# Release Notes for RacerPro 1.9.2 beta

Racer Systems GmbH & Co. KG

## 1 Introduction

A new version of Racer is available now as RacerPro-1.9.2-beta. This version requires a beta tester license, which can be obtained from `www.racer-systems.com`. The new version contains the following improvements

- Performance improvements for the reasoning engine. For instance, RacerPro 1.9.2 can classify a version of Snomed (> 379000 concept names) in 13 minutes (Intel x86, 32bit, 2.4 GHz, 4GB, Mac OS X). It can also handle Aboxes with hundreds of thousands of individuals (see below for results on LUBM and UOBM) benchmarks.
- Performance improvements for reading OWL files
- Support for specific locales and external formats (e.g., for Asian character sets)
- New graphical interface (RacerPorter) for managing multiple RacerPro servers
- Enhancement to the nRQL query language
- Support for reasoning with triples stored in an AllegroGraph triple store
- Bug fixes

In addition to these improvements there are some other new features supported with version 1.9.2-beta.

- Namespaces as found in XML files can be declared to Racer as in `(define-prefix lubm "http://www.lehigh.edu/univ-bench.owl#")`. Then, LUBM concept, role and individual names can be referred to with `#!lubm:` as in `#!lubm:Person`. Querying OWL KBs with nRQL is much more convenient using prefixes. The `#!:` prefix macro can be used to abbreviate the last namespace declared with `define-prefix`. So, if `lubm` is declared last, `#!lubm:Person` can be written as `#!:Person`.
- Syntax support for OWL 1.1: RacerPro can load ontologies specified in OWL 1.1 format. The axioms beyond $\mathcal{SHIQ}(\mathcal{D}_n)$ are ignored or approximated in this beta version.
- Rule formalism extended to support abduction
- Event recognition facility

- Extensions to DIG 1.1 (called DIG 1.2)
- Support for some non-standard inference services (LCS, MSC-k)

These new features are shortly described in the next sections.

# 2 Performance improvements

## 2.1 Tbox classification

The classification engine of RacerPro has been continuously improved. For instance, RacerPro 1-9-2 can classify a version of Snomed with more than 379000 concept names in 13 minutes (Intel CPU, 2.4 GHz, 32bit, 4GB RAM, Mac OS X). For classifying other knowledge bases, RacerPro shows the same performance as systems dedicated only towards classifying ontologies (Tboxes). RacerPro is one order of magnitude faster than systems which – like RacerPro – also offer support for Abox reasoning w.r.t. expressive Tboxes.

## 2.2 Instance retrieval

The significance of the optimization techniques introduced in the new release is analyzed with the well-known LUBM benchmark. The runtimes we present in this section are used to demonstrate the order of magnitude of time resources that are required for solving inference problems. They allow us to analyze the impact of the implemented optimization techniques.

An overview of the size of the LUBM benchmarks is given in Figure 1. With an increasing number of universities, there is a linearly increasing number of instances as well as concept and role assertions. For instance, with 50 universities, 1.000.000 instances have to be handled.

The runtimes for answering all 15 LUBM queries are presented in Figures 2 and 3 (Sunfire, Solaris, 32 GB). In Figure 2 a version of the LUBM TBox is used that does not cause backtracking during ABox satisfiability (or consistency) tests . With this kind of benchmark, we can evaluate storage management and indexing techniques of DL provers. In Figure 3 we used a variant of the TBox that causes backtracking.

Before queries can be answered, the ABox is checked for consistency (see the "Consistency" curve). As can be expected, if there is no backtracking state-of-the-art provers are very fast (Figure 2). If backtracking is required (Figure 3) runtimes for ABox consistency checking increase. ABox consistency checking can be done offline and corresponds to computing index structures in a database system.

Comparing the runtimes for query answering in Figures 2 and 3 (see the corresponding curves "Queries") reveals that backtracking does not influence query answering to a large extent (at least not in the LUBM case we investigated). The total runtime is indicated with "Total".

The results we achieve were possible with dedicated storage management techniques (e.g., offered by the implementation language Franz Allegro Common Lisp). With this basis it is possible to declare that all data structures for storing the LUBM TBox, ABox, and index structures are not examined by the garbage collector. If this is not done, garbage

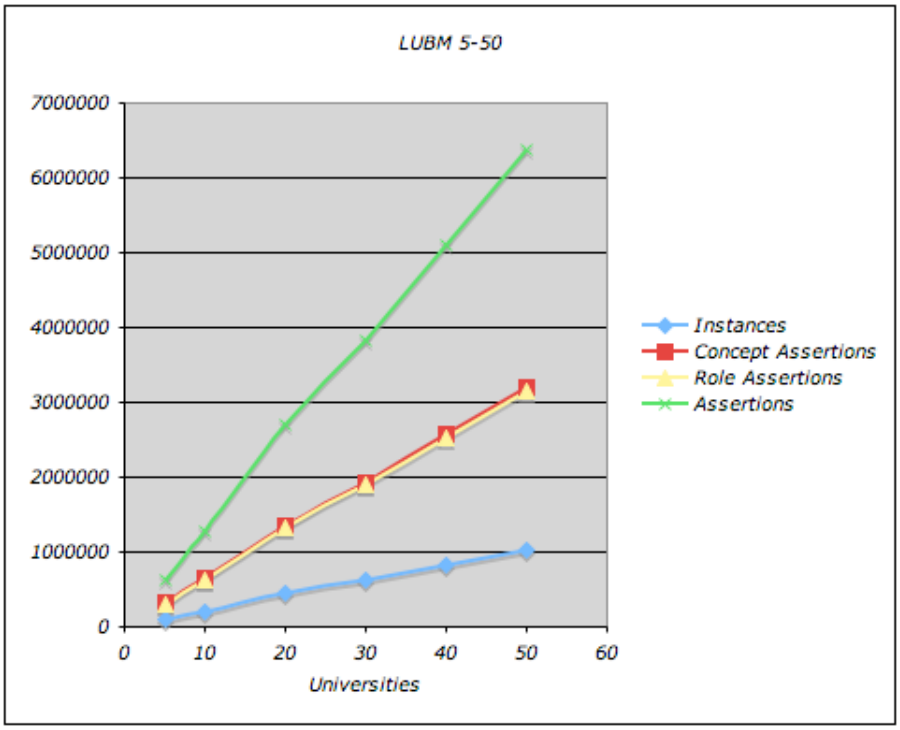


Figure 1: Linearly increasing number of individuals, concept assertions and role assertions for different numbers of universities.

collection time dominate all other runtimes to a large extent. Due to the large amount of data in LUBM runtimes being examined over and over again by the garbage collector, and querying times increase in a superlinear way. The declaration of data structures as persistent (in the sense of being non-garbage) is provided after the ABox consistency check. The expression `(declare-current-knowledge-bases-as-persistent)` is used for declaring knowledge bases as persistent. Racer Systems offers consulting services in order to support use to maximally benefit from these services in industrial applications.

We take LUBM as a representative for largely deterministic data descriptions that can be found in practical applications. The investigations reveal that description logic systems can be optimized to also be able to deal with large bulks of logical descriptions quite effectively. LUBM allows us to study the data description scalability problem.

In addition to LUBM in this section we also discuss the UOBM-Lite benchmark. It is also scalable and was tested with 1-5 universities, each with all departments. The characteristics of the KB and the benchmarks are shown in Figure 4. The logic of UOBM is $\mathcal{ALCF}$ after GCI absorption and the ABox adds datatype properties. The size of the benchmark for 5 universities results in 138K individuals, 509K individual assertions, and 563K role assertions.

Each benchmark was evaluated with 15 grounded conjunctive queries designed by the authors of UOBM. The benchmark has the same structure as for LUBM. The runtimes given in Figure 4 show that RacerPro's ABox consistency performance scales well for up to 10 universities. In contrast to LUBM the UOBM benchmark does not allow the unique

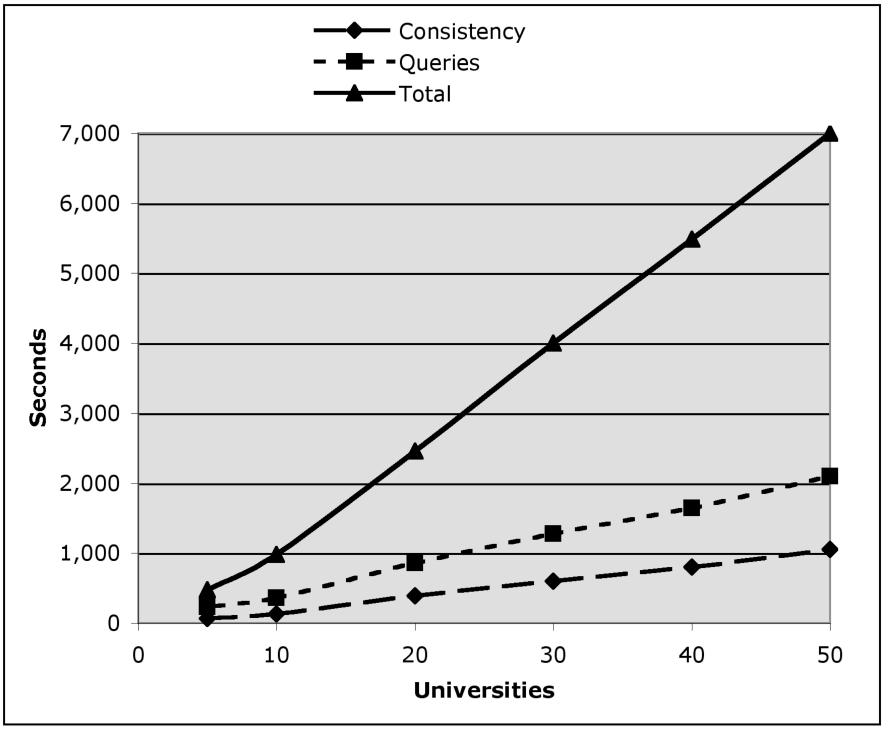


Figure 2: Runtimes for deterministic version of LUBM

name assumption. The query execution time also scales well for up to 3 universities. However, for 4 universities it increased by a factor 4 and timed out for 5 universities after 30 000 seconds. The graph in the lower part of Figure 4 displays the curves for the ABox consistency test (dashed line), query execution (dotted line), and the total benchmark time (solid line). The non-linear trend can be easily noticed. It is interesting to remark that 99.86% of the query runtime is spent for 3 of the 15 queries. This performance asks for a refinement of existing or design of new optimization techniques. This is a topic for future work.

