Comparing Semantic Registries – OWLJessKB and instanceStore

Simone A. Ludwig
Department of Computer Science
University of Saskatchewan
Saskatoon, Canada
ludwig@cs.usask.ca

Omer F. Rana
School of Computer Science
Cardiff University
Cardiff, UK
o.f.rana@cs.cf.ac.uk

ABSTRACT

Service discovery is a critical task in distributed computing architectures for finding a particular service instance. Semantic annotations of services help to enrich the service discovery process. Semantic registries are an important component for the discovery of services and they allow for semantic interoperability through ontology-based query formulation and dynamic mapping of terminologies between system domains. This paper evaluates two semantic registries – OWLJessKB implementation and instanceStore – to determine the suitability of these with regards to the performance of loading ontologies, the query response time and the overall scalability for use in mathematical services.

Categories and Subject Descriptors: I.2 [Computing Methodologies]: Artificial Intelligence.

General Terms: Measurements, Performance.

Keywords: semantics, service registries, service discovery, ontologies, mathematical services.

1. INTRODUCTION

As the Internet provides connectivity and information richness over great distances at any time, it has created a dynamic, open and convenient environment for social and business development. It not only provides the opportunity for new commercial endeavours utilising the Web, but also opens up new opportunities for the old, static, locally based businesses to adopt a new business paradigm and new organisational forms. The Internet has also opened up modes of interaction and dynamic organisational configurations that were previously inconceivable within a wide array of human and business activities. Yet, most of the interactions taking place are managed by humans. The computer science community envisions the future of the interaction on the Internet to be automated and performed by software agents that replace the role of humans. A service in a Service-Oriented Architecture (SOA) is a contractually defined behaviour that can be implemented and provided by a component (service provider) for use by another component (service consumer).

Services perform functions, which can be anything from simple requests to complicated business processes. They allow organisations to expose their core competencies programmatically over the Internet using standard languages and protocols, and be implemented via a self-describing interface based on open standards which permits software applications to interact with the service automatically. Service-oriented environments have special characteristics that distinguish them from other computing environments as follows. The environment is dynamic meaning that service providers might become unavailable and new service providers offering new services might go online. This means the environment will change over time as the system operates. The same principle is applied for service consumers. Service consumers can use a service and goes offline immediately. The number of service providers is unbounded, given that service providers can join the environment at any time. Services are owned by various stakeholders with different aims and objectives. There may be incompetent, unreliable or even malicious service providers which make the environment insecure. There is no central authority that can control all the service providers and consumers. In addition, it is assumed that service providers and consumers are self-interested.

The ability to locate services of interest in an open, dynamic, and distributed environment has become an essential requirement in many distributed systems. Traditional approaches to service discovery have generally relied on the existence of pre-defined registry services, which contain descriptions that follow some shared ontology. Often the description of a service is also very limited in existing registry services, with little or no support for problem-specific annotations that describe properties of a service. Semantic registries attempt to overcome this limitation, and provide: (1) a rich semantic description based on an ontology; (2) reasoning capability that can be applied to the semantic description. Semantic matching generally focuses on the problem of identifying services on the basis of the capabilities that they provide. Such matching is provided through a language to express the capabilities of services, and the specification of a matching algorithm between service advertisements and service requests. The use of standards in representation techniques and application-specific concepts play an important part in such service descriptions.

Many applications in computational science make use of numerical algorithms, either developed as part of the project or obtained from third parties – examples include libraries from the Numerical Algorithms Group (NAG). The complexity of such algorithms can vary from simple matrix solving to more complex data analysis – such as clustering or classification techniques. Furthermore, the ability to access such algorithms as Web Services allows easy integration of such capability within an existing application. This also provides a loose coupling between the application and the numerical algorithm. To enable mathematical objects to be exchanged

SOCP'07, June 26, 2007, Monterey, California, USA.

between computer programs in an unambiguous manner (stored in a database or published on the worldwide web), the OpenMath [1] standard was introduced. The main part of OpenMath is the idea of a "Content Dictionary" (CD) – defining a collection of symbols that may be used in a mathematical formula, and their associated meaning and relationships. Essentially, a CD is a small, specialized ontology which may be extended as new symbols are introduced into mathematics. OpenMath may be used to encode the mathematical part of a problem to be solved – such as a differential equation or an integral. Furthermore, OpenMath may be used to describe properties of a mathematical service, by defining sets of symbols which are understood by the service, and those used to represent the result.

