Implementing the Instance Store

Sean Bechhofer, Ian Horrocks, Daniele Turi*
Information Management Group
Department of Computer Science
University of Manchester
Manchester, UK
<lastname>@cs.man.ac.uk

Abstract

We describe the implementation of the instance store, a DL system using a combination of TBox reasoning and database queries to perform efficient and scalable role-free ABox reasoning.

1 Introduction

Description Logic (DL [8]) reasoning over individuals is an important aspect of the vision behind the Semantic Web and it is crucial in applications of ontologies in areas such as bioinformatics (gene description) and web-service discovery. This also poses new challenges for ontological reasoning. Firstly, applications might require vast volumes of individuals exceeding the capabilities of existing reasoners. Secondly, while one can assume that changes in the terminological part of an ontology are relatively infrequent, the above scenarios require *dynamic*, *frequent* and, possibly, *concurrent modification* of the information related to the individuals in ontologies.

To tackle this problem we have developed a Java component, called *instance Store (iS)* [3]. As a starting point, we postulate there is no direct relation between instances, ie the ABox is role-free. This means that reasoning over instances can be reduced to reasoning over their descriptions (possibly involving properties). From an architectural point of view, the ontology of classes can then be treated as a static schema, loaded from a file into a DIG [9] compliant TBox reasoner such as FaCT [13] or RACER [12], while all the assertions about the individuals are dynamically added to and retrieved from a *database*. This way, we can exploit robust enterprise database technology offering *persistency*, *scalability*, and *secure* and *concurrent* transaction

*Work partly funded by the European Union MONET Project (IST-2001-34145).

management. All this, while ensuring both the soundness and completeness of the resulting reasoning for role-free ABoxes.

We have discussed the basic ideas behind *iS*, its performance, and related and future work in [14]. In this paper, we present its architecture and implementation issues in detail.

2 Architecture

The architecture of *iS* is shown in Figure 1 while its functionality is encapsulated in