For answering Abox queries we have found RacerPro to be as fast as systems that are specifically taylored to answering Abox queries w.r.t. very specific Tboxes such as LUBM (with low expressivity). However, RacerPro can also handle Abox queries with respect to Tboxes which these systems cannot handle.

# 3 RacerPorter

RacerPorter is a text-based ontology editor and the default GUI client of the RacerPro description logic system (DLS). The metaphorical name RacerPorter was chosen to stress that a "user friendly entrance" shall be provided to an otherwise "faceless" DL-server, like a hotel porter. Although quite a number of ontology browsing and inspection tools (called OBIT in the following) as well as authoring tools exist and numerous papers have been written about them [KPS$^+$05, LN05, LN06, KMR04], RacerPorter represents

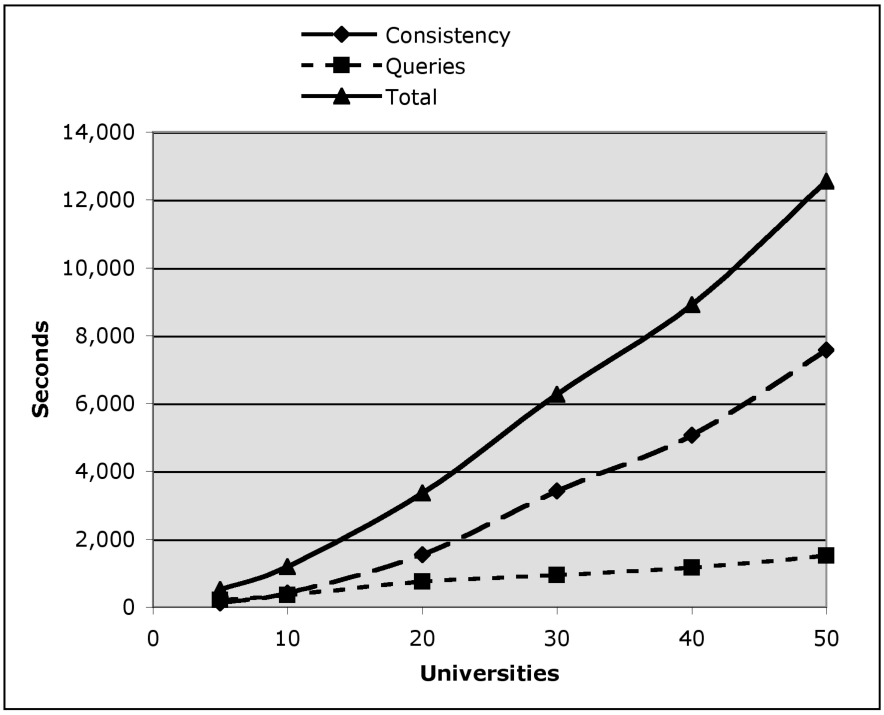


Figure 3: Runtimes for non-deterministic version of LUBM

a different approach. We present the design principles behind RacerPorter as well as the tool.

As already mentioned before (e.g., in [LBF+06]), ontology editors are currently the main software tool for ontology design tasks. They provide for functionality such as browsing and editing single ontology elements and the whole ontology structure, performing communication with background reasoners, visualization of reasoners' feedback and so on.

When developing RacerPorter, the aim was not only to support this basic functionality but also to enhance usability and to solve certain "scalability problems". Users "unscrupulously" load rather large OWL files into the reasoner and expect their taxonomies to be visualized with the ontology design tools such as RacerPorter. We reacted to the complaints of RacerPorter users by enhancing the performance and usability of previous versions of RacerPorter on large KBs.

RacerPorter exclusively uses the KRSS port of RacerPro, although support for OWL is included as well. Compared with DIG, KRSS has the advantage that it can also be used as a *shell language* (DIG was designed under a different perspective). The XML messages standardized by DIG are not on the correct level of abstraction for a shell language (even if a non-XML serialization of DIG messages were used).

In a nutshell, RacerPorter has been designed to meet the following design characteristics:

1. RacerPorter offers a KRSS shell for interactive communication with RacerPro. Al-

| TBox Logic | CN | R | Axioms | ABox Logic |
|---|---|---|---|---|
| $\mathcal{ALCF}$ | 51 | 49 | 101 | $\mathcal{ALCF}(\mathcal{D}^-)$ |

(CN = no. of concept names, R = no. of roles)

| U | Inds | Ind. Ass. | Role Ass. | L | P | Cons | I | Q | T |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 43 642 | 116 092 | 129 695 | 35 | 16 | 100 | 15 | 446 | 608 |
| 5 | 138 452 | 509 902 | 563 699 | 160 | 197 | 7 670 | 40 | 30 000 | 30 000 |

U = no. of universities, L = load time, P = KB preparation time,
Cons = time for initial ABox consistency test, I = query index generation time,
Q = nRQL query execution time, T = total benchmark time.

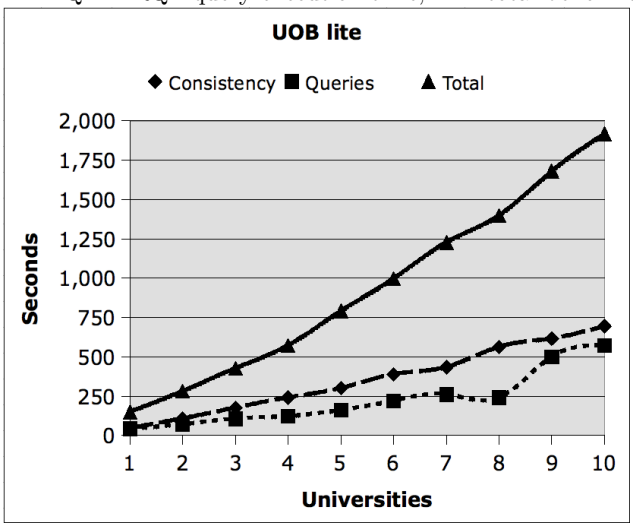


Figure 4: UOBM-Lite benchmark characteristics and runtimes per query set (15 queries, time in secs, timeout after 30 000 secs).

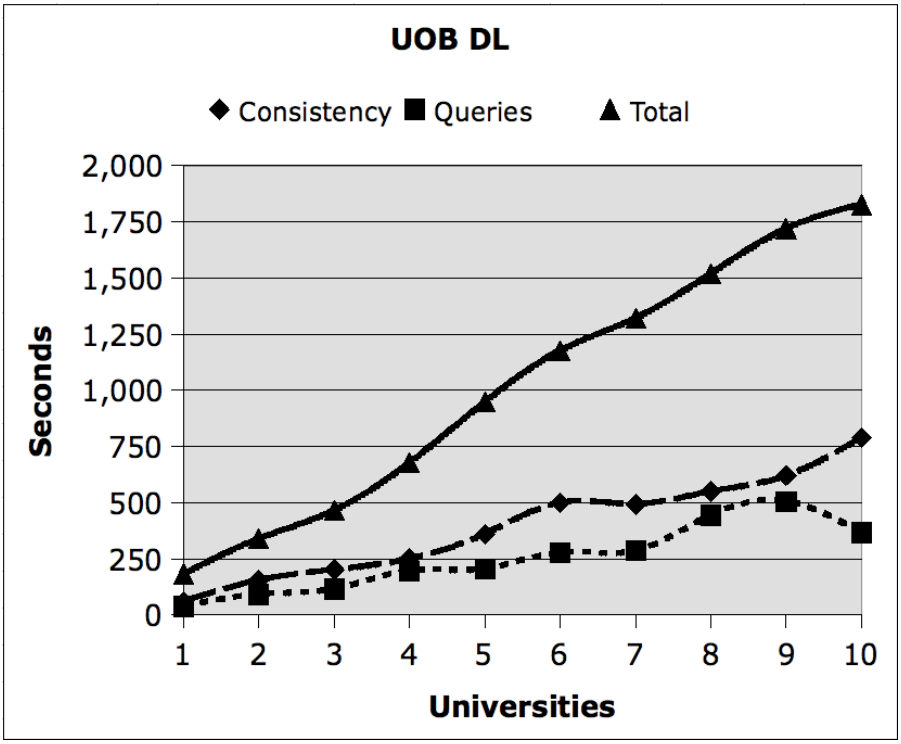


Figure 5: Racer runtimes for UOBM DL and a query set of 15 queries.

ready RICE (visit `http://www.ronaldcornet.nl/rice/`) offered a shell.

2. Unlike tools such as OntoTrack [LN05, LN06], Swoop [KPS+05] and GrOWL [SK06], RacerPorter is not designed to be a graphical OWL authoring tool. However, we believe that for expert users also text-based editors are useful and these text editing facilities have to be tightly integrated with graphical visualization tools. In RacerPorter, we added a text editor with Emacs-styled buffer evaluation mechanisms which in combination with a shell allows for rapid and interactive authoring of KRSS KBs.

3. Obviously, ontology visualization is important as well. Ontologies have different aspects, i.e., intensional and extensional ones. One can expect that an OBIT is able to visualize taxonomies, role hierarchies as well as ABoxes as graphs and/or trees (of certain kind, using certain graph layout algorithms). An OBIT should thus provides appropriate visualization facilities.

4. Given the fact that OWL KBs tend to become bigger and bigger, appropriate navigation, browsing and focusing mechanisms must be provided, since otherwise the user gets "lost in ontology space". An OBIT must thus provides appropriate (syntactic and semantic) mechanisms.

5. OBITs (as well as the corresponding DLSs, of course) should be able to process very large ontologies (scalability aspect).

6. Given that both shell-, gadget- as well as graph-based interactions are offered, the question arises: How to link these interactions, and the results produced by them? An OBIT must provide appropriate solutions.

7. An OBIT should be designed to work in a non-blocking way (asynchronously). This is especially valuable if large ontologies are processed, and processing takes some time.

8. It is desirable if an OBIT can maintain different connections simultaneously to different DLS servers. While one server is busy, the user can change the active connection and continue work with another server.

