, May 2001.

[2] I. Dickinson, "Implementation experience with the DIG 1.1 specification", Hewlett Packard, Digital Media Sys. Labs, Bristol, Tech. Rep. HPL-2004-85, May 2004.

[3] R. Fikes, P. Hayes and I. Horrocks, "DQL - A Query Language for the Semantic Web", Knowledge Systems Laboratory, Stanford, CA, KSL Tech. Rep. 02-05, Aug. 2002.

[4] R. Fikes, P. Hayes and I. Horrocks, "OWL-QL: A Language for Deductive Query Answering on the Semantic Web", Stanford University, Stanford, CA, KSL Tech. Rep. 03-14, 2003.

[5] B. C. Grau, et al., "OWL 2: The next step for OWL", *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, pp. 309-322, Nov. 2008

[6] V. Haarslev, R. Möller and M. Wessel, "Querying the Semantic Web with Racer + nRQL", in *Proc. of the KI-2004 Int. Workshop on Applications of Description Logics (ADL'04),* 2004.

[7] J. Hendler, "Web 3.0: Chicken Farms on the Semantic Web" *Computer ,* vol. 41, pp. 106-108, Jan. 2008.

[8] M. Horridge, S. Bechhofer and O. Noppens, "Igniting the OWL 1.1 Touch Paper: The OWL API", in *Proc. of the OWL Experiences and Directions Workshop (OWLED'07),* 2007.

---

[6] http://www.co-ode.org/ontologies/pizza/ 2007/02/12/pizza.owl

[7] http://www.fadyart.com/ontologies/data/Finance.owl

[9] K. Hutt. A Comparison of RDF Query Languages. (2005, Apr.). Rensselaer at Hartford Computer Science Seminar. [Online]. Available: http://www.rh.edu/~rhb/cs_seminar_2005/SessionE1/hutt.pdf

[10] Y. Kazakov, "SRIQ and SROIQ are Harder than SHOIQ", in *Proc. of the 21st Int. Workshop on Description Logics (DL-2008)*, 2008.

[11] D. Koutsomitropoulos, M. Fragakis and T. Papatheodorou, "Discovering Knowledge in Web Ontologies: A Methodology and Prototype Implementation", in *Proc. of SEMANTICS 2006 International Conference*, 2006, pp. 151-164.

[12] D. Koutsomitropoulos, D. Meidanis, A. Kandili and T. Papatheodorou, "OWL-based Knowledge Discovery Using Description Logics Reasoners", in *Proc. of 8th Int.Conference on Enterprise Information Systems (ICEIS 2006), SAIC track*, 2006, pp. 43-50.

[13] T. Liebig, et al., "OWLlink: DIG for OWL 2", in *Proc. of OWL Experiences and Directions (OWLED 2008)*, 2008.

[14] T. Pomonis, D. Koutsomitropoulos, S. Christodoulou and T. Papatheodorou, "Towards Web 3.0: A unifying architecture for next generation web applications", in *Handbook of Research on Web 2.0, 3.0 and X.0: Technology, Business and Social Applications*, In San Murugesan Ed., IGI Global, 2009.

[15] E. Prud'hommeaux and A. Seaborne. (2008, Jan.). SPARQL Query Language for RDF. W3C Recommendation. [Online]. Available: http://www.w3.org/TR/rdf-sparql-query/

[16] M. Rodriguez-Muro and D. Calvanese, "Towards an open framework for Ontology Based Data Access with Protégé and DIG 1.1", in *Proc. of OWL Experiences and Directions (OWLED 2008)*, 2008.

[17] E. Sirin and B. Parsia, "SPARQL-DL: SPARQL Query for OWL-DL", in *Proc. of OWL Experiences and Directions (OWLED 2007)*, 2007.

[18] S. Tobies, "Complexity results and practical algorithms for logics in Knowledge Representation", Ph.D. Thesis, RWTH-Aachen University, Germany, 2001.

[19] A. Y. Turhan, et al., "DIG 2.0 - Towards a Flexible Framework for Description Logics Reasoners", in *Proc. of OWL Experiences and Directions (OWLED 2006)*, 2006.