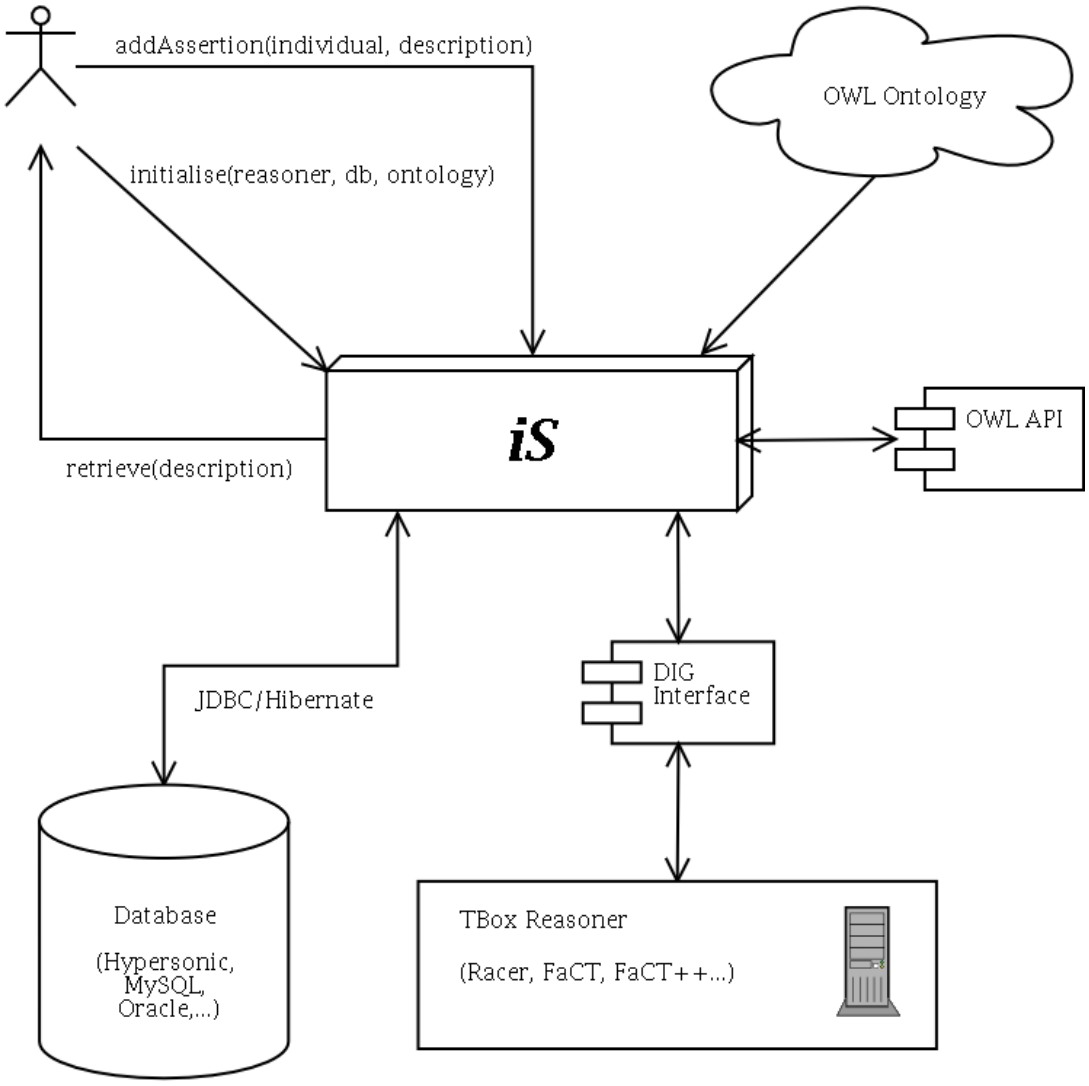

Figure 1: Architecture of *iS*

three basic methods as shown in Figure 2.

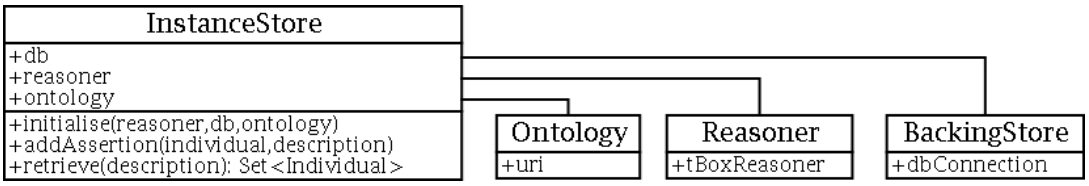

Figure 2: Basic class diagram of *iS* API

The recent W3C recommendation for web ontology languages is OWL [11], while the (de facto) XML standard for communicating with DL reasoners is the DIG Interface [9]. This is reflected in our design of the *iS*, which consists of a core component working with DIG ontologies and descriptions and of an OWL wrapper around it using the OWL API [5] and the DIG Interface API [1] to translate OWL to DIG and vice versa.

2.1 OWL Wrapper

In the OWL wrapper the ontology is in OWL and, at initialisation time, it is parsed into a Java `OWLOntology` object using the OWL API. The iS accepts OWL descriptions either as Java `OWLDescription` objects of the OWL API, or as description strings in OWL Abstract Syntax [10]. OWL ontologies and descriptions are submitted to the reasoner using suitable parsers and renderers in the OWL API and XMLBeans [7] and other classes from the DIG Interface API. (See Figure 3 for an overview.)