Both the MONET (Mathematics on the NET) [2] and GENSS (Grid-Enabled Numerical and Symbolic Services) [3] projects involve mathematical problem solving through service discovery and composition. Both involve the use of OpenMath to describe service capability and user queries. *Mathematical* capability descriptions turn out to be both a blessing and a curse: precise service description is possible thanks to the use of the semantic mark-up provided by OpenMath [1], but service matching can rapidly turn into intractable (symbolic) mathematical calculations unless care is taken.

The aim of this paper is to evaluate two semantic registries – instanceStore and OWLJessKB – to compare their scalability and response time for mathematical services. These two registries were chosen for this evaluation as they both provide description logic (DL) reasoning which is an important necessity in the area of mathematical service matching.

The paper is organised as follows. Section 2 describes the MONET ontologies used for this performance evaluation of the two semantic registries. In section 3 the measurement setup is described by introducing the target systems and the methodology used. Section 4 shows the measurement results outling the performance for loading the ontology, query and scalability. A conclusion of the findings is given in section 5.

2. MONET ONTOLOGIES

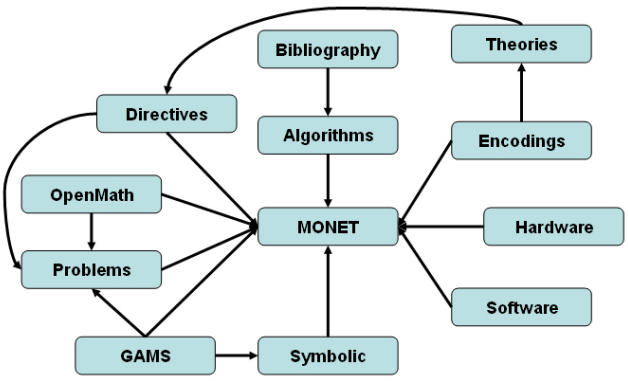

Figure 1: Relationship between MONET Ontologies

Interactions between ontologies[1] (the Ontology Web Language (OWL) is used to describe each ontology) that form part of the MONET project [4] can be found in Figure 1, and include:

- **GAMS (Guide to Available Mathematical Software):** provides an on-line index of available mathematical software, with each software library classified according to the type of problem it solves. The GAMS ontology is a simple class hierarchy. For example, one-dimensional quadrature is a subclass of quadrature, similarly one-dimensional quadrature over a finite interval is a subclass of one-dimensional quadrature etc.
- **Symbolic:** extends GAMS with support for symbolic computation. The symbolic ontology has been extended with a small taxonomy of the GAMS "O" category (symbolic computation systems).
- **OpenMath:** an XML-based encoding format for the representation of mathematical expressions and objects. Terms (referred to as "Constants") of the language have semantics attached to them and are called symbols (e.g., sin, integral, matrix, etc.), and groups of related symbols are defined in content dictionaries.
- **Hardware:** is used to describe either machine types or individual machines. The idea is that a user might request that a service run on a particular architecture (e.g. Sun Enterprise 10000), a general class of machine (e.g. shared memory), or a machine with a certain number of processors.
- **Software:** allows a user to express a preference for a service that makes use of a particular software library.
- **Problems:** may be described in terms of inputs and outputs, pre-conditions and post-conditions, and make use of pre-defined XML schema [5]. Within this ontology, each problem is represented as a class, which can have properties indicating bibliography entries and their generalizations. The most interesting property is `openmath head` whose range is an object from the `OpenMathSymbol` class. This represents a particular symbol which can be used to construct an instance of the problem in question.
- **Algorithms:** there are two sub-classes in this ontology: (1) Algorithm: which describes well-known algorithms for mathematical computations, and (2) Complexity: which provides classes necessary for representing complexity information associated with an Algorithm.
- **Directives:** this ontology is a collection of classes which identify the task that is performed by the service as described in [5] – example to decide, solve or prove a particular mathematical expression.
- **Theory:** this ontology collects classes that represent available formalized theories in digital libraries of mathematics.
- **Bibliography:** represents entries in well-known indices such as Zentralblatt MATH [6] and MathSciNet [7] and allows them to be associated with particular algorithms.
- **Encoding:** this ontology contains a (small) collection of classes which represent the formats used for encoding mathematical objects.
- **Monet:** imports all the ontologies described above.

