Software Tools for Ontology Access, Processing, and Usage

Deliverable TONES-D21

D. Calvanese[1], P. Dongilli[1], G. De Giacomo[2], A. Kaplunova[5], D. Lembo[2], M. Lenzerini[2], R. Möller[5], R. Rosati[2], S. Tessaris[1], M. Wessel[5], I. Zorzi[1]

[1] Free University of Bozen-Bolzano
[2] Università di Roma "La Sapienza"
[3] The University of Manchester
[4] Technische Universität Dresden
[5] Technische Universität Hamburg-Harburg

Project:	FP6-7603 – Thinking ONtologiES (TONES)
Workpackage:	WP4 – Ontology Access, Processing, and Usage
Lead Participant:	Hamburg University of Technology
Reviewer:	—
Document Type:	Deliverable
Classification:	Consortium
Distribution:	TONES Consortium
Status:	Final
Document file:	D21_ToolsAccess.pdf
Version:	1.1
Date:	September 1, 2007
Number of pages:	26

Document Change Record		
Version	**Date**	**Reason for Change**
v.1.0	August 25, 2007	Draft
v.1.1	September 1, 2007	Final version

Contents

1 Introduction

A number of algorithms and techniques developed in the TONES project for tasks of ontology access, processing, and usage (see Deliverable D18 [CGG+07]) have been implemented in software tools presented in this report. Some of these tools (e.g., RacerPro) have been already delivered in the previous software package, TONES Deliverable D15 [CCF+07]. In the current software package we present the new versions of these tools with novel features and optimizations relevant for tasks of ontology access, processing, and usage. This report accomplishes the software package and gives an overview on software specific characteristics such as system requirements, installation instructions, basic configurations, formats of input and output, and short description of relevant features. We omit preliminaries and basics as well as syntax and semantics of Description Logics, which are described in detail in the previous deliverables, especially in Deliverables D13 [BBC+07] and D18 [CGG+07].

2 RacerPro

2.1 Introduction

RacerPro [HM01, EHK+07, EKM+07] is a knowledge representation system that implements a highly optimized tableau calculus for a very expressive description logic $\mathcal{SHIQ}(\mathcal{D}n)$ with several extensions (see e.g., TONES Deliverable D18 [CGG+07] for details about syntax and semantics of description logics). It offers reasoning services for multiple TBoxes and for multiple ABoxes as well. RacerPro provides for all standard DL inference services and includes several add-ons and some non-standard inferences useful for ontology development as well for accessing and processing ontologies being in operation. The list of new features of RacerPro for tasks of ontology design and maintenance can be found in TONES Deliverable D15 [CCF+07]. In this document, we concentrate on system features implemented to support online usage of ontologies.

Several interfaces are available for RacerPro. The reasoner supports file-based interaction as well as socket-based communication with end-user applications or graphical interfaces for ontology development and maintenance. Input can be specified in various syntaxes, e.g., KRSS (TCP), DIG 1.1 (HTTP), or OWL DL (HTTP). A parser for DIG 2.0 [TBK+06] is in preparation. As an extension to DIG 1.1, RacerPro already supports an XML-based interface for conjunctive queries. The specification of this interface is also proposed as part of DIG 2.0 with some slight modifications [TBK+06]. The RacerPro implementation of DIG 2.0 will also support expressive constraints. Unparsers from the internal meta model to a textual representation of ontologies are available for all syntaxes.

2.2 New Services for Ontology Access, Processing, and Usage

The most important extensions to the reasoning engine RacerPro that play a role for the tasks of ontology access, processing, and usage, are in particular:

- Support for reasoning with triples in secondary memory. In order to achieve scalability in various respects, RacerPro now allows for queries referring to triples stored in a so-called persistent triple-store. Ad-hoc queries can be answered with and without reasoning. The latter is possible also on secondary memory.

- Extensions to the RacerPro ABox query language. RacerPro supports grounded conjunctive queries [WM05]. The language implemented in RacerPro is called nRQL. The query language of RacerPro is extended with facilities for server-side programming to support post-processing of query results in the description logic inference system. In addition, head-projection and aggregation operators are defined that support frequently used post-processing requirements. Furthermore, combined TBox and ABox queries can now be specified.

In what follows, we give a short description and usage examples for services mentioned above.

2.3 Optimizations and Usage Examples

Support for Reasoning with Triples in Secondary Memory In practical applications, not all parts always require reasoning with expressive Tboxes (specified for instance in OWL ontologies). Indeed, for some purposes, classical data retrieval is just fine. However, in almost all practical applications there will be a large amount of data. Hence, data access must scale at least for the standard retrieval tasks. At the same time, it must be possible to apply expressive reasoning to (parts of) the data without producing copies of the data. For this purpose, we propose to use a triple store for RDF.

Very efficient software is available to query and manipulate RDF triple stores. RacerPro relies on one of the fastest triple store for billions of triples: AllegroGraph form Franz Inc. (`www.franz.com`). The AllegroGraph triple store is part of RacerPro-1.9.1-beta.

Using a triple store has several advantages. On the one hand, triples may be manipulated and retrieved from programs (Turing-complete representations), e.g. Java programs or Common Lisp programs, without reasoning as usual in industrial applications. On the other hand, the same triples can be queried w.r.t. a background a background ontology. This involves reasoning and might result in additional, implicit triples to be found.

RacerPro allows for accessing existing AllegroGraph triple stores as well as for the creation of new ones. In the following example, an existing triple store is opened, and the triples are read into the knowledge base. Afterwards three nRQL queries are answering over the knowledge base. There is no need to write long-winded data extraction programs that move triples to OWL files on which, in turn, reasoning is then applied.

```
(setf db (open-triple-store "test"))
(use-triple-store db :kb-name 'test-kb)

(retrieve (?x
           (:datatype-fillers (#!name ?x))
           (:datatype-fillers (#!emailAddress ?x))
           (:datatype-fillers (#!telephone ?x)))
  (and (?x #!Professor)
       (?x |http://www.Department0.University0.edu| #!worksFor)
       (?x (a #!name))
       (?x (a #!emailAddress))
       (?x (a #!telephone))))

(retrieve (?x ?y ?z)
  (and (?x ?y #!advisor)
       (?x ?z #!takesCourse)
       (?y ?z #!teacherOf)
       (?x #!Student)
       (?y #!Faculty)
       (?z #!Course)))

(retrieve (?x ?y)
  (and (?y |http://www.University0.edu| #!subOrganizationOf)
```

```
(?y #!Department)
(?x ?y #!memberOf)
(?x #!Chair)))
```

The examples should illustrate the flavor of how triples can be accessed from RacerPro. Currently, for reasoning, the triples are loaded into main memory by RacerPro. Thus, only a limited number of triples should be in the store. In a future version, reasoning will be done also on secondary memory.

