# Query Answering Systems in the Semantic Web

Birte Glimm and Ian Horrocks
Department of Computer Science
The University of Manchester
Manchester, UK
`{glimm|horrocks}@cs.man.ac.uk`

**Abstract**

In this paper a new query answering system is presented for querying knowledge bases in the Semantic Web. The implementation follows the DAML+OIL Query Language Abstract Specification (DQL) and supports acyclic conjunctive queries. The system uses a Description Logic (DL) reasoner to answer the queries and the conjunctive queries are transformed into DL retrieval or boolean queries. After the introduction to the new DQL implementation, a comparison with other systems follows. This includes the recently introduced new Racer Query Language (nRQL), the DQL implementation provided by the Knowledge Systems Laboratory of the Stanford University and a DQL implementation provided by the University of Maryland, Baltimore County. The paper highlights and compares the different approaches of these systems.

## 1 Motivation

Current Description Logic (DL) systems [7, 9] offer a powerful inference mechanism, e.g., to compute the sub-concepts of a given concept, but they usually offer a weak query language. Current DL reasoners generally support the following queries to access the assertional knowledge of a knowledge base:

**retrieval** retrieve the instances of a given concept

**realisation** determine the most specific concept an individual is an instance of

**instantiation** boolean query asking if an individual (a pair of individuals) is an instance of a given concept (role)

There is, however, no support for queries that ask for n-tuples of related individuals or for the use of variables to formulate a query.

**The DQL Specification** To overcome some of these limitations, the DAML Joint Committee announced in August 2002 the DAML Query Language (DQL) Abstract Specification and replaced it in April 2003 with a new release [5]. The specification is based on user requirements for a query language in the Semantic Web, and it defines a number of features that a DQL server must support. For example, a query may contain variables for which a binding to an individual name is required in the answer, and others for which no such binding is required. In the latter case, only the existence of an appropriate individual is required. The specification also defines a protocol for a query answering dialogue. The initially targeted knowledge representation language was DAML+OIL [11], but DQL was meant to be easily adaptable to other knowledge representation languages, such as OWL [2]. Therefore, the specification was given on an abstract level, without a definition for a concrete language. An adaptation, e.g., to OWL, is therefore possible without significant changes. In fact the Knowledge Systems Laboratory of the Stanford University already provides a proposal for an OWL Query Language (OWL-QL) specification [6] adapted from DQL.

**DQL Implementations** Recently a new system, developed at the University of Manchester,[1] became available. The implementation relies on DL reasoners transforming incoming queries to DL statements that can be passed to a DL reasoner. This transformation process is described in detail in the next section.

Two other implementations were already available: one provided by the Knowledge Systems Laboratory of the Stanford University (KSL)[2] and the other developed at the University of Maryland, Baltimore County (UMBC). All three systems use different approaches to answer queries, and Section 3 provides a brief comparison. The comparison also includes the DL reasoner Racer, which has recently been extended with its own new Racer Query Language (nRQL) [8]. nRQL was not meant as a DQL implementation, and therefore does not comply with the specification, but nevertheless it is a step towards better query support.

## 2 From DQL to DL Queries

If one intends to use a DL reasoner to answer DQL queries, a transformation of the query is in most cases necessary. One reason for this is, that DQL allows to query for n-tuples of individuals, which is currently not supported by standard DL reasoners. With DQL a user can also use different kinds of variables in a query. To understand the different variables, the semantics of a DQL query and a query answer has to be explained. A DQL query contains a query pattern that represents a set of DAML+OIL (OWL) sentences in which some URI references

[1] `http://www.cs.man.ac.uk/~glimmbx/download/DQL.zip`

[2] `http://ksl.stanford.edu/projects/owl-ql/owql-20040623.zip`

are replaced by variables. A query answer provides bindings of URI references or literals to some of the variables. After applying the bindings to the variables and treating the remaining variables as existentially quantified, the resulting statement must be entailed by the knowledge bases (KB) used to answer the query. Variables for which a binding is required are called *distinguished* or *must-bind variables*, variables for which no binding should be returned are called *non-distinguished* or *don't-bind variables*, and variables for which a binding may be returned are called *may-bind variables*. May-bind variables do not add any power to the language, because the query answer can also be computed by a sequence of queries using only must-bind and don't-bind variables [4]. Therefore they are not treated in further detail here.