[1] *http://www.cs.usask.ca/faculty/ludwig/monet.owl*

3. MEASUREMENT SETUP

A set of measurements are described that: (1) evaluate the time it takes to load the MONET ontologies – briefly described in section 2, (2) the associated query response times, (3) the overall scalability of two semantic registries. Scalability analysis involved increasing the number of services hosted in the registry to 100,000.

3.1 Target Systems

3.1.1 OWLJessKB Implementation

For the GENSS project a mathematical matchmaker [2] was developed [8]. It is based on OWLJessKB – a memory-based reasoning tool that may be used over ontologies specified in OWL. To store service descriptions, a MySQL database was used, residing on a different machine. OWLJessKB uses the Java Expert System Shell (JESS) [9] as its underlying reasoner. The OWLJessKB implementation loads multiple ontologies (the location of each is specified as a URL) into memory from a remote Web server. Reasoning in this instance involves executing one or more JESS rules over the ontology. JESS makes use of the Rete algorithm [10], which is intended to improve the speed of forward-chained rule systems (this is achieved by limiting the effort required to recompute the conflict set after a rule is fired). However, it has high memory requirements due to the loading of the ontology into the Rete object. Once it is created and set, it is fast to call rules and queries in order to infer the semantic relations of the ontology loaded. A key limiting factor in OWLJessKB is the time to load the ontology into primary memory (RAM), and the total RAM size on the host platform.

3.1.2 instanceStore

A DL knowledge base (KB) is made up of two parts: a *terminological* part (the terminology or Tbox), and an *assertional* part (the Abox). Each part consists of a set of axioms. The Tbox asserts facts about concepts (sets of objects) and roles (binary relations between objects), usually in the form of inclusion axioms. The Abox asserts facts about individuals (single objects), usually in the form of instantiation axioms [11]. The instanceStore [12] is a Java application for performing efficient and scalable DL reasoning over individuals. The OWL ontology is parsed into a Java OWLOntology object as opposed to the OWLJessKB Implementation where the ontology is loaded into memory. The instanceStore system tackles the problem of requiring vast volumes of individuals by applying a well-known idea in knowledge representation, namely supporting reasoning by means of databases. Assertions over individuals are stored in a database, together with information inferred using a DL reasoner over the position in the ontological taxonomy of their corresponding descriptions. This allows the system to reduce the amount of reasoning to pure terminological reasoning, while maintaining soundness and completeness (provided that no relation between individuals exists). The instanceStore is an implementation having the ontologies and the database (Hypersonic [13]) stored locally (i.e. hosted on one machine).

3.2 Methodology

Two semantic registry implementations instanceStore version 1.4.1 [3] and OWLJessKB version owljesskb20040223.jar [4] were tested. The test environment included an Intel Pentium III processor 996MHz, 512MB RAM, and a Windows XP Professional, running Java SDK 1.5.0 and Jess 6.1p8.

3.2.1 Measurements - Loading ontologies

These measurements include memory tests for OWLJessKB as heap size values were set for the loading of different ontology sizes. Additional measurements were carried out to evaluate the performance of loading the ontologies for the OWLJessKB implementation and the instanceStore.

[2] *http://agentcities.cs.bath.ac.uk:8080/genss_axis/GENSSMatchmaker*
[3] *http://sourceforge.net/project/showfiles.php?group_id=95954*
[4] *http://edge.cs.drexel.edu/assemblies/software/owljesskb/*

3.2.2 Measurements - Query response times

The following four different query types were chosen:
(1) Simple assertion: Find all instances of a class `x`;
(2) Simple assertion: Verify whether instance `x` exists;
(3) Assertion individual: Confirm if constraint `y` is satisfied via a single object;
(4) Assertion aggregate: Confirm if constraint `y` is satisfied for an entire group.

3.2.3 Measurements - Scalability

These measurements involved analyzing the performance of query response times for populated registries, starting with 100 services stored and going up to 100,000 services. Hence, scalability is evaluated as the change in query response behaviour as additional services were added to the registry.

4. MEASUREMENT RESULTS

4.1 Loading Ontology Performance

The previously described MONET ontologies (Figure 1) were chosen for the performance measurements. The MONET ontologies consist of 2031 classes, 78 slots and 10 facets. When increasing the ontology size, only the classes within the ontology were considered and expanded. Different ontology sizes were used having the number of classes as shown in Table 1.