A more comfortable querying of the triple store is possible using RacerPorter. In the example shown in Figure 1, the query asking for all chairs of the university departments is formulated in a SQL-like syntax:

```
select ?x where (?x rdf:type lubm:Chair)
```

The result tuples can be selected in a separate presentation tab (Query IO) of the editor as illustrated in Figure 2. Afterwards, e.g., the corresponding inferred relational (role filler) ABox structure can be viewed using one of the graph panes of RacerPorter (see Figure 3).

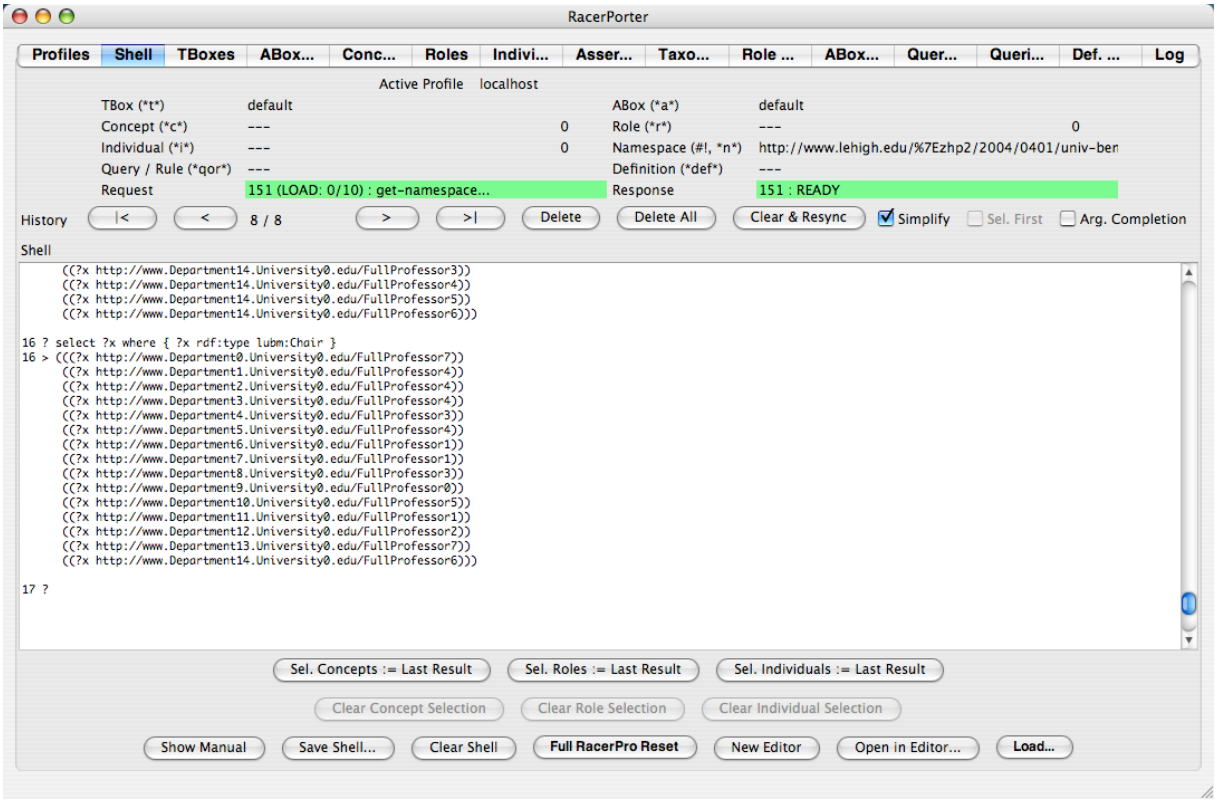

Figure 1: SPARQL Query

As mentioned before, it is not always the case that reasoning is required. Therefore, one can pose the same nRQL queries to secondary memory for very fast access (but without reasoning). RacerPro optimizes the queries in order to provide good average-case performance.

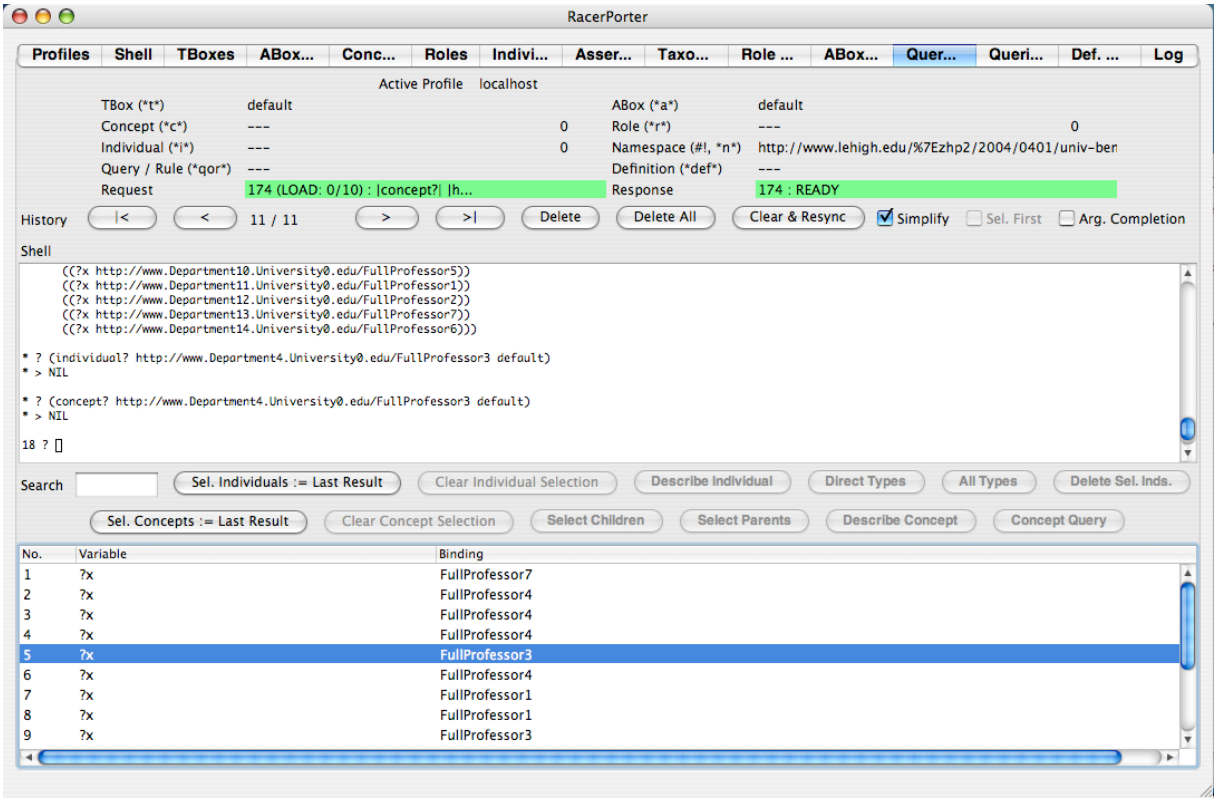

Figure 2: Query results

```
(open-triple-store "test")

(pretrieve (?x
           (:datatype-fillers (#!name ?x))
           (:datatype-fillers (#!emailAddress ?x))
           (:datatype-fillers (#!telephone ?x)))
  (and (?x #!Professor)
       (?x |http://www.Department0.University0.edu| #!worksFor)
       (?x (a #!name))
       (?x (a #!emailAddress))
       (?x (a #!telephone))))

(pretrieve (?x ?y ?z)
  (and (?x ?y #!advisor)
       (?x ?z #!takesCourse)
       (?y ?z #!teacherOf)
       (?x #!Student)
       (?y #!Faculty)
       (?z #!Course)))
```