## 2.1 Conjunctive Queries

Conjunctive queries are of the form $\langle\vec{x}\rangle \leftarrow conj(\vec{x};\vec{y};\vec{z})$. The vector $\vec{x}$ consists of so called distinguished variables that will be bound to individual names of the knowledge base used to answer the query. The vector $\vec{y}$ consists of non-distinguished variables, which are existentially quantified variables. The vector $\vec{z}$ consists of individual names, and $conj(\vec{x};\vec{y};\vec{z})$ is a conjunction of atoms. An atom is of the form $v_1 : \texttt{C}$ or $\langle v_2, v_3\rangle : \texttt{r}$ where $\texttt{C}$ is a concept name, $\texttt{r}$ is a role name and $v_1$, $v_2$, $v_3$ are individual names from $\vec{z}$ or variables from $\vec{x}$ or $\vec{y}$. For easy readability, must-bind variable names in a query are prefixed with `?`, don't-bind variables are prefixed with `!`, and individual names are not prefixed and start with a letter. Concept names are written in upper case letters, while role and individual names are written in lower case.

## 2.2 Query Graphs

A conjunctive query $q$ can be represented as a directed labelled graph $G(q) := \langle V, E\rangle$, where $V$ is a set of vertices, and $E$ is a set of edges. The set $V$ consists of the union of the elements in $\vec{x}$, $\vec{y}$, and $\vec{z}$. The set $E$ consists of all pairs $\langle v_1, v_2\rangle$, such that $v_1, v_2 \in V$ and $\langle v_1, v_2\rangle : \texttt{r}$ is an atom in $q$. A node $v \in V$ is labelled with a concept $C_1 \sqcap \ldots \sqcap C_n$ such that, for each $C_i$, $v : C_i$ is an atom in $q$. Each edge $e \in E$ is labelled with a set of role names $\{\texttt{r} \mid \langle v_1, v_2\rangle : \texttt{r}$ is an atom in $q\}$.

The function $\mathcal{L}(v), v \in V$ returns the label for $v$. If $\mathcal{L}(v)$ is empty, the top concept ($\top$) is returned. The function $\mathcal{L}(e), e \in E$ returns a set of edge labels for $e$. The function $\mathcal{L}^-(e), e \in E$ returns a set of inverted edge labels, such that $\mathcal{L}^-(e) = \{r | r^- \in \mathcal{L}(e)\}$. The function $flip(G, \langle v_1, v_2\rangle)$, $\langle v_1, v_2\rangle \in E$ creates a new graph $G' := \langle V', E'\rangle$, with $V' := V$, $E' := (E \setminus \{\langle v_1, v_2\rangle\}) \cup \{\langle v_2, v_1\rangle\}$, and $\mathcal{L}(\langle v_2, v_1\rangle) = \mathcal{L}^-(\langle v_1, v_2\rangle)$. The function $pred(v_1), v_1 \in V$ returns a set of vertices $\{v_1 | v_1, v_2 \in V \wedge \langle v_2, v_1\rangle \in E\}$.

Two vertices $v_1, v_2 \in V$ are adjacent, if $\mathcal{L}(\langle v_1, v_2\rangle) \neq \emptyset$ or $\mathcal{L}(\langle v_2, v_1\rangle) \neq \emptyset$.

The vertex $v_1 \in V$ is reachable from $v_2 \in V$, if $v_1$ is adjacent to $v_2$ or if there is a another vertex $v_3 \in V$ such that $v_3$ is adjacent to $v_1$, and $v_2$ is reachable from $v_3$. The graph $G(q)$ is cyclic, if there is a $v \in V$, such that $\mathcal{L}(\langle v, v\rangle) \neq \emptyset$ or if there is a $v' \in V$, such that $v$ is adjacent to $v'$ and if one element is removed from $\mathcal{L}(\langle v, v'\rangle)$, $v'$ is still reachable from $v$. $q$ is an acyclic conjunctive query if $G(q)$ is not cyclic.

## 2.3 The Rolling-up Technique

Conjunctive queries are not supported directly by a DL reasoner. If a query contains only distinguished variables, one could replace all variables with individual names from the knowledge base and use a sequence of instantiation queries to determine if the statement is true in the knowledge base. To compute a complete query answer set with this approach, it is necessary to test all possible combinations of individual names. This is very costly, and furthermore, this approach is not applicable to queries with non-distinguished variables.

In 2001 Tessaris [10] proposed a rolling-up technique that can be used to eliminate non-distinguished variables from a query. The technique is applicable to acyclic conjunctive queries and the DQL server implemented in Manchester uses this technique to compute the query answers.