9. An OBIT should avoid opaqueness. Especially if modes are used (and the interface is stateful), then it is necessary to appropriately visualize these modi.

10. Functionality for starting, stopping and controlling DL servers is desirable. Since each DLS has its proprietary functions and peculiarities, it becomes clear that at least part of the OBIT functionality must be tailored for the target DLS.

### 3.1 Towards user-friendly and scalable OBITs

A KRSS or OWL ontology represented in a description logic system has many different aspects: the taxonomy represents the subsumption relationships between concept names or OWL classes, the role hierarchy represents the subsumption relationships between roles or OWL properties, and the ABox represents information about the individuals and their interrelationships (the extensional knowledge). Additional aspects may be present, e.g. queries and rules. Thus, we can make a "shopping list" of "things" which must be accessed, managed and visualized with a DLS OBIT: different DL servers and their connection profiles[1], TBoxes, ABoxes, concepts, roles, individuals, queries and rules, ABox assertions, etc.

In order to avoid an overloaded GUI – which would try to represent these different aspects and aspect-specific functionality in a single window pane – in a similar way as other graphical ontology tools, we favor tabbed interfaces in order to achieve a clean separation of different aspects. Different tabs thus present different aspects of the ontology together with aspect-specific commands. The term "different perspectives" also describes the approach quite well.

Whereas many operations are directly performed on the displayed representations of the objects on the RacerPro servers by means of mouse gestures (direct manipulation), we also favor push buttons to invoke commands. In many cases, push buttons will directly invoke KRSS commands, e.g., send an `abox-consistent?` to the connected DL server. Push buttons also have the neat effect to inform the user directly about commands which are reasonable to apply or which can be applied at all in a given situation, simply by being visible, so there is no need to search for a command in a pull-down menu, which distracts focus. However, it should also be noted that the buttons occupy screen space, and using buttons and menus should be kept in balance.

[1]The connection and server settings can be managed using the so-called connection profiles which are familiar from networking tools such as SFTP browsers.

In many cases, some input arguments must be provided to KRSS commands. Input arguments are provided directly by the user if direct manipulation is employed for the interaction, but with simple push buttons this is not directly possible. Either the user must be prompted for arguments, or a notion of "current objects" must be employed. These current objects may have been (implicitly) selected by the user before and are from then on automatically supplied as input arguments to KRSS functions. This results in a stateful GUI. Sometimes, stateful GUIs are considered harmful. However, we will see that states are unavoidable if non-trivial ontology-inspection tasks shall be performed. Additionally, since also a DLS has a state, this state should be adequately reflected by the GUI as well (which automatically makes it stateful). In order to avoid opaqueness it is very important that the current state is appropriately visualized, e.g., in a status display which is visible at any time. According to the shopping list mentioned before, we must thus have a notion of a current DLS server, a current TBox and ABox, current concept, individual and role, current query, etc. These current objects partially constitute the current state of the OBIT.

The different tabs of the OBIT visualize often different objects. For example, one tab shows the individuals in the current ABox (the individuals tab), and another tab shows the concepts in the current TBox (the concepts tab). The information displayed in a certain tab thus depends on the current state. Additionally, the current concept (or the current individual) will be highlighted in the concepts tab (resp. the individuals tab), so it can be recognized easily. Two different tabs can also present the same objects, but use different visualizations. For example, the taxonomy tab also presents the concepts in the current TBox, but displays them as nodes in a graph whose edges represent subsumption relationships. Since all tabs display information according to the current state, the shown information is interrelated.

For certain ontology inspection tasks, it is further necessary to relate the information displayed on different tabs. One must establish a kind of information flow between different tabs. Let us illustrate this need for an input/output flow of information with an example.

As described, the individuals tab presents the list of individuals in the current ABox. If an individual is selected from that list, it automatically becomes the current individual. In order to explore of which concepts this individual is a direct instance, it is possible to push the "Direct Types" button which sends the `direct-types` KRSS command to the DLS, using the current ABox and current individual as arguments. In many cases, further operations shall be applied to the result concepts just returned, e.g., in order to explore which other instances of these concepts are presented in the KB. Thus, the concepts tab should provide a functionality which allows to refer to and highlight the just returned concepts, so that subsequent operations can be easily applied on them.

In order to establish this kind of information flow, we augment the notion of the current state by also including sets of selected objects in the state. Thus, the concepts returned by the `direct-types` command can become selected concepts. Selected concepts are shown as highlighted, selected items in the tabs which present concepts. Moreover, there are also selected individuals and selected roles. Objects can either be selected manually by means of mouse gestures, or automatically by means of KRSS commands, no matter how they are invoked. All what matters is the notion of selected objects. The set of selected objects is also called the clipboard. The current objects are seen as spe-

cific selected objects. Furthermore, all this state information is session-specific, given the fact that the OBIT should be able to maintain several connections and thus associated sessions simultaneously.

As said earlier, a shell tab is provided for interactive textual communication with the DLS. We claim that only shell-based interactions can offer the required flexibility and expressivity needed for advanced ontology inspection tasks. The shell must be incorporated into the above mentioned information flow as well. For example, if the `direct-types` command is entered into the shell, then it must be possible to refer to the current ABox as well as to the current individual which has been selected with the mouse in the individuals tab before (without having to type its name or having to use "copy & paste"). Furthermore, after the command was executed, it must be possible to select the returned concepts which are now shown in the shell as command output.

Focus control and navigation are two other important issues. It is well-known that the notion of current and selected objects can be used to control the focus. For example, the current concept can provide the root node in the taxonomy graph display. Only the descendants of the current concept will be shown. To browse larger taxonomies, a "depth limit" cutoff on the paths to display can be specified, and an interactive "drill down"-like browsing style can be realized. The node gesture "select" (e.g., a left mouse click) automatically changes the current concept and thus the graph root. If the graph is redrawn immediately, this allows to drill down a large taxonomy interactively and dynamically. However, this automatic graph recomputation changes the focus.

In principle, changing the focus automatically can be very distracting. In Web browsers, the navigation buttons (back and forth) are thus of utmost importance; they allow to reestablish the previous focus effortlessly. Thus, a focus or history navigator should be present in an OBIT, as also found in Swoop or GrOWL [KPS$^+$05, SK06]. However, many users are unhappy with hyperlink-like focus-destroying operations. In Web browser, tabbed browsing has been invented to address this problem. Thus, we think that the user should be able to determine when and how the focus is changed once a gadget is selected.

Sometimes, it is also desirable to focus on more than one object, e.g., for ABox graphs. We can simply use the selected objects for that as well. In case of the ABox graph, each selected individual can specify a graph root, and unraveling (as understood in Modal Logics) is used to establish a local perspective from that individual's point of view (so there is one graph per selected individual). This resolves many visual cluttering problems. The clipboard is thus not only a structure that enables flow of information, but can also be used to control the focus. This also implies that the focus control is now highly flexible: Since the clipboard can be filled from results of KRSS commands, even a semantic focus control is possible. For example, an ad-hoc nRQL query can be typed into the shell, and, with the push of a button, one can focus on the returned ABox individuals in the ABox graph tab. In our terminology, this kind of focusing by invoking inference services (such as, e.g., `direct-types`) is called semantic focusing. However, one also often wants to focus on individuals simply by their names. Thus, a kind of search field is needed. Objects that contain the search string get selected automatically. This enables a so-called syntactic focusing. We have found that many available tools don't offer adequate mechanisms to achieve this kind of information flow and focus control.

Summing up, we conclude that the current state must be a vector `<current objects,`

`selected objects,active tab,display options>`. Each time a state changing operation is performed by the user (e.g., the current objects or the clipboard is changed), a so-called history entry is automatically created, which is just a copy of the current state vector. History entries are inserted at the end of a queue, the so-called navigation history. A history navigator offers the standard navigation buttons. The OBIT always reflects the current state, no matter whether this is the latest one or a historic state from the history. A history entry only preserves the state information of the GUI, but not the content of the DLS at that time. Thus, a well-known problem arises here: If a historic state is reactivated, then it may no longer be possible to actually refer to the objects referenced by that state, since they may have been already deleted from the DLS. This problem is well-known to WWW users which keep a browsing history. There is no practical solution to this problem (one cannot preserve "copies" of DLS server states in history entries).

We believe that OBITs should allow for asynchronous usage. While a time-consuming command is processed by the DLS, the GUI shouldn't block; instead, the result should be delivered and displayed asynchronously once available. Although the busy DLS will not accept further commands until the current request had been fulfilled (nowadays, there are no true multi-user DLS), in the meantime the OBIT should a) display status information in order to inform the user what the DLS is currently doing (future versions of RacerPro and RacerPorter will also support progress bars), and b), if possible, allow the user to do other things, e.g., continue editing a KRSS KB, or connect to and work with a different DLS.

### 3.2 RacerPorter – How to use

RacerPorter was designed according to the design principles just explained. Each tab has a uniform organization, which, we believe, makes the GUI consistent and comprehensible. With the exceptions of the log tab and the about tab, each tab has **six areas.** Figure 6 shows the taxonomy tab. Let us describe the six areas of this tab.

The **first area** shows the available tabs: Profiles, Shell, TBoxes, ABoxes, Concepts, Roles, Individuals, Assertions, Taxonomy, Role Hierarchy, ABox Graph, Queries, Rules, Log, and About tab. The **second area** is the status display. It displays the current objects, the current namespace, the current profile (representing the current server connection), as well as the current communication status. The clipboard content is not shown, only the cardinality of the sets of selected objects (in the small number fields). The selected objects are highlighted once an appropriate tab is selected. The **third area** shows the history navigator. The **fourth area** is the tab-specific main area. Tab-specific display options and commands are then presented in the **fifth area.** Finally, there is the info display which is the **sixth area**. The info area is similar to the shell; however, it only "echos" the shell interaction (accepts no input). All user-invoked KRSS commands are put into the shell which are thus also echoed in the info display. This helps to avoid opaqueness, and as a side effect, the user learns the correct KRSS syntax.