Table 1: Ontology Sizes

Ontology size	No. of classes
1	2031
1.5	3046
2	4062
2.5	5077
3	6092
3.5	7107
4	8122

During preliminary measurement tests it was found that the OWLJessKB implementation needs the heap size in the Java Virtual Machine to be modified to load all different ontology sizes. The first set of measurements were undertaken to investigate the necessary heap size required for the different ontologies. Figure 2 shows the memory heap size for varying ontology sizes. It shows a linear distribution starting from 109MB for ontology size 1 and ending at 847MB for ontology size 4. The regression line and the derived equation shown in the figure allow to calculate the memory heap size needed for a particular ontology size.

The maximum heap size in both target systems was set to 1 GB, which was found to be a sufficient value for OWLJessKB measurements shown in Figure 2. The following set of measurements compare the load time of the OWLJessKB implementation and the instanceStore. Ten test runs were conducted for each ontology size and target system. Figure 3 shows that the instanceStore loads the ontologies much faster than the OWLJessKB implementation. The growth rate in loading times for OWLJessKB is significantly higher than the instancestore – this indicates that OWLJessKB requires loading of the entire ontology into RAM prior to analysis – although this is primarily a one-off cost at initialization time.

4.2 Query Performance

The next set of measurements were carried out having four queries:

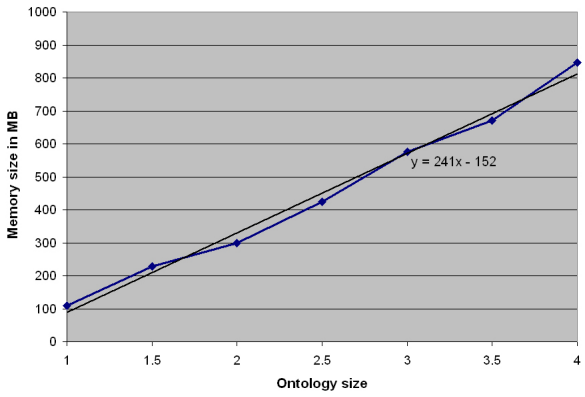

Figure 2: Memory heap size of OWLJessKB

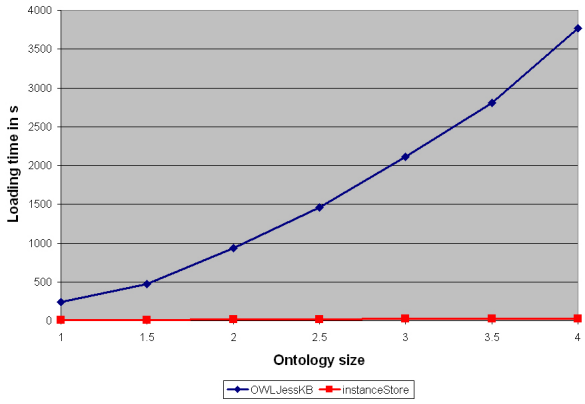

Figure 3: Comparison of load time

- Query 1: Query for services with GAMS classification GamsB (simple assertion finding all instances of a class x)
- Query 2: Query for services with RootFinding algorithm (simple assertion asking whether instance x exists; example query see below)
- Query 3: Query for services running on a single processor (assertion individual)
- Query 4: Query for services whose platform is not parallel (assertion aggregate)

Tested were the response times of varying ontology sizes for both target systems. The maximum heap size in both target system was set to 1 GB. For each query and each target system ten test runs were conducted and the average values were taken.

Queries for the OWLJessKB implementation need to be specified in the JESS notation, which is the following:

```
(defquery query-sub-class "Find all sub-classes"
  (triple
    (predicate "http://www.w3.org/2000/01/rdf-schema#
                subClassOf")
    (subject "http://monet.nag.co.uk/owl#
              service_algorithm")
    (object ?y)))
```

The queries for the instanceStore need to be defined in OWL Abstract Syntax (AOWL) [14], which is supposedly far more concise and human readable when expressing the fragments of an ontology. An example of one of the chosen queries in AOWL (which is query 2 for finding services with RootFinding algorithm) looks as follows:

```
restriction(
 <http://monet.nag.co.uk/owl#service_implementation>
 someValuesFrom( restriction(
   <http://monet.nag.co.uk/owl#service_algorithm>
    someValuesFrom(<http://monet.nag.co.uk/algorithm#
                    Root_Finding>))))
```