The basic principle behind the rolling-up technique is based on the semantic equivalence of the two formulae $\langle$`?x`$\rangle \leftarrow$ `?x:C` $\wedge$ $\langle$`?x, !y`$\rangle$`:r` and $\langle$`?x`$\rangle \leftarrow$ `?x:(C` $\sqcap$ $\exists$`r.`$\top$`)` [3]. In the latter statement `!y` is omitted, but all bindings for `?x` still imply the existence of an appropriate individual. Bindings for `?x` are now available through a normal retrieval query for the concept `(C` $\sqcap$ $\exists$`r.`$\top$`)`.

**Queries with One Distinguished Variable** Queries with only one distinguished variable can always be transformed into such a single retrieval query. The process is best illustrated using the query graph $G(q)$ of a query $q$ (Figure 1). For the readers convenience the distinguished variables are represented by a filled node ($\bullet$), whereas non-distinguished variables and individuals are represented by an unfilled node ($\circ$).

$\langle$`?w`$\rangle \leftarrow$ `?w:PERSON` $\sqcap$ $\langle$`?w, !x`$\rangle$`:owns` $\sqcap$ $\langle$`?w, !y`$\rangle$`:loves` $\sqcap$ $\langle$`!z, !y`$\rangle$`:haschild`

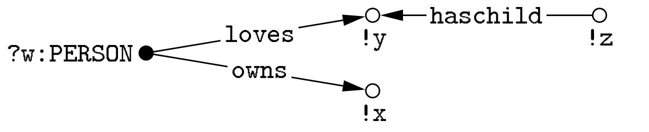

Figure 1: A query and its query graph.

First of all, the query graph is transformed into a tree with the distinguished variable as root. The function $flip(G, e), e \in E$ is applied to change edge directions if necessary to transform the graph into a proper tree. The left hand part of Figure 2 shows the resulting tree. Then the rolling-up starts from the leaves of the tree. A leaf, e.g. `!z`, is selected and the vertex and its incoming edge are replaced by conjoining the concept $\exists\,\mathcal{L}(pred(\texttt{!z}), \texttt{!z}).\mathcal{L}(\texttt{!z})$ to the label of $pred(\texttt{!z})$. The right hand part of Figure 2 shows the result of the first rolling-up step. The $\top$ conjunct could be omitted without changing the semantics. This step is applied to each leaf until only the distinguished variable at the root is remaining. The label of the root node can now be used to retrieve the individual names that are valid bindings for the distinguished variable. For this example these are instances of the concept $\texttt{PERSON} \sqcap \exists\,\texttt{owns.}\top \sqcap \exists\,\texttt{loves.(}\top \sqcap \exists\,\texttt{haschild}^{-}\texttt{.}\top\texttt{)}$.

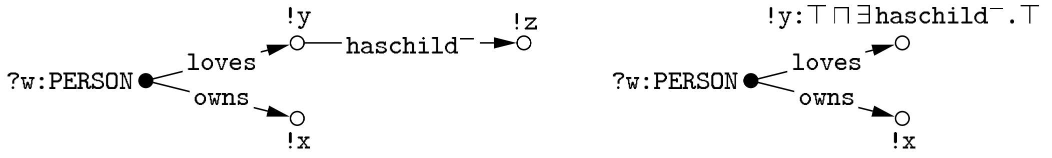

Figure 2: Two states of a query graph in the rolling-up process.

**Queries with Individual Names** In a DL that supports the oneOf constructor, which allows the definition of a concept by enumerating its instances, the rolling-up can use the individual name directly in the concept expression. Nodes for an individual name can then be treated like a non-distinguished variable with the concept {individual name} as label. E.g., the query $\langle \texttt{?x} \rangle \leftarrow \langle \texttt{?x, mary} \rangle\texttt{:loves}$ is rolled-up into a retrieval query for instances of the concept $\exists\,\texttt{loves.\{mary\}}$. Unfortunately most reasoners do not support the oneOf constructor, but it is still possible to deal with such queries using a so called representative concept [10]. The representative concept is a so far unused concept name, which is used instead of the individual name, the ABox being extended with an assertion stating that the individual is an instance of its representative concept. E.g., the query could be answered by retrieving the concept instances of $\exists\,\texttt{loves.P}_{mary}$, after the assertion $\texttt{mary:P}_{mary}$ is added to the KB.