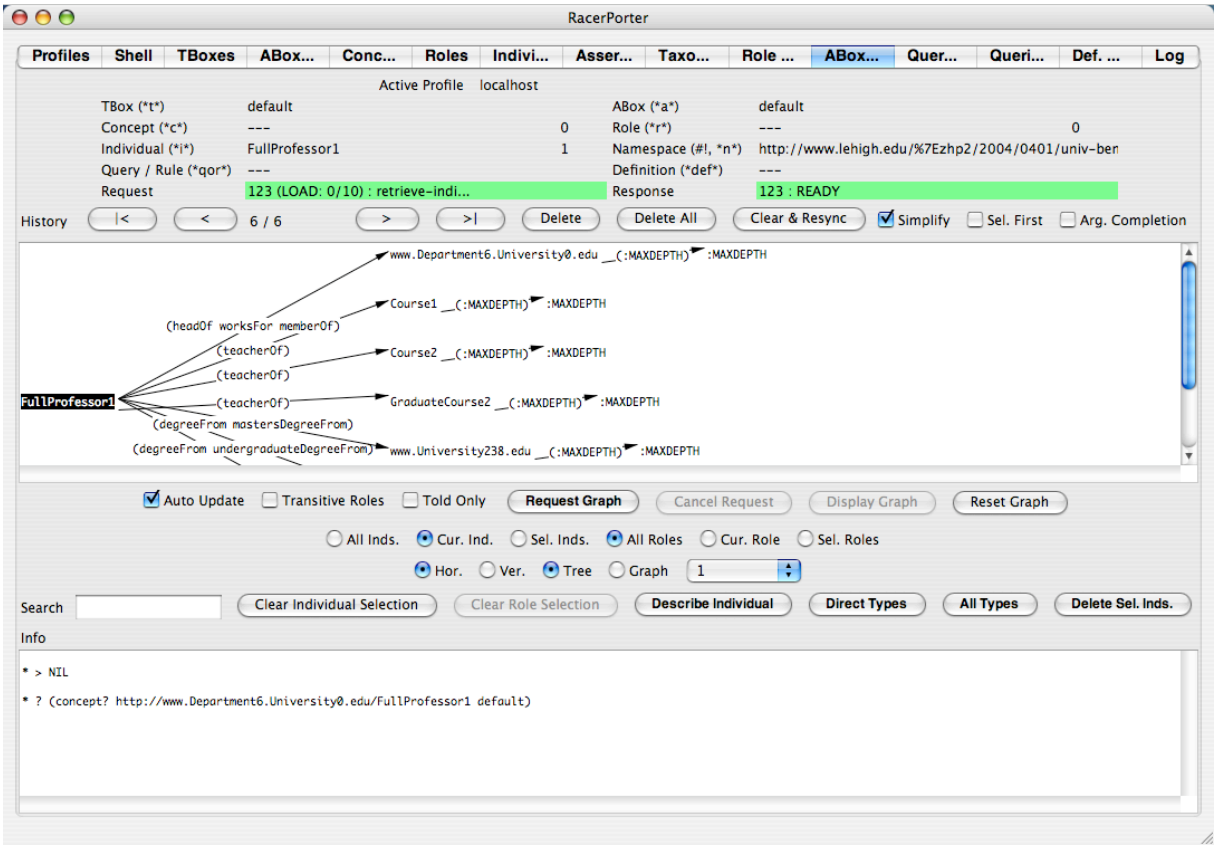

Figure 3: ABox Graph

```
(pretrieve (?x ?y)
  (and (?y |http://www.University0.edu| #!subOrganizationOf)
       (?y #!Department)
       (?x ?y #!memberOf)
       (?x #!Chair)))
```

In addition, it is possible to materialize in a triple store what can be computed by applying reasoning w.r.t. a background ontology such that later on the results are available to all applications which may or may not use reasoning. It is possible to optimized index data structures for subsequent query answering.

```
(minilisp-evaluate
     (let ((db (open-triple-store "test")))
       (use-triple-store db :kb-name 'test-kb :ignore-import t)
       (materialize-inferences 'test-kb :db db :abox t :index-p t)))
```

A triple store may be created by RacerPro as well.

```
(minilisp-evaluate
```

```
(let ((db (create-triple-store "test" :if-exists :supersede)))
  (triple-store-read-file "..." :db db)))
```

Ad-hoc Querying with SPARQL Queries can also be specified in the SPARQL language and can be executed with or without reasoning w.r.t. background ontologies.

```
(sparql-retrieve "
PREFIX lubm: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
SELECT ?x, ?y
WHERE
 ( ?x lubm:subOrganizationOf "http://www.University0.edu" )
 ( ?y rdf:type lubm:Department )
 ( ?x lubm:memberOf ?y )
 ( ?x rdf:type lubm:Chair )
")
```

This is the SPARQL pendant to the third query in the example above.

Note that there is no need to store the OWL ontology in the triple store. With RacerPro you can access your existing triple store with various different OWL ontologies. An ontology can be put into a file or can possibly be retrieved from the Web. A triple store is then opened an queries are answered w.r.t. this triple store. The triple store corresponds to the ABox in this case.

Server-side Programming in nRQL The native query language of RacerPro, nRQL, is extended by means of a simple expression language called "MiniLisp". With MiniLisp a user can write a simple, termination safe, "program" which is executed on the RacerPro server. Such MiniLisp programs can be used to improve the flexibility of the query answering. MiniLisp allows a restricted kind of server-side programming and can thus be used to implement, for example

1. user-defined output formats for query results (e.g., the query results can also be written into a file),
2. certain kinds of combined ABox/TBox queries,
3. efficient aggregation operators (e.g., sum, avg, like in SQL),
4. user-defined projection operators.

Here are some MiniLisp examples which demonstrates the points 1. to 3. Consider the following simple ABox:

```
(related i j r)
(related j k r)
```

Suppose you want to create a *comma separated values file* called `test.csv` which contains all the (possible implied) `R` role assertions. MiniLisp allows you to do this:

```
(retrieve (((:lambda (x y)
               (with-open-output-file ("~/test.csv")
                   (format *output-stream* "~A;~A~%" x y)))
            ?x ?y))
      (?x ?y r))
```

So, the query body retrieves all `?x, ?y` individuals which stand in an `R` relationship; for each `?x, ?y` tuple, one more line is attached to the file `test.csv`.