The taxonomy and the ABox graph tabs use graph panes for the fourth area. With the exception of the shell, log and logo tab, the other tabs use list panes. List panes allow single or multiple selections of items; selected items represent the selected objects (clipboard). The last selected item specifies the current object. A search field is always present and allows to select objects by name. Selected items will appear on the top of

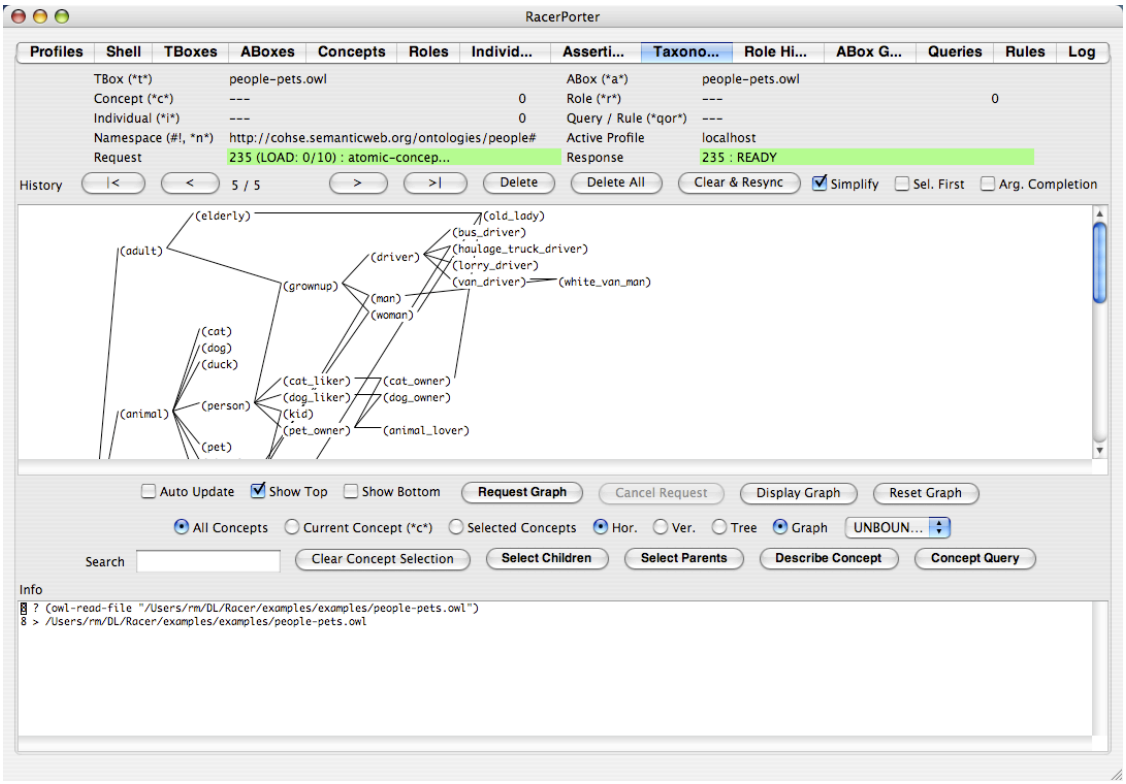


Figure 6: RacerPorter – The Taxonomy Tab

the list if the "Selected First" checkbox is enabled. Some list panes display additional information on their items in multiple columns; e.g., in case of the TBox pane, not only the TBox name is shown, but also the number of concepts in that TBox, profile and DLS server information is shown in the profiles list, etc.

The graph panes are more complicated to handle since they allow to specify focus, layout as well as update options. In case of the ABox graph pane, one can determine which individuals and which edges are displayed. Thus, for both individuals and roles, the focus can be set to the current objects, to the selected objects, or to all objects. Appropriate radio buttons are provided. Additional radio buttons control whether only told role assertions, or also inferred role assertions shall be shown. Additional buttons allow to specify whether the graph display shall be updated automatically if the focus or layout options changes, or whether the user determines when an update is performed. In the latter case, the user first uses the button "Request Graph" to acquire the information from RacerPro (phase 1). Once the graph is available, the "Display Graph" button becomes enabled; if pushed, the graph layout is computed and displayed. Both phases can be canceled (and different focus and layout options selected subsequently) if they should take too long[2].

Finally, let us briefly discuss some features of the shell. The shell provides automatic

[2]Although RacerPorter does not block in phase 1, unfortunately we have to block the GUI in phase 2 due to a restriction of the GUI framework we are currently using.

command completion (simply press the tab key) as well as argument completion. The potential commands / arguments are presented in a pop-up list. Command completion is achieved by accumulating all results ever returned by KRSS commands. In order to make it easy to reexecute commands, the shell maintains its own shell history (not to be mistaken with the navigation history). Since the shell is tailored for KRSS commands in Lisp-syntax, we provide parenthesis matching, convenient multi-line input as well as pretty printing. Moreover, one no longer has to use full qualified names for OWL resources. To select the objects returned by a shell command, one has to hit the appropriate button (e.g., the "Selected Individuals := Last Result" button).

The log tab keeps a communication log which can be inspected at any time in order to learn what the DLS is currently doing. The current communication with RacerPro is also visualized in the request and response status fields; appropriate colors are used to visualize the different stages of such a communication (first the request is send, then RacerPro is busy, then the result is received over the socket, finally the result is parsed, etc.; note that errors can occur at any time in such a processing chain).

RacerPorter includes an Emacs-compatible editor with buffer evaluation mechanism (see Figure 7). Furthermore, RacerPorter will provide for visualization of explanations if an inconsistency occurs. This is done in two ways: a) by highlighting of culprits for inconsistency in the axioms and assertions tabs, and b) by highlighting of culprits in the editor window.

In the queries tab, nRQL queries can be executed. Besides of this, next versions of RacerPorter will also allow for sending SPARQL queries and displaying result tuples in the special SPARQL tab.

RacerPorter also includes basic functionality to start and stop RacerPro servers; startup and connection options can be specified with a profile editor. Finally, we want to stress that RacerPorter is a multi-session tool; thus, the current state and history, but also the shell content, is session or profile specific. Thus, the content of the shell must be saved and reinstalled once the original profile or session is reactivated. As one expects, this can be very memory-intensive, but thats the only way to do it.

### 3.3 Some notes about performance

We learned that a lot of effort must be put into an OBIT until it can be successfully used on large KBs. We tested the redesigned version of RacerPorter on the OpenCyc ontology [Ope] consisting of 25568 concepts, 9728 roles and 62469 individuals. The result is that RacerPorter can be used to browse and visualize this ontology. Moreover, time and memory requirements are not too bad. To achieve this, many aspects of the original code had to be reworked thoroughly. This not only concerns the choice of appropriate container data structures "that scale", but also issues like communication over socket streams. For example, in our case it was no longer possible to simply coerce sequences of characters read from the socket connected to RacerPro to strings (although this is a very fast operation), since these strings simply get too big to be represented in the environment we use. For the implementation of the clipboard, we originally used lists as data structures. However, if the clipboard contains some ten-thousand instances, one can easily imagine that performance breaks down, since checking whether an object is selected (and thus a member of the clipboard) or not is an operation which has to be performed

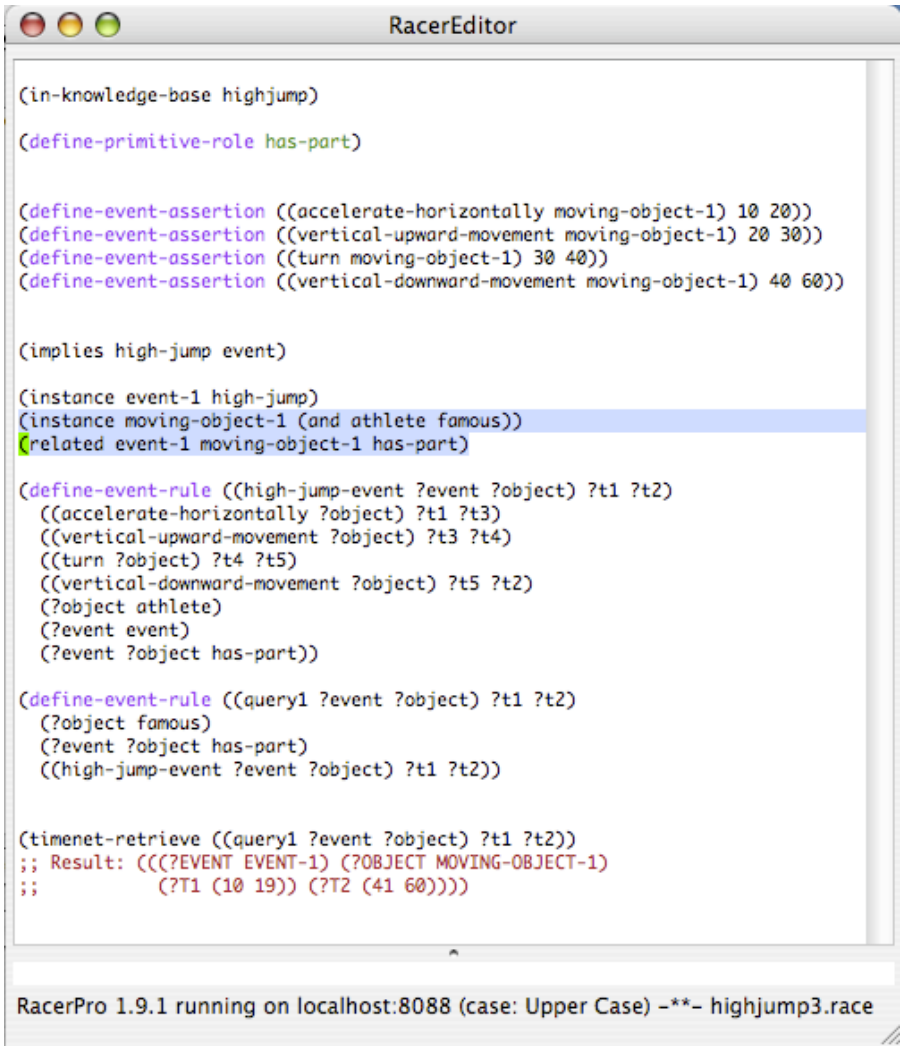


Figure 7: RacerEditor

very frequently. Furthermore, in order to reduce socket communication latency, caches must be used in order to achieve an acceptable performance. Even the design of theses caches is demanding.