**Boolean Queries** If a query contains only non-distinguished variables, the query answer is true, iff for each variable the KB entails the existence of an individual that fulfils all defined constraints, i.e., concept or role assertions. The boolean query $\langle\rangle \leftarrow \langle \texttt{acar, !x} \rangle\texttt{:ownedby} \wedge \texttt{!x:PERSON}$ against the knowledge base in Example 2.1 should be answered with true, since the existence of such a person is entailed by the KB.

**Example 2.1**
$$\begin{aligned} \text{KB} &= \{\mathcal{T}, \mathcal{A}\} \\ \mathcal{T} &= \{\texttt{CAR} \sqsubseteq \exists\,\texttt{ownedby.PERSON}\} \\ \mathcal{A} &= \{\texttt{acar:CAR}\} \end{aligned}$$

**Queries with Multiple Distinguished Variables** If a query contains multiple distinguished variables, the query can not be rolled-up into a single DL retrieval query. To avoid a test of all possible combinations of individual names, as necessary for the simple approach described in Section 2.3, the rolling-up technique is nevertheless helpful. To start the query answering process, one of the distinguished variables is selected as the root node, and all other variables are treated as non-distinguished. The query graph is transformed into a tree and the rolling-up process is applied as described above for the case with only one distinguished variable. The retrieved individual names are candidates for the binding of the variable. This step is repeated for all distinguished variables.

Not every combination of the retrieved candidates is possible, and to determine the valid combinations further boolean tests are necessary. To avoid as many boolean tests as possible further optimisations can be used at this point.

## 2.4 Optimisation Techniques

One promising approach is to use an iterative process that eliminates unsuitable combinations as soon as possible. Consider, e.g., the query and its query graph in Figure 3, where `?x` has four candidates (i.e., $x_1 \ldots x_4$), `?y` has two candidates ($y_1$, $y_2$), and `?z` has ten candidates ($z_1, \ldots, z_{10}$) after the rolling-up.

$$\langle \texttt{?x}, \texttt{?y}, \texttt{?z} \rangle \leftarrow \langle \texttt{?x}, \texttt{?y} \rangle : \texttt{r} \wedge \langle \texttt{?y}, \texttt{?z} \rangle : \texttt{s}$$

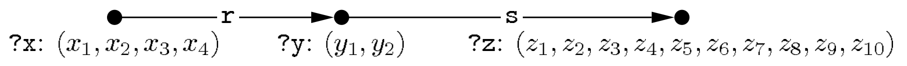
r
s
?x: $(x_1, x_2, x_3, x_4)$
?y: $(y_1, y_2)$
?z: $(z_1, z_2, z_3, z_4, z_5, z_6, z_7, z_8, z_9, z_{10})$

Figure 3: An example query with its query graph and candidates.

If we had not used the rolling-up to retrieve the candidates, the number of necessary boolean tests would have been factorial in the number of named individuals in the KB. With the rolling-up and boolean tests for all possible candidate combinations, the number of tests is still the product of the number of candidates, i.e., 80 tests in this example. An optimised strategy could start at the variable with the most candidates (i.e., `?z`) and retrieve the concept instances of $\exists\, s^{-}.P_{y_1}$, where $P_{y_1}$ is the representative concept for $y_1$. In this way, one can determine which of the candidates for `?z` are related to $y_1$. This is repeated for $y_2$. By testing for valid pairs first, one can skip many unnecessary test, e.g., if $y_1$ and $z_1$ are not related, no tests for candidates of `?x` are necessary. The process is repeated for the variable with the next highest number of candidates (i.e., `?x`). Compared to the 80 boolean tests necessary before, this approach needs four retrieval queries to determine the valid candidate combinations.

Another optimisation could use structural knowledge about the roles in the KB to exclude impossible candidate combinations even before the above tests are used. The system developed in Manchester does not yet fully implement these optimisations.

## 3 System Comparison

The algorithm described in the previous section was used to implement a web service, compliant with the DQL specification (except for some unsupported features, e.g., the use of multiple KBs to answer a query). The system answers acyclic conjunctive queries correctly and completely and can be used with any DIG [1] compliant DL reasoner, e.g., with the DL reasoner RACER. The next section highlights the differences to other query answering systems that are currently available, with the main focus on DQL (OWL-QL) implementations.