Regarding point 2, let us illustrate how "combined" TBox/ABox queries can be used to retrieve the *direct* instances of a concept, which has been requested by many users.

Let us create two concepts `c` and `d` such that `d` is a sub concept (child concept) of `c`:

```
? (full-reset)
> :okay-full-reset

? (define-concept c (some r top))
> :OKAY

? (define-concept d (and c e))
> :OKAY
```

We can verify that `d` is indeed a child concept of `c`, using a so-called TBox query:

```
? (tbox-retrieve (?x) (c ?x has-child))

> (((?x d)))
```

Let us create two individuals so that `i` and `j` are instances of `c`; moreover, `j` is also an instances of `d`:

```
? (related i j r)
> :OKAY

? (related j k r)
> :OKAY

? (instance j e)
> :OKAY

? (retrieve (?x) (?x c))
> (((?x j)) ((?x i)))

? (retrieve (?x) (?x d))
> (((?x j)))
```

Thus, both `i` and `j` are `c` instances. However, only `i` is a *direct* `c` instance. We can retrieve these direct instances of `c` as follows:

```
? (retrieve1 (?x c)
      ( ((:lambda (x)
          (if (some (lambda (subclass)
                      (retrieve () `(,x ,subclass)))
                    (flatten
                      (tbox-retrieve1 `(c ?subclass has-child)
                        `( ((:lambda (subclass) subclass) ?subclass)))))
              :reject
            `(?x ,x)))
        ?x)))

> (((?x i)))
```

Basically, `retrieve1` is like `retrieve`, but first comes the body, and then the head. Thus, `?x` is bound to a `c` instance. Using this binding, it is checked by means of a TBox subquery (`tbox-retrieve1`) whether the individual bound to `?x` is also an instance of any subclass of `c`. If this is the case, the result tuple (resp. the current binding of `?x`) is *rejected* (see the special `:reject` token); otherwise, the result tuple is *constructed* and returned. The returned result tuples make up the final result set.

Please note that the combination of MiniLisp and `:reject` token gives you the ability to define arbitrary, used-defined *filter predicates* which are executed efficiently since they are directly on the RacerPro server.

Finally, let us consider how *aggregation operators* can be implemented in MiniLisp. Consider the following book store scenario:

```
(full-reset)

(instance b1 book)
(instance b2 book)

(related b1 a1 has-author)
(related b1 a2 has-author)
(related b2 a3 has-author)

(define-concrete-domain-attribute price :type real)
(instance b1 (= price 10.0))
(instance b2 (= price 20.0))
```

We can now determine the number of authors of the single books with the following query:

```
? (retrieve (?x
              ((lambda (book)
```

```
                (let ((authors
                        (retrieve '(?a) '(,book ?a has-author))))
                  (length authors)))
              ?x))
      (?x book)
      :dont-show-lambdas-p t)

> (((?x b1) 2) ((?x b2) 1))
```

Another interesting question might be to ask for the *average price* of all the books. This is a little bit more tricky, but works in nRQL as well:

```
? (retrieve
      (((lambda nil
         (let ((prices
                (flatten
                  (retrieve '((told-value-if-exists (price ?x)))
                            '(?x book)
                           :dont-show-head-projection-operators-p t))))
           '(average-book-price
               ,(float (/ (reduce '+ prices) (length prices))))))))
     true-query
    :dont-show-lambdas-p t)

> (((average-book-price 15.0)))
```

The trick here is to use the always true query body `true-query` in order to let nRQL first evaluate the lambda body (since lambda bodies are only valid in nRQL heads, but not available as "first order statements" in RacerPro); thus, the lambda simply acquires all the prices of the individual books and then computes and returns the average price, as expected. However, there is also a function `minilisp-evaluate` for exactly this purpose.

Often, user-defined query format output must be generated from query results, e.g., HTML reports. This is easy with MiniLisp as well. For example, the result of the query

```
(retrieve (?x ?y) (and (?x #!person) (?x ?y #!has_pet) (?y #!cat)))
```

on the famous `people+pets.owl` KB is

```
(((?x http://cohse.semanticweb.org/ontologies/people#Fred)
  (?y http://cohse.semanticweb.org/ontologies/people#Tibbs))
 ((?x http://cohse.semanticweb.org/ontologies/people#Minnie)
  (?y http://cohse.semanticweb.org/ontologies/people#Tom)))
```

Suppose this result shall be presented as an HTML table, together with some additional information about the query which has been posed. In principle, it is easy to generate a HTML file using `with-open-output-file` and print (`format`) statements to write some HTML. However, this results in ugly code. For this reason, some syntactic sugar is available. The following query will generate an HTML file `minilisp.html` which is shown in Figure 4:

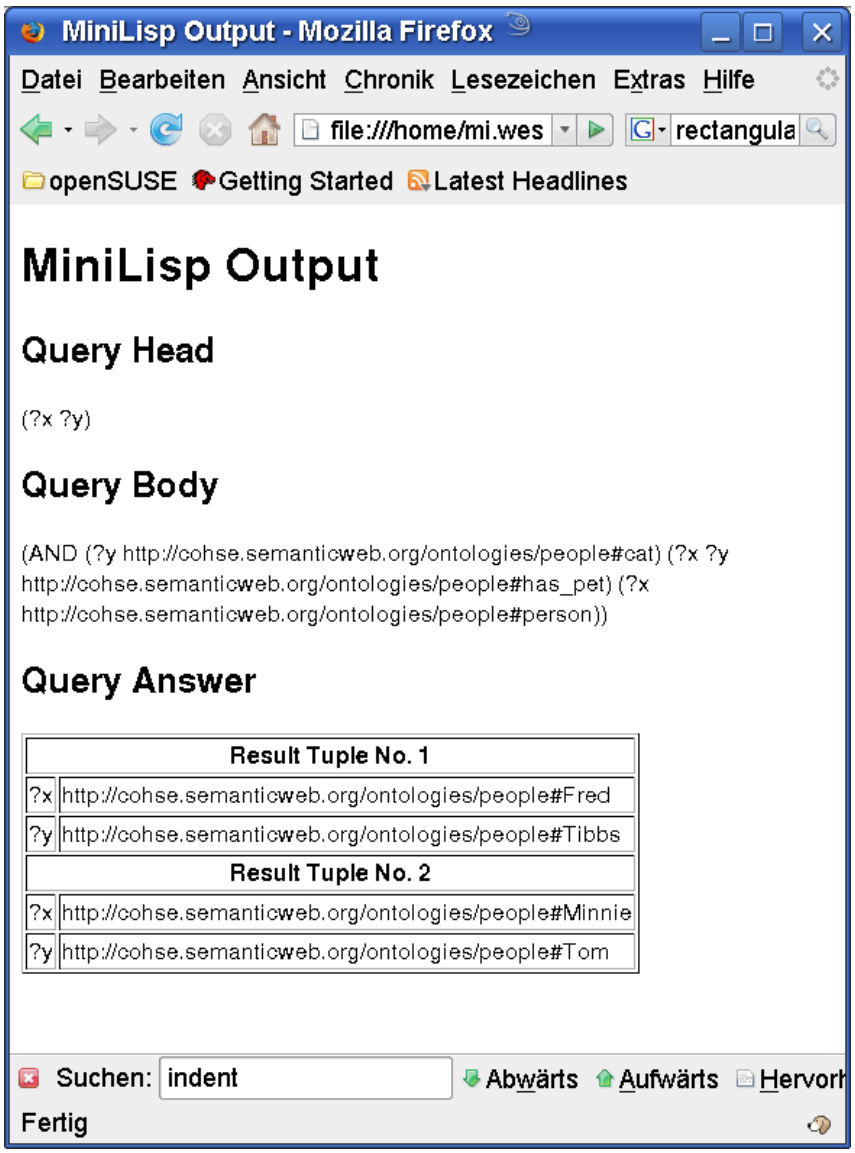

Figure 4: Generated HTML Page

```
(minilisp-evaluate
      (let ((res
              (retrieve '(?x ?y)
                        '(and (?x \# !"person")
                              (?x ?y \# !"has_pet")))))
        (with-html ("minilisp.html")
          (html (head) (html (title) (content "minilisp output")))
          (html (body)
              (html (h1) (content "minilisp output"))
              (html (h2) (content "query head"))
              (content (query-head :last))
```

```
(html (h2) (content "query body"))
(content (query-body :last))
(html (h2) (content "query answer"))
(html
 (table :border 1)
 (let ((count 0))
   (maplist (lambda (bindings)
              (html
               (tr)
               (html
                (th :colspan 2)
                (content
                 (format nil
                         "result tuple no. ~a"
                         (incf count)))))
              (maplist (lambda
                         (var-val)
                         (let
                          ((var (first var-val))
                           (val (second var-val)))
                          (html
                           (tr)
                           (html (td) (content var))
                           (html (td) (content val)))))
                       bindings))
            res)))))))
```

2.4 Installation

RacerPro is being delivered using a certain deployment system (Allegro Common Lisp). However, RacerPro can be obtained also for another deployment system (LispWorks) which works slightly different than ACL. E.g., it allows one-file executables. The LW-based RacerPro kernel executable is available as a ZIP-compressed executable. Because this program comes without installer and support files you have to do the installation process yourself. If you are in doubt which version to use, try the installer version.

RacerPro supports several command line options (see RacerPro reference manual). The ACL-based RacerPro for the Windows platform has command line options which work slightly different than documented in the manuals. E.g., you can invoke help by starting `RacerPro +p -- -help` and for using RacerPro in a batch file (without console) you can use `RacerPro +c -- -f [input.racer] -q [query.racer] -o [output.racer]`. To make RacerPro a more convenient processor for use within batch files or from the command line the Windows installer also adds the installation directory to the PATH system environment variable.

There is a new debug option enabled in the latest release which will write some internal status information into a file. The contents of the file may help the developers

supporting you if you report a problem. To switch the debug option on RacerPro is started as follows: `./RacerPro -log ‘‘[path-to-logfile]’’ -debug;` or under Windows `RacerPro.exe -- -log ‘‘[path-to-logfile]’’ -debug`. If you do not provide a filename for the logfile, then the output is written into “racer.log” next to the RacerPro executable.

RacerPro opens TCP ports 8080 and 8088 to communicate with remote clients like RacerPorter or Protégé. If the preset port numbers interfere with other services on your machine or network, please consult the manuals how to change the setting with the help of command line options.

The latest release of RacerPro also includes RacerPorter. RacerPorter is a new part of the RacerPro system and works as a stand-alone application.

This version of RacerPro as well as RacerPorter includes associations with the file types .racer and .owl on the Windows and Mac OS X platforms and the installer packages include information about the file types and icons.

Third party tools which utilize the DIG interface (e.g., Protégé) might use a Java-based converter from OWL to DIG. The converter approximates OWL by translating it to DIG but the semantics may be changed. If the same OWL knowledge base is loaded directly into RacerPro, the semantics of OWL are obeyed (indeed, with very few restrictions). This difference in loading a knowledge base may cause differing answers when queries are compared.

The installer packages for Windows and Mac OS X also include a folder of example files which will help following the explanations in the user guide.

3 QuOnto

In this section we provide a description of the tool QuOnto (Querying Ontologies), a reasoner for the DLs of the *DL-Lite* family [CDGL$^+$07]. We will mention relevant tasks for ontology access addressed by QuOnto and briefly describes algorithms that it implements.

3.1 Introduction

QuOnto [1] is a free (for non-commercial use) Java-based reasoner for *DL-Lite* with GCIs.

QuOnto is able to manage a large amount of concept and role instances (from thousands to millions) through relational database technology, and implements a query rewriting algorithm for both consistency checking and query answering of complex queries (unions of conjunctive queries) over *DL-Lite* knowledge bases, whose ABox is managed through relational database technology. Currently, it supports its own Java-based interface, and accepts inputs in a proprietary XML format.

3.2 Features and Optimizations

TBox Specification in QuOnto As already said, Knowledge Bases (KBs) managed in QuOnto are specified in the DLs of the *DL-Lite* family. DLs of this family are able to capture the main notions of conceptual modeling formalism used in databases and software engineering such as ER and UML class diagrams. Basically, *DL-Lite* assertions allow for specifying (in a controlled way) *ISA* and *disjointness* between concepts and roles, *role-typing*, *participation* and *non-participation constraints* between a concept and a role, *functionality restrictions* on roles, *attributes* on roles and concepts. The DLs of the *DL-Lite* family differ one another for the kind of assertions they allow (among those mentioned above), and for the way in which such assertions can be combined. All such DLs, however, allow for tractable reasoning. Notably, answering unions of conjunctive queries over *DL-Lite* KBs is in LOGSPACE in data complexity, i.e., the complexity measured only w.r.t. the size of the ABox, and the tuning in the use of the assertions in each DL of the *DL-Lite* family is aimed at guaranteeing such a nice computational behavior.

We do not provide here details on the syntax and semantics of the DLs of the *DL-Lite* family, and refer the reader to [CDGL$^+$07] and TONES Deliverables D06 [BCG$^+$06], D08 [CCD$^+$06], and D13 [BBC$^+$07] for an in depth and formal description of these matters. We only point out that in QuOnto the TBox is provided in a proprietary XML format. Instructions on how to produce a QuOnto XML TBox, i.e., the DTD for the TBox, and examples can be found in the QuOnto distribution directory (see the software CD delivered together with the present report) following the paths `./quonto/resources/DTDs/TBox.dtd` and `./quonto/docs/XMLTBox.html`.