### 3.4 Summary

Summing up, we have presented design principles for OBITs and showed how they are realized in RacerPorter. Given the abundance of visualization facilities required, an OBIT has to link interactions and results into a coherent information flow. In RacerPorter, this information flow is established not only between the tabs and the system core (like a plugin architecture as it is realized, e.g., in Protégé) but also between certain tabs. To be user-friendly, an OBIT must come up with easy browsing and navigation solutions even for large ontologies. Although the existing tools are already very impressive, there

is certainly room for enhancements, especially regarding visualization of and navigation in large ontologies in combination with powerful text-based editing techniques.

If you have specific requirements for visualizing your Aboxes or if you have specific application code that should be integrated into the user interface for the reasoning component, ask Racer Systems for consulting.

# 4 Enhancements to nRQL: Server-side programming

nRQL has been revised extensively. Some important features of the new nRQL versions are:

- nRQL offers now termination-safe functional lambda expressions. These lambda expressions can be used in query heads and rule antecedences. nRQL allows a restricted kind of server-sided 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).
- Adequate handling of *synonym individuals* and proper treatment of `same-as` assertions.
- A more complete `told-value-if-exists` head projection operator to retrieve told values of concrete domain attributes
- Enhanced data substrates.
- Enhanced TBox queries.
- The RCC substrate now works for OWL.
- New head projection operators (e.g., `types`, `direct-types`, `describe`, `individual-synonyms`)
- Various bug fixes and speed enhancements.

Here are some nRQL 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. nRQL 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 lambda expressions 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. 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.

Please refer to the chapter on nRQL in the RacerPro User Guide in order to learn how to implement *even more efficient* versions of these queries.

Functional programming statements cannot only be used in Lambda terms in queries, but also at toplevel.

```
(evaluate (let ((x ...)) ...))
```

Often, user-defined query format output must be generated from query results, e.g., HTML reports. This is easy with nRQL 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 `example.html` which is shown in Figure 8:

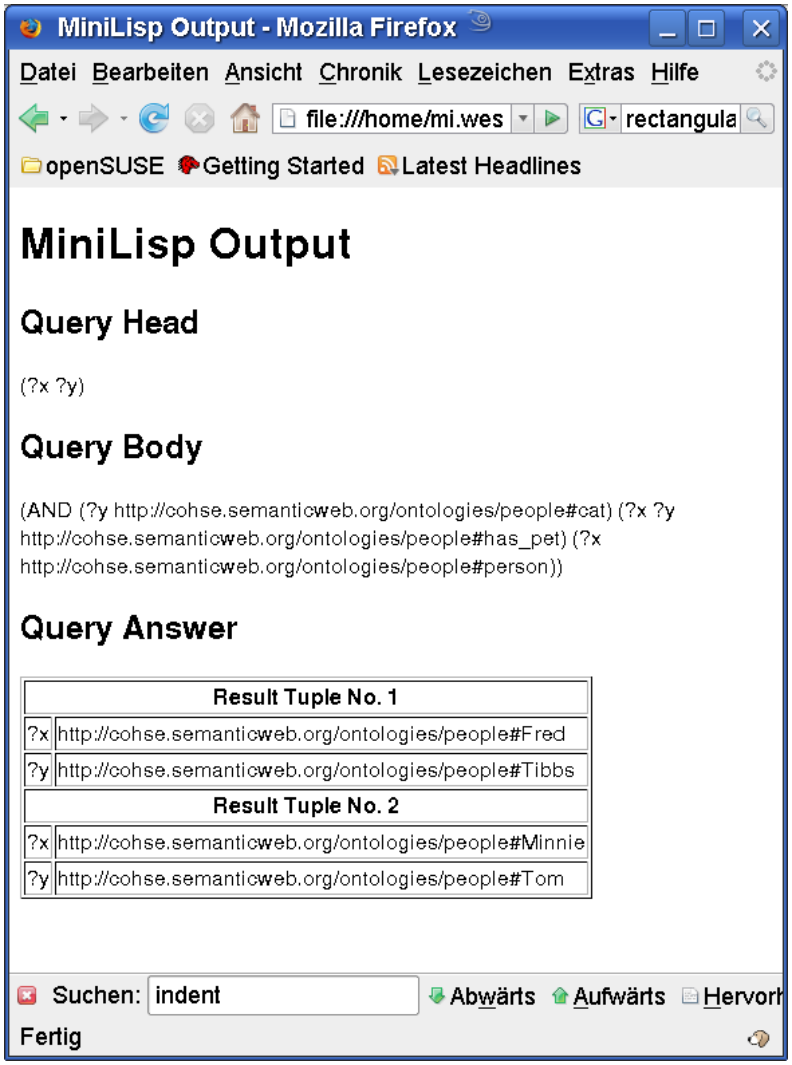


Figure 8: Generated HTML Page

```
(evaluate
      (let ((res
              (retrieve '(?x ?y)
                        '(and (?x \# !"person")
                              (?x ?y \# !"has_pet")))))
        (with-html ("example.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)))))))
```

## 5 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 manipu-

lated 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.

```
(evaluate
     (let ((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 9, 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 10. Afterwards, e.g., the corresponding inferred relational (role filler) ABox structure can be viewed using one of the graph panes of RacerPorter (see Figure 11).

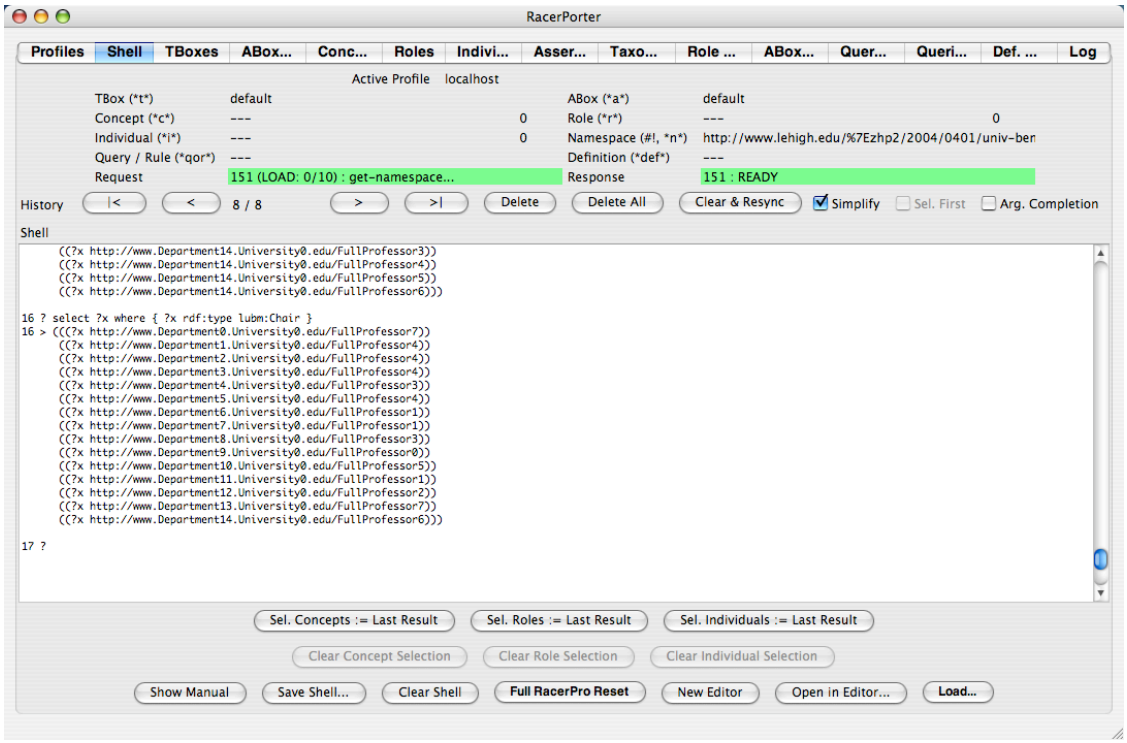


Figure 9: SPARQL Query

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.

```
(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))))
```

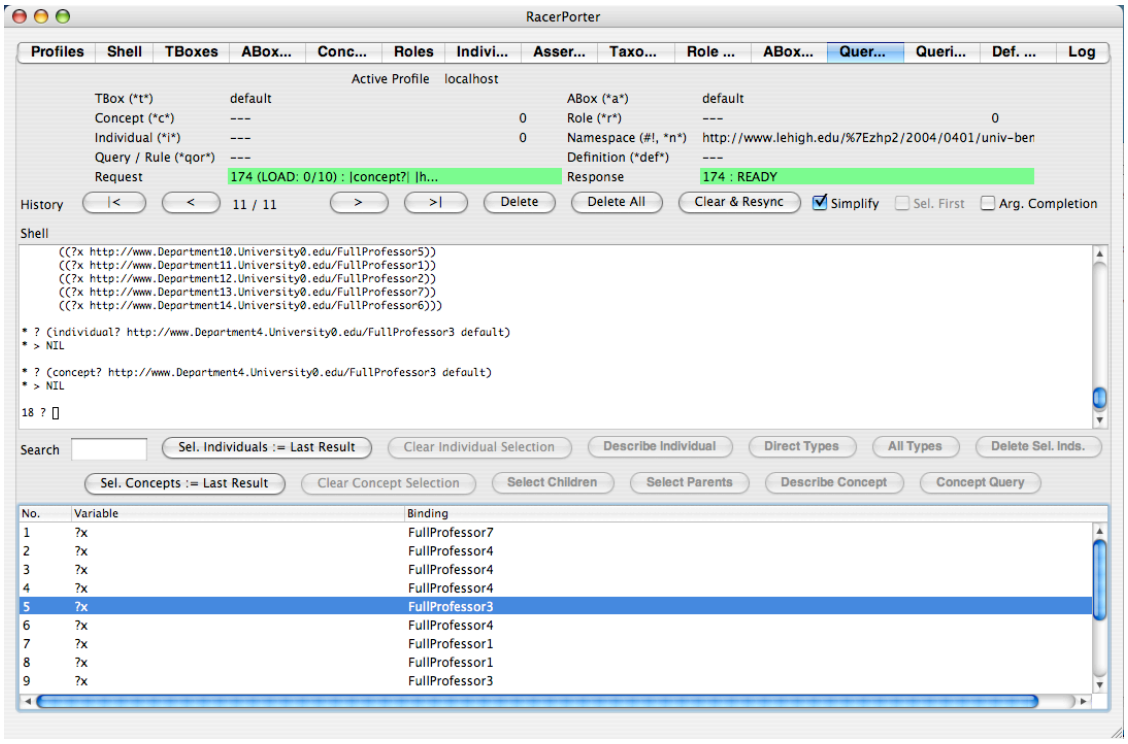


Figure 10: Query results

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

(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.