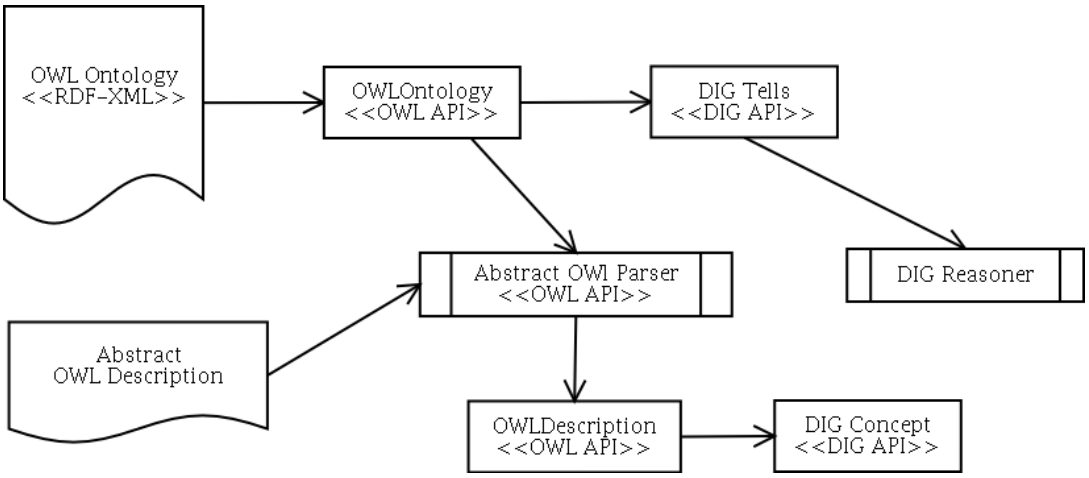

Figure 3: OWL to DIG

The *iS* also has a simple XSLT stylesheet which can convert DIG descriptions to descriptions in OWL abstract syntax.

2.2 Core Component

The first operation (`initialise`) connects to the database (creating the tables if needed) and the reasoner and loads the ontology into the reasoner.

The second operation (`addAssertion`) stores in the database the fact that individual (as a URI) is an instance of (DIG) description, together with additional information gathered through calls to the reasoner, such as the atomic concepts in the ontology the individual is instance of, and the position of the description in the taxonomy.

The third operation (`retrieve`) again uses the database and the reasoner (with respect to the ontology) to retrieve all individuals (again as URIs) which are instances of the description query.

3 Database

The *iS* tries to minimise the amount of reasoning required at retrieval time by storing as much information as possible about the descriptions used both during assertion and retrieval. For this we use a relational database whose schema is depicted in Figure 4.