ABox Specification in QuOnto In QuOnto, the extensional level of the knowledge base is a *DL-Lite ABox*, i.e., a set of plain *membership assertions*. For example, for DLs

[1] http://www.dis.uniroma1.it/ quonto/

of the *DL-Lite* family that do not allow for the specification of attributes on concepts and roles, an ABox is a set of assertions of the form

$$A(c), R(c, b),$$

where A is an atomic concept, R is an atomic role, c and b are constants. These assertions state respectively that the object denoted by c is an instance of the atomic concept A, and that the pair of objects denoted by (c, b) is an instance of the atomic role R.

One of the distinguishing features of QuOnto is that the ABox is stored under the control of a DBMS, in order to effectively manage objects in the knowledge base by means of an SQL engine. To this aim, QuOnto constructs a relational database which faithfully represents an ABox $\mathcal{A}$: for each atomic concept A, a relational table tab_A of arity 1 is defined, such that $\langle c \rangle \in tab_A$ iff $A(c) \in \mathcal{A}$, and for each role R, a relational table tab_R of arity 2 is defined, such that $\langle c, b \rangle \in tab_R$ iff $R(c, b) \in \mathcal{A}$ (analogously in the presence of membership assertions involving concept or role attributes).

We point out that the above construction is completely transparent for the user, and that ABoxes in input to QuOnto are simply sets of plain membership assertions, represented in a proprietary XML format. Instructions on how to produce a QuOnto XML ABox, i.e., the DTD for the ABox, and examples can be found in the QuOnto distribution directory (see the software CD delivered together with the present report) following the paths `./quonto/resources/DTDs/ABox.dtd` and `./quonto/docs/XMLABox.html`

Query Answering in QuOnto In order to take advantage of the fact that the ABox is managed in secondary storage by a Data Base Management System (DBMS), the query answering algorithm used in the QuOnto system is based on the idea of reformulating the original query into a set of queries that can be directly evaluated by an SQL engine over the ABox. Note that this allows us to take advantage of well established query optimization strategies.

Query reformulation is therefore at the heart of our query answering method. The basic idea of our method is to reformulate the query taking into account the TBox: in particular, given a union of conjunctive queries q over a *DL-Lite* knowledge base $\mathcal{K}$, we compile the assertions of the TBox into the query itself, thus obtaining a new union of conjunctive queries q'. Such a new query q' is then evaluated over the ABox of $\mathcal{K}$, that is over the relational database representing the ABox. Since the size of q' does not depend on the ABox, the data complexity of the whole query answering algorithm is LOGSPACE in data complexity (i.e., the data complexity of evaluating a union of conjunctive queries over a database instance). We refer the reader to [CDGL$^+$07] for more details on the query answering algorithm implemented in QuOnto. We simply point out here that our tool is also equipped with some optimization techniques that aim at "minimizing" each disjunct occurring in the rewritten query q', i.e., each disjunct in the query q' is further rewritten in order to drop some of its atoms to avoid useless join computations.

3.3 Installation and Usage

3.3.1 System requirements

In order to get the QuOnto system propertly working you need the following software products correctely installed on your machine:

- JAVA SDK 1.5[2]
- MySQL 5.0 Server and CLient

and the following java libraries available through the CLASSPATH environment variable

- MySQL Java Connector

In order to run the QuOnto system you have to create the 'quonto' user, with full rights, on the MySQL database.

Issue the following statements in order to create it:

```
CREATE USER 'quonto' IDENTIFIED BY 'quonto';
GRANT ALL PRIVILEGES ON *.* TO 'quonto'@'localhost' identified by 'quonto';
FLUSH PRIVILEGES;
```

3.3.2 Installing and running QuOnto

In order to install the QuOnto system simply extract the distribution to any directory. The structure of the extracted directory is necessary for the software to work correctly. Such a directory is structured as follows:

```
quonto - the QuOnto system directory
 |
 |
 +--resources - contains all the resources needed by QuOnto
 |   |
 |   +--inputs
 |   |    |
 |   |    +--own ontologies - contains the input XML file of DL-Lite TBox
 |   |    |
 |   |    +--queries XML - contains the input XML files for the user's queries
 |   |
 |   +--DTDs - contains the Document Type Definitions for the XML input files
 |        |
 |        +--TBox.dtd - the DTD for the TBox XML specification
 |        |
 |        +--UCQ.dtd - the DTD for the user's queries specification
 |
 +--doc - contains the documentation of the QuOnto system
 |   |
```

[2]The QuOnto system DOES NOT work with JAVA SDK 1.6.

```
|   +--manual.doc - this document
|   |
|   +--license.pdf - the free trial copy software agreement
|   |
|   +--XMLontologies.html - a short guide on writing  DL-Lite ontologies in XML
|   |
|   +--XMLqueries.html - a short guide on writing union of conjunctive queries in XML
|
+--quonto.jar - the QuOnto implementation library
|
+--quonto - the batch file for the QuOnto execution on Windows systems
```

The QuOnto distribution includes simple examples of TBox, ABox, and queries in the directory `./resources/inputs/`.

3.3.3 Usage

QuOnto performs query answering of union of conjunctive queries in *DL-Lite*$_{\mathcal{F}}$ ontologies (TBox and ABox). It implements the technique described in the deliverable D18. In particular it stores the ABox assertions of the *DL-Lite*$_{\mathcal{F}}$ ontology in a relational database (managed by DBMS MySQL in the present QuOnto distribution), and performs query answering over the *DL-Lite*$_{\mathcal{F}}$ ontology (TBox and ABox) in two steps:

1. *Expansion step:* it rewrites the input union of conjunctive queries, which is a query over the *DL-Lite*$_{\mathcal{F}}$ ontology, as a new union of conjunctive queries, which is a query over the ABox only, seen as a standard relational database;

2. *Evaluation step:* translate in SQL and evaluate the resulting query over the database storing the ABox assertions.

In the current version all interactions are based on textual inputs on a command shell.

Query Expansion Both the TBox and the query must be specified in XML according to the DTDs provided. In order to run the query expansion step only the user issues the following command:

```
quonto.bat -t tboxFile.xml -q queryFile.xml
```

The system parses the input files (validating them by means of the DTDs provided) and will produce the rewriting by printing it on the shell output according to the Datalog syntax for union of conjunctive queries. A typical output generated by a QuOnto execution follows:

```
Parsing TBox specification file ...
Parsing query specification file ...
Expanded query:
q(X) :- man(X)
q(X) :- person(X)
 ... thank you for trying QuOnto!!!
```

In the text above, the expanded query is constituted by the two lines

```
q(X) :- man(X)
q(X) :- person(X)
```

where each line represents a disjunct of the union of conjunctive queries produced by the rewriting process. Notice that QuOnto looks into the "./resources/inputs/own ontologies" folder for the TBox XML input file, whereas it looks into the "./resounces/inputs/queries XML" folder for the query input file.