A service stored in both the instanceStore and the OWLJessKB implementation is `nagopt` – fitting the GAMS taxonomy `G1a1a` [2] (a variant of unconstrained optimisation) and using OpenMath as its I/O format. In this case the description also indicates that the service uses NAG's implementation of the safeguarded quadratic-interpolation algorithm.

```
<service name="nagopt">
    <classification>
        <gams_class>GamsG1a1a</gams_class>
        <problem>constrained_minimisation</problem>
        <input_format>OpenMath</input_format>
        <output_format>OpenMath</output_format>
        <directive>find</directive>
    </classification>
    <implementation>
        <software>NAG_C_Library_7</software>
        <platform>PentiumSystem</platform>
        <algorithm>Safeguarded_Quadratic-Interpolation
        </algorithm>
    </implementation>
</service>
```

In Figure 4 and 5 the response time of the four queries are shown for the OWLJessKB implementation and the instanceStore. It shows that the response times of the OWLJessKB are much smaller than the ones of the instanceStore. In both figures no increase in query response time for larger ontology sizes can be seen, which means that the ontology size does not increase the query response time. The average deviation (specifying measurement accuracy) is 1.8 ms for the OWLJessKB implementation, and 703 ms for the instanceStore. Comparing the instanceStore with the OWLJessKB implementation regarding the query performance shows that the OWLJessKB implementation performs on average 910 times faster than the instanceStore.

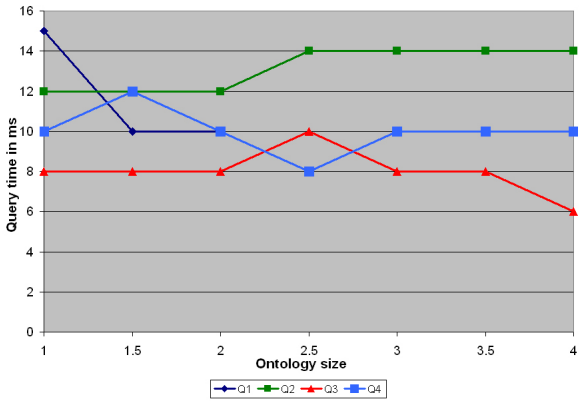

Figure 4: OWLJessKB Query Performance (Q1, Q4 times identical after ontology size 2)

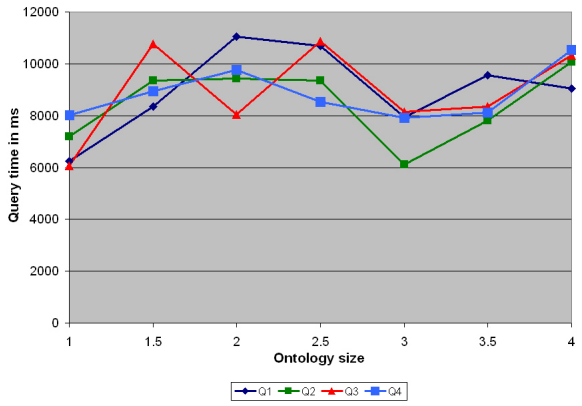

Figure 5: Query Performance of instanceStore

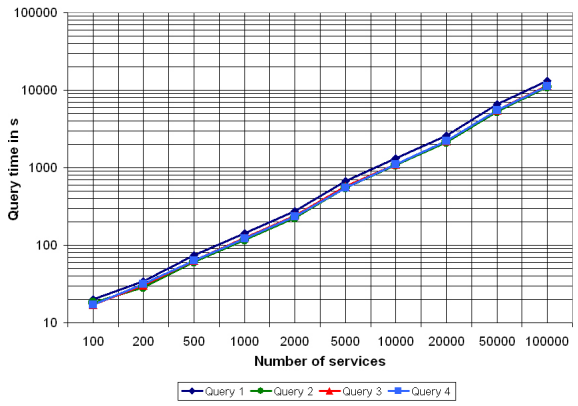

Figure 7: Scalability of populated services - instanceStore

4.3 Scalability Performance

Measurements were taken populating the OWLJessKB implementation and the instanceStore with an increasing number of services/individuals. Ontology size 1 was taken for this set of measurements, with a maximum heap size of 1 GB. Both semantic service registries were populated with 100, 200, 500, 1,000, 2,000, 5,000, 10,000, 20,000, 50,000 and 100,000 services (therefore, Figure 6 and 7 use a logarithmic scale). In Figure 6 a linear distribution between query time and number of services can be seen. However, the time measurements for 100, 200 and 500 services in Query 1, 2 and 4 are scattered around 10 ms and 20 ms, this is due to the measurement accuracy. It can be seen that Query 3 has the highest query times for the scalability measurements because the query performs an assertion of individuals which is more time consuming than the other queries. In Figure 7, a linear distribution can be seen between the time it takes to search for a service, subject to the number of services populated in the registry.