**The Stanford OWL-QL Server** The Knowledge Systems Laboratory (KSL) of the Stanford University provides an OWL-QL implementation that supports DAML+OIL and OWL knowledge bases. It seems to be the successor of their DQL implementation, which is no longer accessible, and it uses the same settings. The system uses the first order logic theorem prover JTP[3] to answer the queries. The DQL server is implemented as a wrapper around the theorem prover. A query consists of DAML+OIL or OWL statements (in RDF triple notation) with URI references replaced by variables. Compared to acyclic conjunctive queries, the supported query language is therefore richer. Unfortunately the system does not answer all allowed queries. For some queries the server simply terminates the communication with a client. This is allowed by the DQL specification, but probably not what a client expects.

As an example, consider again the KB in Example 2.1 on page 5. The query $\langle \texttt{?x} \rangle \leftarrow \texttt{?x:CAR} \wedge \langle \texttt{?x}, \texttt{!y} \rangle\texttt{:ownedby} \wedge \texttt{!y:PERSON}$ is correctly answered with the binding `acar` for `?x`. However, the slightly modified query $\langle \texttt{?x} \rangle \leftarrow \texttt{?x:CAR} \wedge \langle \texttt{?x, !y} \rangle\texttt{:ownedby} \wedge \texttt{!y:CAR}$, asking for a car that is owned by a car, is also answered with the binding `acar` for `?x`.

Both implementations were also tested with a second, more complicated query: $\langle \texttt{?x} \rangle \leftarrow \langle \texttt{?x, !y} \rangle\texttt{:r} \sqcap \langle \texttt{!y, b1} \rangle\texttt{:r} \sqcap \texttt{!y:C}$ against the KB in figure 4. The query asks for individuals that have an `r` successor that is a `C` and that has `b1` as `r` successor. The difficulty is that in this case there is no nameable instance of the concept `C`, but it can be inferred that either `c1` or `c2` is a `C`. Using this inference, `a1` is clearly a correct binding for `?x`. However, the KSL implementation also provides `c1` and `c2` as a binding for `?x`.

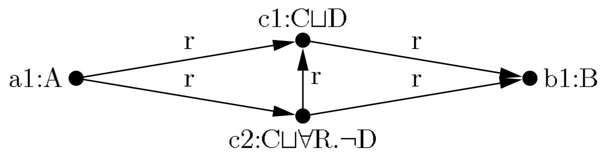

Figure 4: An example knowledge base.

[3]`http://www.ksl.stanford.edu/software/JTP`

It seems that the system has difficulties with non-distinguished variables, and queries often cause unexpected results. The reasons for this behaviour could be due to the communication with the used theorem prover or in the theorem prover itself. If the implementation is improved in this respect, however, it would provide a powerful and complete implementation of the DQL specification. For practical use, the system would benefit from better error handling and error explanation and a detailed documentation would be desirable.

**The UMBC Implementation** Another DQL implementation (compliant to the predecessor of the current DQL specification) has been realised by the University of Maryland, Baltimore County (UMBC).[4] DQL was implemented by the UMBC for communication between agents in a multi-agent environment. The implementation is based on the FIPA protocol,[5] and has no publicly available query interface. DQL is only used internally for specific agent to agent communication tasks. Therefore the implementation is probably not interesting for users who look for a better query support in general, and the system is not further introduced here. The reasoning in this system is based on JESS (Java Expert System Shell).[6]

**The new RACER Query Language** The recently introduced new RACER Query Language (nRQL) [8] is not geared to the DQL specification, therefore it misses all the protocol specific elements, such as termination tokens or the delivery of answers in a bundle with a specifiable size bound. In addition nRQL does not support non-distinguished variables. Although nRQL is far away from the DQL implementation, it is nevertheless a step towards better query support, and it is therefore introduced here very briefly. The query language itself is very rich, as it supports the retrieval of variable bindings in arbitrary concept and role expressions. In contrast to the other systems introduced here, all variables are distinguished, even if they are not included in the answer. For an example, the reader may again consider the KB in Example 2.1 (page 5). The nRLQ query `(retrieve (?x) (and (?x CAR) (?y PERSON) (?x ?y ownedby)))` returns all cars that are owned by a person. Although only cars are in the answer, a named individual must exist in the KB that is specified as owner of the car. As a result the query answer for this example is empty.