Query Answering In order to perform query answering in the QuOnto system the user issues the following command:

```
quonto.bat -t tboxFile.xml -a aboxFile.xml -q queryFile.xml evaluate
```

QuOnto performs query evaluation and writes the answer in a file called `result.xml`

4 Query Tool

4.1 Introduction

The query tool is meant to support a user in formulating a precise query – which best captures her/his information needs – even in the case of complete ignorance of the vocabulary of the underlying information system holding the data. The intelligence of the interface is driven by an ontology describing the domain of the data in the information system. The final purpose of the tool is to generate a conjunctive query ready to be executed by some evaluation engine associated to the information system.

4.2 Installation and Usage

The system requires at least JRE 1.4 and a DL reasoner providing a DIG 1.x interface.

The query interface is provided with four Tabs:

- Admin: administrative interface used to load the ontology and connect to the reasoner.
- Compose: main query composition interface.
- Query: displays the actual query, mainly for debugging purposes.
- Results: displays the results of the query evaluation.

Initially the user is presented with the *Admin* Tab (see Figure 5). Here, some preliminary operations necessary for the query formulation have to be executed:

1. **Connection setup**: one of the operations the user has to carry out consists in testing the reasoner connection. A reasoner with reference to the ontology is used by the system to drive the query interface: in particular, it is used to discover the terms and properties which are proposed to the user to manipulate the query.
2. **Loading and managing ontology files**: all the operations the system provides cannot be accomplished without loading an ontology. The interface allows the user to specify an ontology in DIG 1.x format to be loaded into the system. Once the ontology is loaded into the system, the user has also the possibility to adjust the content of that ontology, depending on her/his needs; if the user wants that the modifications take a permanently effect, he/she can save them back to the file. As a matter of fact, users might frequently have the necessity to extend an ontology in order to obtain different results or to correct it as a consequence of unexpected behaviour.
3. **Loading a metadata file**: the interface gives the possibility to the user to specify a metadata file to be loaded into the system. Metadata files contain valuable information about the terms in the ontology; that information concerns essentially the lexicalizations of those terms. Actually, as the terms contained in the ontology could be expressed by a sort of shorthand, their lexicalizations are provided so that the user can deal with clearly understandable terms.

4. **Customising lexicalizations**: given the metadata file, the interface should offer to the user the opportunity to apply desired variations to the lexical information of the terms. Those variations can be saved back to a metadata file or just saved temporarily in the system. The query to be generated should be as unambiguous as possible: if the user can assign to the terms the lexicalizations which best give significant importance to him/her, the query formulation will be transparent and therefore the really intended result will be retrieved.

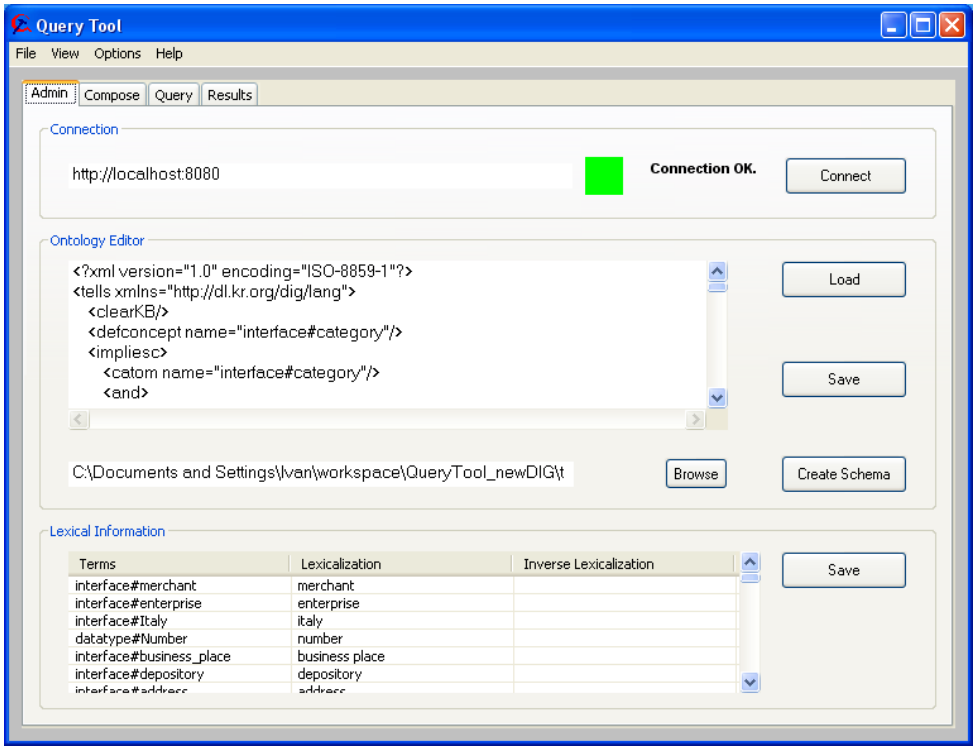

Figure 5: Administrative interface of the Query Tool.

As you can see in Figure 5, the reasoner connection has been tested by means of the "Connect" button. An ontology has been loaded ("Load" button) and also a metadata file ("Browse" button). Subsequently, the "Create Schema" button has been clicked and all the lexicalizations of the ontology terms are presented in the "Lexical Information" table. Here the user can change the lexicalizations by clicking on the cell corresponding to the lexicalization he/she wants to modify.

In the *Compose* Tab (see Figure 6) the user can formulate the query by means of pop-up menus presenting the possible operations. Initially the user is presented with a choice of different starting terms (all the concepts in the ontology or a subset defined by means of the metadata file): he/she selects the first term to be added in the query. Subsequently, the interface gives the possibility to perform the following operations:

- **Add compatible terms**: other terms specified in the ontology can be added to the query. The compatible terms are automatically suggested to the user by means of appropriate reasoning tasks on the ontology describing the data sources. Indeed, the

system suggests only the operations which are compatible with the current query expression.

- **Substitute terms**: the system gives the opportunity of substituting the selected term of the query with a more specific or more general term. It can be also the case that in the ontology there are terms which are equivalent to the selected one: in this case the user is offered to replace the selection with an equivalent term.
- **Delete terms**: as the query is specified through an iterative refinement process, it could be the case that the user needs to delete some terms from the query.
- **Add or delete properties**: analogously, the user can add properties to the query. A property can be a relation or an attribute. The interface suggests a list with the possible alternatives. The user can specify some restriction values to attributes.