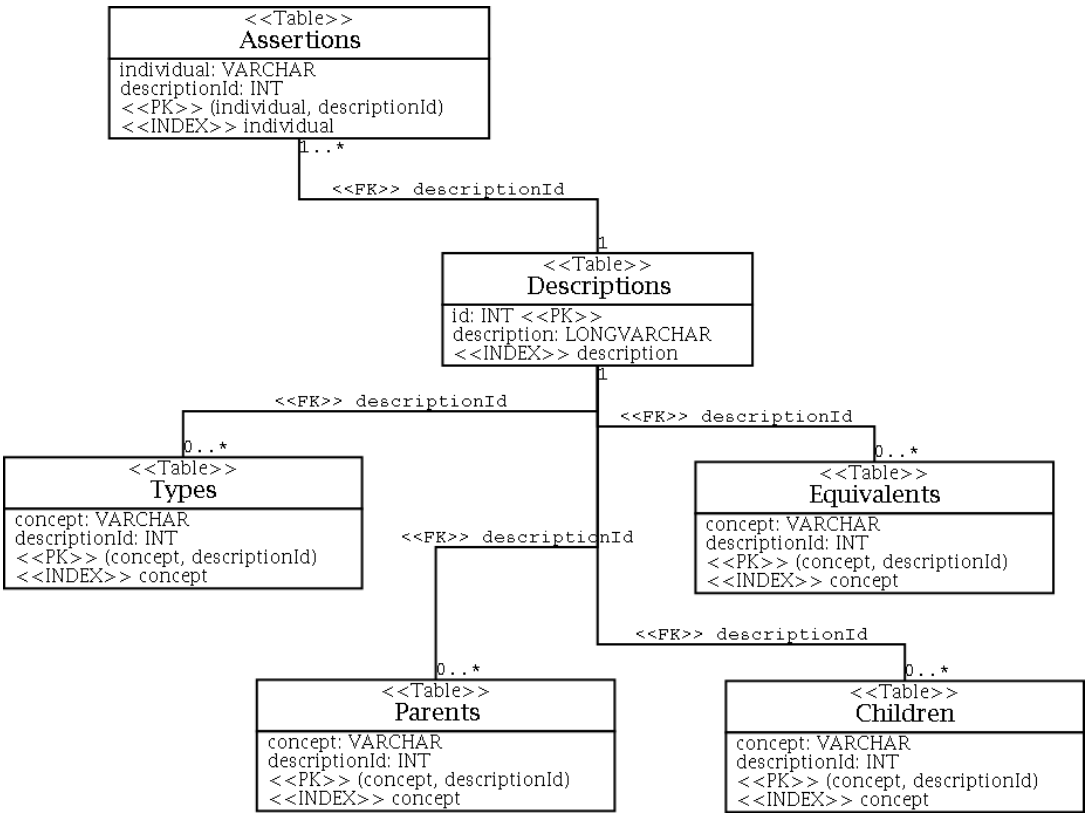

Figure 4: Database Schema for *iS*

Each description (either asserted or retrieved) is assigned a unique numerical identifier and stored in a dedicated `Descriptions` table; the identifier acts as primary key and the descriptions are indexed. The assertions are stored in the corresponding table[1] containing contains individuals and the identifiers of their associated descriptions; the latter are foreign keys referencing ids in the descriptions table; the individual/descriptionId pair forms a primary key. Next, we use a `Types` table containing description ids and all the primitive concepts in the ontology which subsume them; again the description ids are foreign keys, the concept/descriptionId pair is a primary key, and the concepts are indexed. Finally, we use three more tables of the same type as `Types` to store the primitive concepts which are, respectively, equivalent to, parent of and child of a given description.

We have implemented the above schema using three different DBMS: MySQL, Oracle, and Hypersonic. The latter is mostly for testing and demo purposes; it is entirely in Java and requires no installation other than including the corresponding jar in the classpath; each database consists of simply two files, one with properties and one with a script incrementally updated with the issued SQL statements.

MySQL offers the best performance, but it lacks set operations so in our algorithms below we mostly refer to the Oracle implementation.

4 Algorithms

Apart from the initialisation phase, the main functionality of *iS* consists of asserting that an individual is an instance of a class and retrieving all instances of a description. In both cases, *iS* first checks if the description is already in store. If this is not the case, then *iS* uses the TBox reasoner to classify it and then stores the result as shown in Algorithm 1.

Algorithm 1 $storeClassification(Description\ D) :\ int$

```
1: id ← database.addDescription(D)
2: classification ← reasoner.classify(D)
3: database.addToTypes(id, classification.getAncestors())
4: if classification.getEquivalents() ≠ ∅ then
5:     database.addToTypes(id, classification.getEquivalents())
6:     database.addToEquivalents(id, classification.getEquivalents())
7: else
8:     database.addToParents(id, classification.getParents())
9:     database.addToChildren(id, classification.getChildren())
10: end if
11: return id
```

[1] Here we refer to this table as `Assertions`, but in the actual implementation it is, misleadingly, called `primitive`.

In order to satisfy the constraint that there is a single description for each individual in the assertions table, before adding an assertion we first check whether the individual is already in store. If this is the case, then the corresponding description is conjuncted with the new description (unless subsumed). See Algorithm 2 for details.