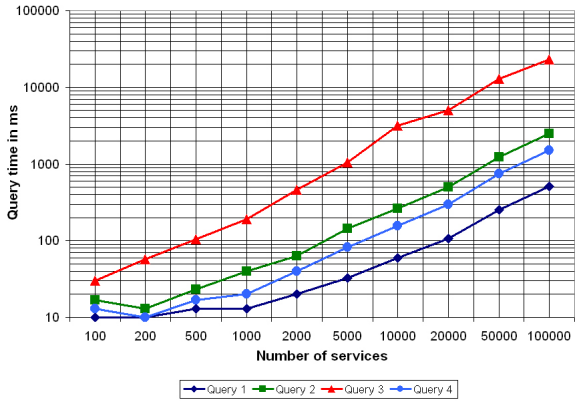

Figure 6: Scalability of populated services - OWLJessKB implementation

Comparing the OWLJessKB implementation with the instanceStore, regarding the scalability of services stored in the registry, it can be seen that the OWLJessKB implementation performs 1780 times faster than the instanceStore (average for OWLJessKB implementation was 1,374 ms versus 2,445,637 ms for instanceStore).

5. CONCLUSION

The conducted measurements for the two semantic registries revealed that the scalability of the OWLJessKB implementation is much higher than the instanceStore when searching for a service. However, the initialization stage, where the ontologies are loaded into memory is much slower. The initialisation of the OWLJessKB implementation uses the Rete algorithm, a drawback of which is its high memory requirements due to the loading of the ontology into the Rete object. Once it is created and set, it is fast to call rules and queries in order to infer the semantic relations specified in the ontology. One other influential factor, which is however a minor one, is that the instanceStore is a local implementation, which means that the ontology, database etc., is stored locally, whereas the OWLJessKB implementation has its ontologies and database stored on one or more remote machines. In general, the instanceStore performs better as a semantic registry when different ontologies constantly need to be loaded, and the OWLJessKB implementation performs better when a static set of ontologies is used and a large number of services are stored in the registry. It would be interesting for further work to find the maximum amount of services which can be stored in both semantic registries.

6. ACKNOWLEDGMENTS

The authors would like to thank Daniele Turi for his comments on the instanceStore.

7. REFERENCES

[1] OpenMath Society. Details at: `http://www.openmath.org`.

[2] MONET Consortium. Details at: `http://monet.nag.co.uk`.

[3] The GENSS Project. Details at: `http://genss.cs.bath.ac.uk`.

[4] O. Caprotti, M. Dewar and D. Turi, “Mathematical service matching using Description Logic and OWL”, Proceedings of 3rd Int’l Conference on Mathematical Knowledge Management (MKM’04), Springer-Verlag, 2004.

[5] O. Caprotti, D. Carlisle, A. Cohen and M. Dewar, “Problem Ontology: final version”, The MONET Consortium.

[6] Zentralblatt MATH. Available from: `http://www.emis.de/ZMATH/`.

[7] American Mathematical Society, "MathSciNet: Mathematical Reviews on the Web". Available from: http://www.ams.org/mathscinet.

[8] S.A. Ludwig, O.F. Rana, W. Naylor and J. Padget, "Matchmaking of Mathematical Web Services", In Proceedings of 6th International Conference on Parallel Processing and Applied Mathematics, Poznan, Poland, September 2005.

[9] Ernest J. Friedman-Hill, "Java Expert Systems Shell". Available from: http://herzberg.ca.sandia.gov/jess/docs/61/index.html.

[10] C. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem". Journal of Artificial Intelligence, 19:17–37, 1982.

[11] I. Horrocks, U. Sattler and S. Tobies, "Reasoning with individuals for the description logic SHIQ". Proceedings of the 17th International Conference on Automated Deduction (CADE-17), Springer-Verlag, 2000.

[12] D. Turi, "Instance Store". Available at: http://instancestore.man.ac.uk.

[13] The Hypersonic Database (HSQL DB). Available from http://hsqldb.org/.

[14] S. Bechhofer, P. F. Patel-Schneider and D. Turi, "OWL Web Ontology Language Concrete Abstract Syntax", Technical report, The University of Manchester, 2003.