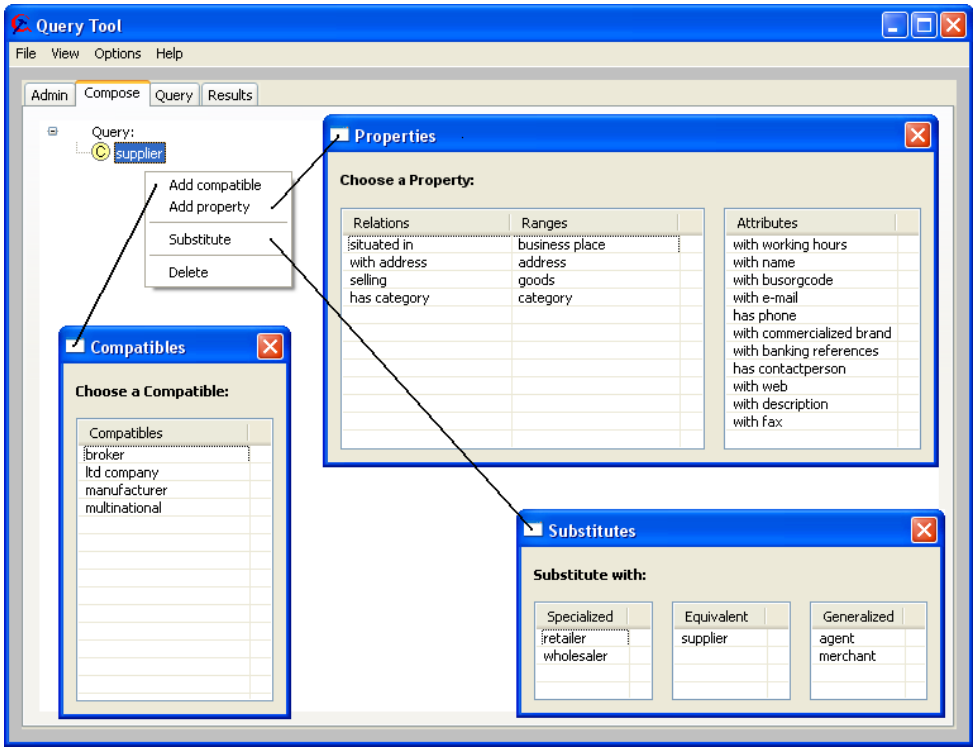

Figure 6: Query composition interface.

The first operation to compose a query consists in selecting the starting term. By clicking on a pop-up menu ("Choose starting term") the user is presented with a windows showing all the terms that can be used as starting term.

Once the user has selected the starting term, it is possible to refine the query using again the pop-up menu. The operations allowed are listed in the pop-up menu; the user can add a compatible term, add a property (relation or attribute), substitute the term or delete it.

If the user selects an attribute, it is possible to set it as distinguished variable or to add a restriction to the attribute. The user can also delete properties or terms from the query and select new ones.

Once the user has formulated the query, the *Query* tab shows the query in XML and DIG formats (the menu bar "Options" allows also to view the query in the corresponding SQL code). Finally, in the *Results* tab the user can retrieve the results (if any) corresponding with the formulated query.

In the menu bar, by clicking on "View" menu, the user can have a look to the log file ("View" log) of the application and also a concise description of the schema with all the taxonomy ("View" schema).

References

[BBC+07] F. Baader, R. Bernardi, D. Calvanese, A. Calì, B. Cuenca Grau, M. Garcia, G. D. Giacomo, A. Kaplunova, O. Kutz, D. Lembo, M. Lenzerini, L. Lubyte, C. Lutz, M. Milicic, R. Möller, B. Parsia, R. Rosati, U. Sattler, B. Sertkaya, S. Tessaris, C. Thorne, and A.-Y. Turhan. Techniques for Ontology Design and Maintenance. Project Deliverable D13, TONES, 2007. `http://www.tonesproject.org/`.

[BCG+06] F. Baader, D. Calvanese, G. D. Giacomo, P. Fillottrani, E. Franconi, B. Cuenca Grau, I. Horrocks, A. Kaplunova, D. Lembo, M. Lenzerini, C. Lutz, R. Möller, B. Parsia, P. Patel-Schneider, and R. Rosati. Formalisms for Representing Ontologies: State of the Art Survey. Project Deliverable D06, TONES, 2006. `http://www.tonesproject.org/`.

[CCD+06] D. Calvanese, B. Cuenca Grau, G. De Giacomo, E. Franconi, I. Horrocks, A. Kaplunova, D. Lembo, M. Lenzerini, C. Lutz, D. Martinenghi, R. Möller, R. Rosati, S. Tessaris, and A.-Y. Turhan. Common Framework for Representing Ontologies. Project Deliverable D08, TONES, 2006. `http://www.tonesproject.org/`.

[CCF+07] D. Calvanese, B. Cuenca Grau, E. Franconi, I. Horrocks, A. Kaplunova, C. Lutz, R. Möller, B. Sertkaya, S. Tessaris, and A.-Y. Turhan. Software Tools for Ontology Design and Maintenance. Project Deliverable D15, TONES, 2007. `http://www.tonesproject.org/`.

[CDGL+07] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *J. of Automated Reasoning*, 2007. To appear.

[CGG+07] D. Calvanese, G. D. Giacomo, B. Glimm, B. Cuenca Grau, V. Haarslev, I. Horrocks, A. Kaplunova, D. Lembo, M. Lenzerini, C. Lutz, M. Milicic, R. Möller, R. Rosati, U. Sattler, and M. Wessel. Techniques for Ontology Access, Processing, and Usage. Project Deliverable D18, TONES, 2007. `http://www.tonesproject.org/`.

[EHK+07] S. Espinosa, V. Haarslev, A. Kaplunova, A. Kaya, S. Melzer, R. Möller, and M. Wessel. Reasoning Engine Version 1 and State of the Art in Reasoning Techniques. Technical report, Hamburg University Of Technology, 2007. BOEMIE Project Deliverable D4.2.

[EKM+07] S. Espinosa, A. Kaya, S. Melzer, R. Möller, T. Näth, and M. Wessel. Reasoning Engine Version 2. Technical report, Hamburg University Of Technology, 2007. BOEMIE Project Deliverable D4.5.

[HM01] V. Haarslev and R. Möller. RACER System Description. In R. Goré, A. Leitsch, and T. Nipkow, editors, *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, Siena, Italy*, pages 701–705. Springer-Verlag, 2001.

[TBK+06] A.-Y. Turhan, S. Bechhofer, A. Kaplunova, T. Liebig, M. Luther, R. Moeller, O. Noppens, P. Patel-Schneider, B. Suntisrivaraporn, and T. Weithoener. DIG 2.0 – Towards a Flexible Interface for Description Logic Reasoners. In B. Cuenca Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *OWL: Experiences and Directions 2006*, 2006.

[WM05] M. Wessel and R. Möller. A High Performance Semantic Web Query Answering Engine. In *Proc. of the 2005 Description Logic Workshop (DL 2005)*, pages 84–95. CEUR Electronic Workshop Proceedings, http://ceur-ws.org/, 2005.