Algorithm 2 $addAssertion(Individual\ I, Description\ D)$

```
1: if isInStore(I) then
2:    D_I ← database.getDescription(I)
3:    if not reasoner.subsumes(D, D_I) then
4:       D ← D ⊓ D_I
5:       retract(I)
6:    else
7:       return
8:    end if
9: end if
10: if reasoner.isInconsistent(D) then
11:    raise Exception
12: end if
13: if isStored(D) then
14:    id ← database.getDescriptionId(D)
15: else
16:    id ← storeClassification(D)
17: end if
18: INSERT INTO Assertions VALUES (I, id)
```

As for retrieval, the first step is to check whether the description whose instances are required is consistent. Next, *iS* checks whether the description is in store and, if not, the description and its classification are stored in the database. After this preprocessing phase, the actual algorithm begins.

Firstly, *iS* checks whether the description is equivalent to any primitive concept in the ontology. Since, at this stage, the classification of the description is already in the database, it is sufficient to query the table of equivalents. This is done in Query 1. Note that the query is a join which effectively returns all the ids of descriptions subsumed by primitive concepts equivalent to the query description. If the set returned

Query 1 $idsFromEquivalents(int\ \mathsf{id})$

```
SELECT DISTINCT Types.descriptionId FROM Types, Equivalents
  WHERE id = Equivalents.descriptionId
  AND Equivalents.concept = Types.concept
```

by such query is not empty then *iS* can just return the individuals corresponding to such description ids in the table of assertions.

In general, however, there might be no primitive concept equivalent to the query description. In that case we have to return the instances of children of the query

description and also check all the instances of its parents. There is a particular case though when this task can be accomplished in a relatively straightforward way, namely when the conjunction of the parents of the query description is equivalent to the query description: we then know that the desired set of instances consists of the intersection of the instances of each parent, union the instances of the children of description. (See Query 2.)

Query 2 $allIndividuals(int\ \mathsf{id},\ Set\ \{p_1, \ldots, p_n\})$

```
SELECT DISTINCT individual FROM Assertions
  WHERE descriptionId IN (
    SELECT descriptionId FROM Types WHERE concept = p_1
    INTERSECT
    ...
    INTERSECT
    SELECT descriptionId FROM Types WHERE concept = p_n
    UNION
    SELECT Types.descriptionId FROM Types, Children
      WHERE Children.concept = Types.concept
      AND Children.descriptionId = id)
```

When the above does not hold, the *iS* has then to perform the following, more complex process. First, collect all the description ids corresponding to instances of parents which are not also instances of children. This is performed by Query 3.

Query 3 $getCandidates(int\ \mathsf{id},\ Set\ \{p_1, \ldots, p_n\})$

```
SELECT DISTINCT description FROM Descriptions
  WHERE Descriptions.id IN (
    SELECT descriptionId FROM Types WHERE concept = p_1
    INTERSECT
    ...
    INTERSECT
    SELECT descriptionId FROM Types WHERE concept = p_n
    MINUS
    SELECT Types.descriptionId FROM Types, Children
      WHERE Children.concept = Types.concept
      AND Children.descriptionId = id)
```

Next, ask the TBox reasoner to compute the subset of the above candidate descriptions which are subsumed by the query description. This requires a single DIG ask, but as many internal checks as there are candidates. Here the performance of the reasoner really matters. The instances of this set together with the instances of the children of the query description forms then the final result. (See Algorithm 3 for full details.)