```
(evaluate
     (let ((db (open-triple-store "test")))
```

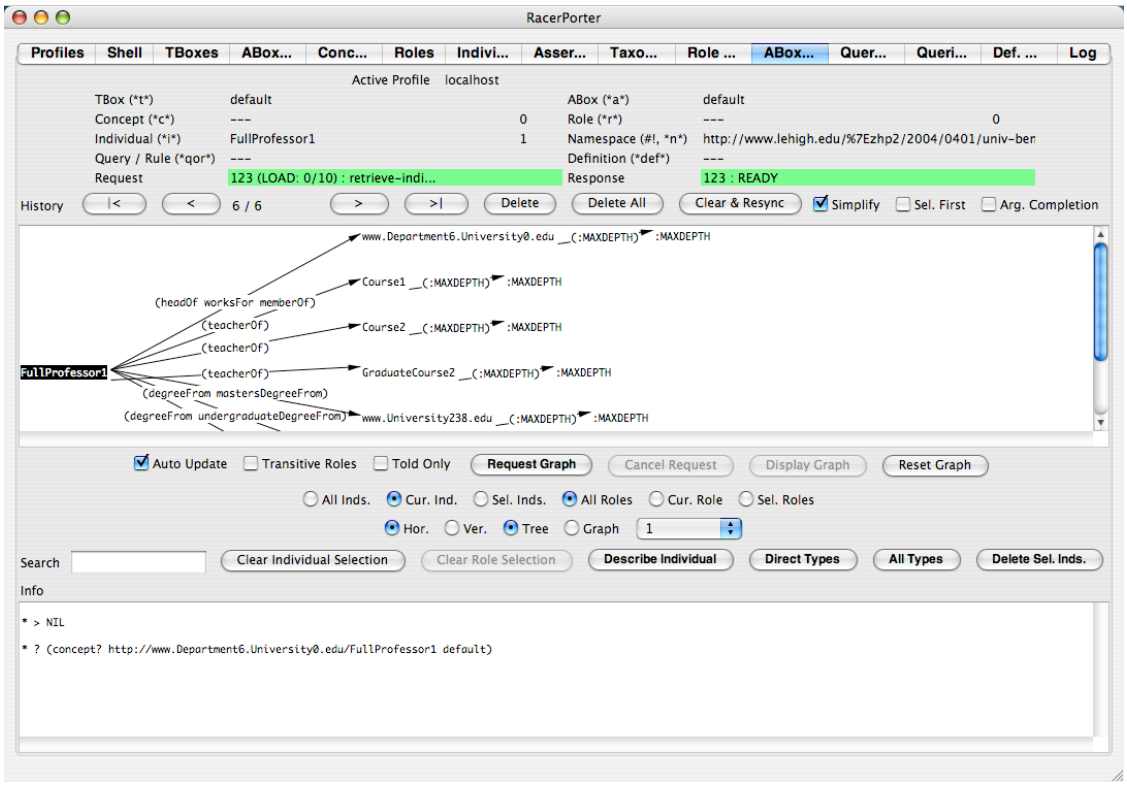


Figure 11: ABox Graph

```
    (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.

```
(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.

# 6 Rule formalism extended to support abduction

In RacerPro, Abox rules can be specified that are applied w.r.t. the autoepistemic semantics. In a nutshell this means the following: if for the variables in the rule body (precondition), some bindings withindividuals of the Abox can be found such that all instantiated query atoms of the body are entailed, then the assertion of the rule head (conclusion) is added to the Abox with variables instantiated appropriately (forward-chaining). All rules whose preconditions match the assertions of the Abox in the way just described are applied until nothing new can be added. The rule mechanism allows for a convenient augmentation of Aboxes with assertions. For details of the formalism see the nRQL User's Guide. An example for such a rule is the following statement.

```
(define-rule (?x ?y has-sibling)
  (and (?z ?x has-child)
       (?z ?y has-child)
       (?x human)))
```

The form `(?x ?y has-sibling)` is the head and the rest is the body.

Rules may also be specified as part of an OWL document in SWRL syntax (see Figure 12 for a graphical specification of SWRL rules in Protege).

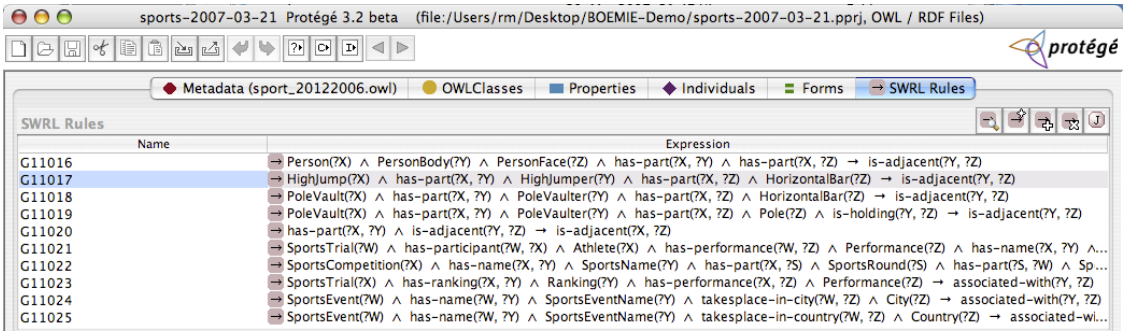


Figure 12: Rules in Protege.

Rules are required because for some purposes, OWL is not expressive enough. Rather than just adding statements to the Abox whenever preconditions are satisfied as suggested above, in some cases it makes sense to just ask whether the head is entailed by an Abox (rules are applied in a backward-chaining way). This facility can also be used for computing what should be added to an Abox to make an assertion be entailed (abduction). Depending on the rule, this could involve the addition of new individuals to the Abox. In the following we present an example involving the detection of a pole-vault event.

```
(full-reset)

(in-knowledge-base test)

(define-primitive-role near)
(define-primitive-role has-part)

(disjoint pole bar)
(disjoint pv-in-start-phase pv-in-turn-phase)

(define-rule (?x ?y near)
  (and (?z ?x has-part)
       (?z ?y has-part)
       (?x pole)
       (?y human)
       (?z pole-vault)
       (?z pv-in-start-phase)))

(define-rule (?x ?y near)
  (and (?z ?x has-part)
       (?z ?y has-part)
       (?x bar)
       (?y human)
       (?z pole-vault)
       (?z pv-in-turn-phase)))

(instance p1 pole)
(instance h1 human)
```

Given the statements in this knowledge base one can ask the following query.

```
(retrieve-with-explanation () (p1 h1 near) :final-consistency-checking-p t)
```

The answer is

```
(t (((:tuple)
      (:new-inds IND-2)
      (:hypothesized-assertions
         (related IND-2 p1 has-part)
         (instance IND-2 pole-vault)
         (instance IND-2 pv-in-start-phase)
         (related IND-2 h1 has-part)))))
```

The answer is `t` (as expected), and the result returned by RacerPro suggests that there is one way to achieve this answer. One new individual is required (`IND-2`). In addition, a set of hypothesized assertions to achieve the positive answer is indicated. Multiple ways to (positively) answer the query are possibly generated in other cases. If the optional

keyword argument `:final-consistency-checking-p t` is used, RacerPro checks if the Abox would remain consistent if the hypothesized assertion were added to the Abox. The additional option `:incremental-consistency-checking-p t` might be more efficient in some cases (intermediate checks) but there is some overhead with multiple consistency checks.

Now, assume that there are some additional assertions added as in the following example.

```
(full-reset)

(in-knowledge-base test)

(define-primitive-role near)
(define-primitive-role has-part)

(disjoint pole bar)
(disjoint pv-in-start-phase pv-in-turn-phase)

(define-rule (?x ?y near)
  (and (?z ?x has-part)
       (?z ?y has-part)
       (?x pole)
       (?y human)
       (?z pole-vault)
       (?z pv-in-start-phase)))

(define-rule (?x ?y near)
  (and (?z ?x has-part)
       (?z ?y has-part)
       (?x bar)
       (?y human)
       (?z pole-vault)
       (?z pv-in-turn-phase)))

(instance p1 pole)
(instance h1 human)
(instance e1 pole-vault)
(instance e1 pv-in-start-phase)
(related e1 p1 has-part)
(related e1 h1 has-part)
```

Given the statements in this knowledge base on can ask the query from above again.

```
(retrieve-with-explanation () (p1 h1 near) :final-consistency-checking-p t)
```

The answer is now:

```
(t (((:tuple) (:new-inds) (:hypothesized-assertions))))
```

This means, `(p1 h1 near)` would be added if the rule was applied in a forward-chaining way, i.e., no hypothesized assertions are required. The `retrieve-with-explanation` facility can also be used for explaining to a user what must be added to an Abox in order to positively answer a query.

Racer Systems has applied the abduction operator to media interpretation problems (image interpretation and natural language text interpretation). We offer consulting to support industrial application projects that could use these facilities.

# 7 Event recognition facility

In some applications it might be interesting to state that assertions are valid only within a certain time interval. This is not possible with standard description logic systems. There are several research resuls available on qualitative temporal reasoning in the context of description logics. RacerPro can support qualitative temporal reasonsing as part of the nRQL system (compare the section on RCC substrates in the User's Guide). In addition, in some cases, quantitative information about time intervals might be available and could be relevant for query answering. RacerPro now also supports Abox query answering w.r.t. assertions that are associated with specifications for time intervals. The following example shows the main idea.