Another feature, which was added to nRQL, is negated query atoms, implemented using a negation as failure semantics. This is contrary to the Open World semantics normally used in DL systems (and also by RACER). nRQL uses the same operator (`not`) for negated query atoms and for concept negation, which could probably lead to confusion and the users have to be care-

[4]http://www.cs.umbc.edu/~finin/papers/dqlFIPA.html

[5]http://www.fipa.org/

[6]http://herzberg.ca.sandia.gov/jess/

ful with the formulation of such a query. The nRQL query `(retrieve (?x) (not (?x PERSON)))`, using the negation as failure semantics, therefore returns `acar`. Due to the Open World semantics for concept negation, the modified query `(retrieve (?x) (?x (not PERSON)))` returns an empty answer set, since RACER cannot prove that `acar` is not an instance of the concept `person`.

nRQL offers more features than the ones described here and for details the reader is referred to the RACER documentation.[7]

## 4 Conclusion

Efforts are currently being made, to develop better query support for knowledge representation systems. The establishment of OWL as a W3C recommendation may also promote the proposed OWL-QL specification[8] and so encourage improvements for the currently available systems or the development of new query answering systems.

So far, all introduced systems have some drawbacks. The Stanford implementation covers all features defined by the DQL specification, but delivers in some cases incorrect answers and rejects some queries, without providing an answer. The Manchester implementation does not support all DQL features and is restricted to acyclic conjunctive queries. Both systems are available as Java applications and the Stanford implementation is also available as a servlet, while the Manchester implementation is also available as a web service. Both provide a web client interface and are able to deal with OWL and DAML+OIL knowledge bases.

The UMBC implementation is not publicly available and does not, therefore, help to improve query support for knowledge bases in general. nRQL provides richer query support, but is not meant as a DQL implementation and is therefore missing many DQL features. In addition, the restriction that a binding is required for all variables, even for those not expected to appear in the answer set, would make it difficult to formulate queries such as the one in Section 3 against the KB in figure 4. Apart from this, nRQL is easy to use, and the documentation provides a good introduction to the new features of nRQL.

For all described systems there are still improvements possible. One main topic for query answering systems is scalability. The query answering times for knowledge bases with large amounts of individuals are still far away from the results achieved by databases. For the DQL implementation developed in Manchester, the boolean queries that are necessary to check valid combinations of variable bindings, can cause major delays in case of many candidates. The

[7]The documentation, which includes a section about nRQL, is available from the RACER download page: `http://www.cs.concordia.ca/~haarslev/racer/download.html`

[8]`http://ksl.stanford.edu/projects/owl-ql`

system would clearly benefit of a further optimisation of this phase in the query answering process, some of which were discussed in Section 2.4.

## References

[1] S. Bechhofer. The DIG Description Logic interface: DIG/1.1. Technical report, University of Manchester, Feb 2003.

[2] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language reference. Technical report, W3C, Feb 2004.

[3] A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1–2):353–367, 1996.

[4] R. Fikes, P. Hayes, and I. Horrocks. DQL - a query language for the semantic web. Technical report, Knowledge Systems Laboratory, Stanford University, 2002.

[5] R. Fikes, P. Hayes, and I. Horrocks. DAML Query Language (DQL) abstract specification. URL, `http://www.daml.org/2003/04/dql`, Apr 2003.

[6] R. Fikes, P. Hayes, and I. Horrocks. OWL Query Language (OWL-QL) abstract specification. URL, `http://ksl-web.stanford.edu/KSL_Abstracts/KSL-03-14.html`, Oct 2003.

[7] V. Haarslev and R. Möller. Racer system description. In *Automated Reasoning: First Int. Joint Conference, IJCAR*, volume 2083 / 2001 of *LNCS*, pages 701–705, Siena, Italy, Jun 2001. Springer-Verlag Heidelberg.

[8] V. Haarslev, R. Möller, and M. Wessel. RACER user's guide and reference manual, version 1.7.19. URL, `http://www.sts.tu-harburg.de/~r.f.moeller/racer/racer-manual-1-7-19.pdf`, Apr 2004.

[9] I. Horrocks. FaCT and iFaCT. In *Proc. of the International Workshop on Description Logics (DL'99)*, volume 22, pages 133–135, Linköping, Sweden, Jul – Aug 1999. CEUR Workshop Proceedings.

[10] S. Tessaris. *Questions and answers: reasoning and querying in Description Logic.* Phd thesis, University of Manchester, 2001.

[11] F. van Harmelen, P. F. Patel-Schneider, and I. Horrocks. Reference description of the DAML+OIL (march 2001) ontology markup language. URL, `http://www.daml.org/2001/03/reference`, Mar 2001.