Algorithm 3 $retrieve(Description\ D) :\ Set$

if $reasoner.isInconsistent(D)$ **then**
  raise Exception
end if
if $isStored(D)$ **then**
  id $\leftarrow$ $database.getDescriptionId(D)$
else
  id $\leftarrow$ $storeClassification(D)$
end if
equivalents $\leftarrow$ $database.idsFromEquivalents$(id)
if equivalents $\neq \emptyset$ **then**
  return $database.getIndividuals$(equivalents)
end if
parents $\leftarrow$ $database.getParents$(id)
if $reasoner.subsumes(D, and$(parents)$)$ **then**
  return $database.allIndividuals$(id, parents)
end if
candidates $\leftarrow$ $database.getCandidates$(id, parents)
descriptions $\leftarrow$ $database.childrenDescr$(id) $\cup$ $reasoner.getSubsumed(D,$ candidates$)$
return $database.getIndividuals$(descriptions)

5 Enterprise *iS*

In many applications, many different clients might want to access *iS* concurrently and securely. This can be easily accomplished thanks to our architectural choices. Indeed, we have developed an Enterprise Java Beans version of *iS* which is deployed using the JBoss [4] J2EE application server. (See Figure 5.) It consists of:

1. a wrapper for `InstanceStore` implementing the `SessionBean` interface;
2. a managed bean (`MBean`) wrapper for the reasoner;
3. a Hibernate [2] version of `BackingStore`.

Hibernate allows to persist the plain Java objects corresponding to *iS* descriptions, individual, and primitive concepts. Object-relational mappings define the way the relevant properties in the objects are mapped to entries in a relational database. We used XDoclet [6] to generate the Hibernate object-relational mapping as well as the local and remote interfaces to the beans.

As for the reasoner, the use of managed beans means that several instances of the same TBox reasoner can be dynamically spawned when required by the clients (at least if the TBox reasoner is available as a DLL, ie dynamically linked library).