```
(in-knowledge-base traffic-analysis)

(define-primitive-role r :inverse r)

(implies car vehicle)
(implies volkswagen car)

(instance vw1 volkswagen)
(instance vw2 volkswagen)
(instance ralf pedestrian)
(related vw2 vw1 r)

(define-event-assertion ((move vw1) 7 80))
(define-event-assertion ((move vw2) 3 70))
(define-event-assertion ((move ralf) 3 70))
(define-event-assertion ((approach vw1 vw2) 10 30))
(define-event-assertion ((behind vw1 vw2) 10 30))
(define-event-assertion ((beside vw1 vw2) 30 40))
(define-event-assertion ((in-front-of vw1 vw2) 40 80))
(define-event-assertion ((recede vw1 vw2) 40 60))

(define-event-rule ((overtake ?obj1 ?obj2) ?t1 ?t2)
  ((?obj1 car) ?t0 ?tn)
  ((?obj1 ?obj2 r) ?t0 ?tn)
```

```
((move ?obj1) ?t0 ?t2)
((move ?obj2) ?t1 ?t2)
((approach ?obj1 ?obj2) ?t1 ?t3)
((behind ?obj1 ?obj2) ?t1 ?t3)
((beside ?obj1 ?obj2) ?t3 ?t4)
((in-front-of ?obj1 ?obj2) ?t4 ?t2)
((recede ?obj1 ?obj2) ?t4 ?t2))
```

The following query check whether there exists an overtake event hidden in the Abox and the event assertions.

```
(timenet-retrieve ((overtake ?obj1 ?obj2) ?t1 ?t2))
```

The query returns a set of binding for variables if an event can be detected (and nil otherwise). For time variables an interval for the lower-bound and upper-bound are returned. Thus, the `overtake` event start at time unit 10 at the earliest and 29 at the latest. It ends at time unit 29 at the earliest and 60 at the latest. In a future version of RacerPro, redundant binding specifications will be removed.

```
(((?obj1 vw1) (?obj2 vw2) (?t1 (10 29)) (?t2 (41 60)))
 ((?obj1 vw1) (?obj2 vw2) (?t1 (10 29)) (?t2 (41 60))))
```

Note that the example involves reasoning. The fact that `vw1` is a car is only implicitly stated. In addition, temporal constraint have to be checked for the time intervals.

In the previous queries, variables are used. However, one might very well use constants in event queries as shown in the following example.

```
(timenet-retrieve ((overtake ?obj1 vw2) ?t1 ?t2))
```

Now, only bindings for the variable `?obj1` are returned.

```
(((?obj1 vw1) (?t1 (10 29)) (?t2 (41 60)))
 ((?obj1 vw1) (?t1 (10 29)) (?t2 (41 60))))
```

In a future version, a forward-chaining application of event rules will be supported, so there is no need to repeatedly cycle through all known events using respective queries. Racer Systems offers consulting for industrial partners who need event recognition w.r.t. expressive background ontologies.

# 8 Wait for DIG 2.0 – or use DIG 1.2 now

In many practical application systems based on DLs, a powerful ABox query language is one of main requirements. Conjunctive queries and unions of conjunctive queries are topics of recent research in this context. Additionally, query languages for so-called grounded conjunctive queries and some other practically motivated constructs have been used in existing DL systems for a long time. In this section, an ABox query language as part of the DIG protocol is introduced as part of an extension to DIG 1.1 (called DIG 1.2). The query language will be part of the upcoming DIG 2.0 standard with slight modifications (mostly, tags will be renamed to match up with other parts of DIG 2.0). For now, conjunctive queries are offered as part of RacerPro DIG 1.2.

## 8.1 Identification response

The response to an `<identifier/>` request includes a new element `<retrieve>` within the `<ask>` tag. Additionally, the tag `lang` can be specified to identify the fragment of a query language supported by the reasoner. Possible values are e.g. `cq` (conjunctive queries), `ucq` (unions of conjunctive queries), `focq` (first order conjunctive queries), `ugcq` (unions of grounded conjunctive queries) and so on. An example for a reasoner identification:

```
<identifier
      xmlns="http://dl.kr.org/dig/lang/schema"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
      http://dig.cs.manchester.ac.uk/schema.xsd"
      name="Racer"
      version="1.9.5"
      message="Racer running on localhost">
   <supports>
     <language>
       [...]
     </language>
     <tell>
       [...]
     </tell>
     <ask>
       [...]
       <retrieve lang="ugcq"/>
     </ask>
   </supports>
 </identifier>
```

## 8.2 Ask language

A query consists of a *query head* and a *query body*. Moreover, variables and individuals can be used in queries.

An `asks` part now can contains the new ask statement `retrieve` which has an attribute `id` as unique identification of the query and for which a query head (tag `head`) and a query body (tag `body`) must be defined. Within the `head` tag Abox individuals and variables (denoted as `indvar`) can be used. Variables are bound to those individuals which satisfy the query. For boolean queries, the head must be empty. The query body is built from query atoms (see below) using boolean constructs `qand`, `qunion`, and `qneg`, respectively. Query atoms can be concept query atoms or role query atoms, denoted with `cqatom` and `rqatom`, respectively. Concept query atoms consist of variables (or individuals) and complex concept expressions. Role query atoms consists of at least two identifiers for variables (or indidivuals) followed by a role expression.

The following conjunctive query with `id=q1` asks for all individuals of the concept `woman` which have `female` children. The requested knowledge base is identified by means

of a URI:

```
<asks
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
    uri="urn:uuid:...">
  <retrieve id="q1" lang="ugcq">
    <head>
      <indvar name="x"/>
    </head>
    <body>
      <qand>
        <cqatom>
          <indvar name="x"/>
          <and>
            <catom name="woman"/>
            <some>
              <ratom name="hasChild"/>
              <catom name="female"/>
             </some>
          </and>
        </cqatom>
      </qand>
    </body>
  </retrieve>
</asks>
```

The following query `q2` consists of conjunction of a concept query atom and a role query atom. It returns all mother-child pairs (bound to variables x and y):

```
<asks
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
    uri="urn:uuid:...">
  <retrieve id="q2">
    <head>
      <indvar name="x"/>
      <indvar name="y"/>
    </head>
    <body>
      <qand>
        <cqatom>
          <indvar name="x"/>
```

```
          <catom name="woman"/>
        </cqatom>
        <rqatom>
          <indvar name="x"/>
          <indvar name="y"/>
          <ratom name="hasChild"/>
        </rqatom>
      </qand>
    </body>
  </retrieve>
</asks>
```

The following boolean query `q3` asks if there are any individuals of the concept `woman` in the current ABox:

```
<asks
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
    uri="urn:uuid:...">
  <retrieve id="q3">
    <head>
    </head>
    <body>
      <qand>
        <cqatom>
          <indvar name="x"/>
          <catom name="woman"/>
        </cqatom>
      </qand>
    </body>
  </retrieve>
</asks>
```

Instead of variables, also ABox individuals can be used in the query, as illustrated by example of the following (boolean) query `q4`:

```
<asks
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
    uri="urn:uuid:...">
  <retrieve id="q4">
    <head>
    </head>
```

```
    <body>
      <qand>
        <cqatom>
          <individual name="eve"/>
          <catom name="woman"/>
        </cqatom>
      </qand>
    </body>
  </retrieve>
</asks>
```

As mentioned above, within the `retrieve` statement we can build unions of conjunctive queries using the operator `qunion` in front of the conjunctive queries:

```
<qunion>
  <qand>
    [...]
  </qand>
  [...]
  <qand>
    [...]
  </qand>
</qunion>
```

The `qneg` operator can be used in front of query body atoms, conjunctive queries and unions of conjunctive queries. For the semantics, see below.

```
<qneg>
  <qunion>
    <qneg>
      <qand>
        [...]
        <qneg>
          [...]
        </qneg>
      </qand>
    </qneg>
  </qunion>
</qneg>
```

A `qsameas` query atom can be used to enforce a binding of a variable (e.g., `<indvar name="betty"/>`) to an individual (e.g., `<individual name="betty"/>`), or to enforce that two non-injective variables are bound to the same individual.

```
<asks
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
    uri="urn:uuid:...">
  <retrieve id="q6">
    <head>
      <indvar name="x"/>
    </head>
    <body>
      <qand>
        <rqatom>
          <indvar name="x"/>
          <indvar name="y"/>
          <ratom name="loves"/>
        </rqatom>
        <cqatom>
          <indvar name="y"/>
          <catom name="human"/>
        </cqatom>
        <qsameas>
          <indvar name="x"/>
          <indvar name="y"/>
        </qsameas>
     </qand>
    </body>
  </retrieve>
</asks>
```

Instead of `qsameas` one may use `qdifferent` to search some `human` that does not love himself.

Sometimes an explicit projection operator in the query body is required in order to reduce the "dimensionality" of a tuple set when the query answer is computing. For DIG we propose a tag `project` which can be used in any position within a query body and contains a head and a body parts. The following query returns all mothers which do not have a known (i.e. explicitly modeled in an ABox) child:

```
 <asks
     xmlns="http://dl.kr.org/dig/lang/schema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
     http://dig.cs.manchester.ac.uk/schema.xsd"
     uri="urn:uuid:...">
   <retrieve id="q7">
     <head>
       <indvar name="x"/>
     </head>
     <body>
```

```
      <qand>
        <catom>
          <indvar name="x"/>
          <catom name="mother"/>
        </cqatom>
        <qneg>
          <project>
            <head>
              <indvar name="x"/>
            </head>
            <body>
              <rqatom>
                <indvar name="x"/>
                <indvar name="y"/>
                <ratom name="hasChild"/>
              </rqatom>
            </body>
          </project>
        </qneg>
      </qand>
    </body>
  </retrieve>
</asks>
```

When a reasoner has to deal with large ABoxes, iterative query answering can help to improve the performance of query answering. To support the incremental loading the answer tuple by tuple, the statement `retrieve` can have an additional attribute `ntuples` instantiated with the maximum number of tuples which are assumed to be returned. If `ntuples` is not specified, all tuples have to be returned.

```
<asks
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
    uri="urn:uuid:...">
  <retrieve id="q5" ntuples=10>
    [...]
  </retrieve>
</asks>

<asks
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
```

```
    uri="urn:uuid:...">
  <retrieve qid="as9999" ntuples=5/>
</asks>
```

The statement `retrieve` without head and body is used to retrieve the next tuple(s) for a particular query identified with `qid`. The value of `qid` can be a query `id` or an answer set id (`asid`):.

### 8.3 Tell language

In order to tell the reasoner that no more tuples will be requested, we propose the tag `releaseQuery`.