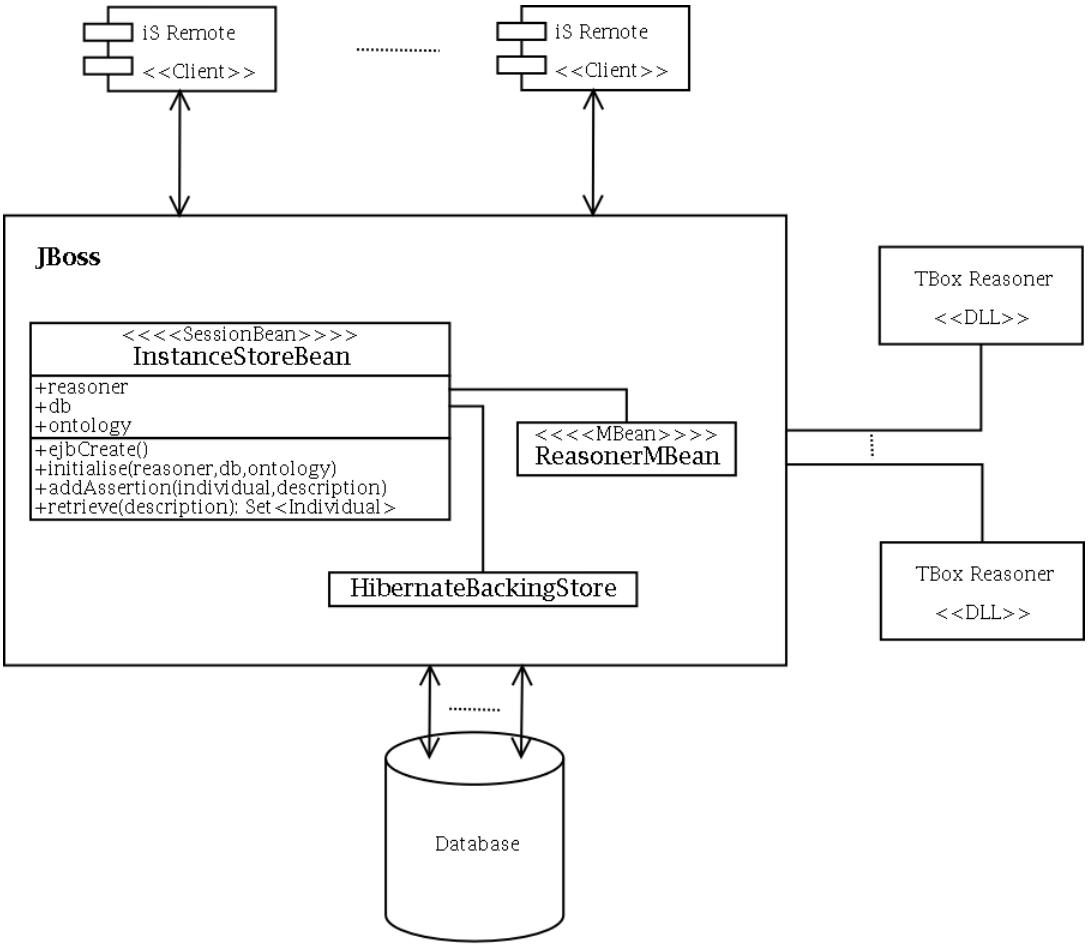

Figure 5: Enterprise *iS* deployed using JBoss

6 Conclusions

The architectural choices made in the implementation of *iS* ensure that we use appropriate technologies for appropriate tasks. It is clear that at some point the reasoner must be used in order to retrieve individuals, but in our approach it is only used when necessary. Databases are well suited to handling large amounts of data and are optimised for the performance of operations such as joins and intersections – for example the queries described in the query in Figure 2.

As discussed in Section 5, the architecture also allow us to make use of functionality supported by existing component architectures such as Enterprise Java Beans. Then issues like concurrency and security can be passed off to the application servers running the beans.

The use of a database back end also allows us to support persistency. A-boxes may well include large amounts of data, and we can make use of the fact that this is

precisely a task that databases are designed to support. There is still, of course, the question of persistent storage of the ontology within the reasoner, but again the separation of the reasoning component from the instance storage means that alternative reasoner implementations can simply be "swapped in" as necessary.

Acknowledgements. Phillip Lord participated in the early stages of the implementation of the *iS* and Nick Taylor and David Roberts from Stilo International PLC participated in the development of the enterprise *iS*.

References

[1] DIG Interface API. `http://dig.sourceforge.net`.

[2] Hibernate. `http://www.hibernate.org`.

[3] Instance Store API. `http://instancestore.man.ac.uk`.

[4] JBoss. `http://www.jboss.org`.

[5] OWL API. `http://sourceforge.net/projects/owlapi`.

[6] XDoclet. `http://xdoclet.sourceforge.net`.

[7] XMLBeans. `http://xml.apache.org/xmlbeans`.

[8] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook — Theory, Implementation and Applications*. Cambridge University Press, 2003.

[9] Sean Bechhofer. The DIG description logic interface: DIG/1.1. In *Proceedings of the 2003 Description Logic Workshop (DL 2003)*, 2003.

[10] Sean Bechhofer, Peter F. Patel-Schneider, and Daniele Turi. OWL Web Ontology Language Concrete Abstract Syntax, December 2003. Available from `http://owl.man.ac.uk/2003/concrete/latest/`.

[11] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. Technical Report REC-owl-ref-20040210, The Worldwide Web Consortium, February 2004. Available from `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`.

[12] Volker Haarslev and Ralf Moller. Description of the RACER system and its applications. In Rajeev Gore, Alexander Leitsch, and Tobias Nipkow, editors, *Automated reasoning: First International Joint Conference, IJCAR 2001, Siena, Italy, June 18–23, 2001: proceedings*, volume 2083 of *Lecture Notes in Artificial Intelligence*, New York, NY, USA, 2001. Springer-Verlag Inc.

[13] Ian Horrocks. Using an expressive description logic: FaCT or fiction? In *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 636–647, 1998.

[14] Ian Horrocks, Lei Li, Daniele Turi, and Sean Bechhofer. The Instance Store: DL reasoning with large numbers of individuals. In *Proc. of the 2004 Description Logics Workshop (DL 2004)*, pages 31–40, 2004.