```
<tells
     xmlns="http://dl.kr.org/dig/lang/schema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
     http://dig.cs.manchester.ac.uk/schema.xsd"
     uri="urn:uuid:...">
   <releaseQuery qid="q5"/>
 </tells>
```

### 8.4 Response syntax

The response to a `retrieval` request have an attribute `id` which corresponds with the id of the submitted query. The answer set id (`asid`) will be generated by the reasoner. The response contains tuples of bindings for variables mentioned in the query head. The response head is the same as the head of the corresponding query. The variable bindings follow the `bindings` tag. The head is inserted just for convenience; the reasoner may not reorder components of tuples. For example, the following answer is returned for the query with the id `q2` posed above:

```
 <responses
     xmlns="http://dl.kr.org/dig/lang/schema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
     http://dig.cs.manchester.ac.uk/schema.xsd"
   <bindings id="q2" asid="asid1000">
     <head>
       <indvar name="x"/>
       <indvar name="y"/>
     </head>
     <tuple>
       <individual name="mary"/>
       <individual name="betty"/>
     </tuple>
     <tuple>
```

```
      <individual name="susan"/>
      <individual name="peter"/>
    </tuple>
  </bindings>
</responses>
```

The response to a boolean query is a binding list with an empty tuple for `true` or an empty binding list for `false`.

Within the `bindings` statement, the tag `notifier` can be used for reasoner-specific messages. E.g., considering an incremental query answering, the reasoner can report that the given tuple is the last one:

```
<responses
    xmlns="http://dl.kr.org/dig/lang/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://dl.kr.org/dig/lang/schema
    http://dig.cs.manchester.ac.uk/schema.xsd"
  <bindings id="q2" asid="asid1000">
    <head>
      [...]
    </head>
    <tuple>
      [...]
    </tuple>
    <notifier message="last tuple"/>
  </bindings>
</responses>
```

Other messages are possible. For instance, a reasoner could indicate that subsequent requests for tuple will require considerably more resources. Motivated by this example, a reasoner might decide to return fewer tuples than requested with the attribute `ntuples` (see above).

### 8.5 Taxonomy queries

If only slight modifications are made to a taxonomy, for instance, if only a single axiom is added, applications that need access to the taxonomy have to retrieve the whole taxonomy using single queries in DIG 1.1. In DIG 1.2 there are two new queries defined: one for the retrieval of the whole taxonomy, and another for the incremental retrieval of the taxonomy (i.e., only the parts that have changed are indicated). We illustrate the facilities using an example. Let us assume that in the default Tbox, there is the following axiom.

```
(implies a b)
```

The following DIG 1.2 message is sent to RacerPro (`uri=""` indicates a reference to the default Tbox of RacerPro).

```
<?xml version="1.0" encoding="UTF-8"?>
<asks uri="" xmlns="http://dl.kr.org/dig/2003/02/lang">
   <allConceptNames/>
    <incrementalTaxonomy id="init"/>
</asks>
```

The result is the following.

```
<responses
     xmlns="http://dl.kr.org/dig/2003/02/lang"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://dl.kr.org/dig/2003/02/lang
     http://dl-web.man.ac.uk/dig/2003/02/dig.xsd">
<taxonomy id="init">
<concept>
 <catom name="C"/>
</concept>
<concept>
 <catom name="A"/>
<parents>
 <catom name="C"/>
</parents>
</concept>
</taxonomy>
</responses>
```

Next, two axioms

```
(implies a b)
(implies b c)
```

are added, and the following DIG 1.2 message is sent to RacerPro.

```
<?xml version="1.0" encoding="UTF-8"?>
<asks uri="" xmlns="http://dl.kr.org/dig/2003/02/lang">
    <incrementalTaxonomy id="newb"/>
</asks>
```

The answer returned by RacerPro is:

```
<responses
     xmlns="http://dl.kr.org/dig/2003/02/lang"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://dl.kr.org/dig/2003/02/lang
     http://dl-web.man.ac.uk/dig/2003/02/dig.xsd">
<incrementalTaxonomy id="newb">
<concept>
```

```
 <catom name="A"/>
<addedParents>
 <catom name="B"/>
</addedParents>
<removedParents>
 <catom name="C"/>
</removedParents>
</concept>
<concept>
 <catom name="C"/>
<addedParents>
 <catom name="C"/>
</addedParents>
<removedParents>
 <top/>
</removedParents>
</concept>
<concept>
 <bottom/>
<addedParents>
 <top/>
</addedParents>
<removedParents>
 <catom name="A"/>
</removedParents>
</concept>
</incrementalTaxonomy>
</responses>
```

If there are very few changed in a large ontology, the `incrementalTaxonomy` query reduces communication overhead. Instead of `incrementalTaxonomy` there is also a query `taxonomy` which produces the result of the first `incrementalTaxonomy`.[3]

RacerPro does not delete a knowledge base corresponding to a `releaseKB`. The statement `releaseKB` is sent by tools at the end of the interaction. Many users, however, would like to inspect the KB with RacerPorter afterwards. Thus `releaseKB` keeps the KB on the server. Use `deleteKB`. instead if you generate you own knowledge bases using DIG.

# 9 Support for non-standard inference services

RacerPro 1.9.1 provides a prototypical implementation of the LCS operator (least-common subsumer) for the description logic $\mathcal{ALE}$ with unfoldable Tboxes. Furthermore, the MSC-k (most-specific concept up to k levels of nested existentials) is supported. In order to demonstrate the application of these operators, the following knowledge base is used.

[3] Use `allConceptNames` before you use `taxonomy` in order to trigger that the KB is actually processed. This is a known bug in the beta release.

```
(in-knowledge-base test)

(equivalent x (some r (and a d)))
(equivalent y (some r b))
(implies a b)
(implies c b)

(instance j a)
(instance i (all r (and (some r b) d)))
(related i j r)
(related j k r)
(related k i r)
```

We no apply the LCS operator to two input concepts.

```
(lcs-unfold x (and (all r d) (some r c)))
```

The result is `(some r (and d b))`. If no unfolding is desired, use `lcs` instead of `lcs-unfold`. It is possible to specify the Tbox as an optional last argument (default is the current Tbox).

For some purposes, an extract of the information contained in an Abox might be useful. RacerPro 1.9.1 provides an implementation of the MSC-k operator. MSC-k is applied to an individual and a nesting depth. For instance, given the knowledge base above, the following form is executed.

```
(msc-k i 3)
```

The result is

```
(and (some r (and a
                  (some r (and (some r (and (all r (and (some r b) d))
                                            (some r (and a top))))))))))
```

It is possible to suppy an additional boolean argument which indicates whether for each individual, the direct types are included in the MSC approximation. For instance,

```
(msc-k i 3 t)
```

returns

```
(and (some r
           (and x
                a
                (some r
                      (and a
                           d
                           y
                           (some r (and *top*
                                        (all r (and (some r b) d))
                                        (some r (and x a top))))))))))
```

The default for the third argument is `nil` (indicating that direct types are not to be included in the MSC-k output). Given this boolean argument is specified, there can be a fourth argument, the Abox (which defaults to the current Abox).

# 10 Outlook: Explanation

The RacerPro reasoning engine is currently being extended to provide explanations for unsatisfiable concepts, for subsumption relationships (either unwanted or unexpected), and for unsatisfiable Aboxes. An example for an unsatisfiable Abox is shown in the following example.

```
(in-knowledge-base test)

(implies a b)
(equivalent c (some r a))
(equivalent d (some r b))

(instance i (and c (not d)))

(check-abox-coherence)
```

In Figure 13 it is indicated which axioms in the Tbox and which assertions in the Abox are the culprits for the inconsistency of the Abox. The figure shows a window from the development environment of RacerPro in order to demonstrate what will be provided in the near future with the next beta release. The explanation facilities will be integrated into RacerPorter and RacerEditor.

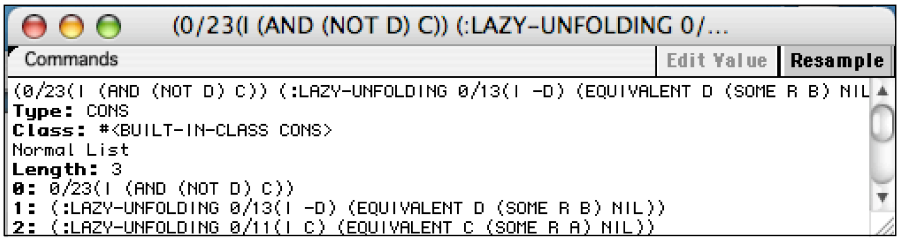


Figure 13: Explanation for an unsatisfiable Abox.

Explanation for query answering is already available (see above) in case a boolean query is answered with false but the expected answer is true. One can use the operator `retrieve-with-explanation` as described above. In the near future, this kind of explanation will also be integrated into the RacerPorter and RacerEditor environments as well.

# References

[KMR04] Holger Knublauch, Mark A. Musen, and Alan L. Rector. Editing description logic ontologies with the protégé owl plugin. In *Description Logics*, 2004.

[KPS+05] Aditya Kalyanpur, Bijan Parsia, Evren Sirin, Bernardo C. Grau, and James Hendler. Swoop: A web ontology editing browser. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):144–153, June 2005.

[LBF+06] C. Lutz, F. Baader, E. Franconi, D. Lembo, R. Möller, R. Rosati, U. Sattler, B. Suntisrivaraporn, and S. Tessaris. Reasoning Support for Ontology Design. In B. Cuenca Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *In Proceedings of the second international workshop OWL: Experiences and Directions*, November 2006. To appear.

[LN05] Thorsten Liebig and Olaf Noppens. OntoTrack: A semantic approach for ontology authoring. *Journal of Web Semantics*, 3(2-3):116–131, October 2005.

[LN06] Thorsten Liebig and Olaf Noppens. Interactive Visualization of Large OWL Instance Sets. In *Proc. of the Third Int. Semantic Web User Interaction Workshop (SWUI 2006)*, Athens, GA, USA, November 2006.

[Ope] OpenCyc. `http://www.opencyc.org/`.

[SK06] Ferdinando Villa Sergey Krivov, Rich Williams. Growl, visual browser and editor for owl ontologies. *Journal of Web Semantics*, 2006.