Extended Query Facilities for Racer and an Application to Software-Engineering Problems

Volker Haarslev[□], Ralf Möller[◇],
Ragnhild Van Der Straeten[△] and Michael Wessel[○]

Concordia University[□]
Montreal, Canada
haarslev@cs.concordia.ca

Technical University of Hamburg-Harburg[◇]
Hamburg-Harburg, Germany
r.f.moeller@tuhh.de

Vrije Universiteit Brussel[△]
Brussels, Belgium
rvdstrae@vub.ac.be

University of Hamburg[○]
Hamburg, Germany
mwessel@informatik.uni-hamburg.de

Abstract

This paper reports on a pragmatic query language for Racer. The abstract syntax and semantics of this query language is defined. Next, the practical relevance of this query language is shown, applying the query answering algorithms to the problem of consistency maintenance between object-oriented design models.

1 Motivation

Practical description logic (DL) systems such as Racer [3] offer a functional API for querying a knowledge base (i.e., a tuple of T-box and A-box). For instance, Racer provides a query function for retrieving all individuals mentioned in an A-box that are instances of a given query concept. Let us consider the following A-box: $\{has_child(alice, betty), has_child(alice, charles)\}$. If we are interested in finding individuals for which it can be proven that a child exists, in the Racer system, the function *concept_instances* can be used. However, if we would like to find all tuples of individuals x and y such that a common parent exists, currently, it is not possible to express this in sound and complete DL systems such as, for instance, Racer. Other logic-based representation systems, such as, e.g., LOOM [6], however, have offered query languages suitable for expressing the second query right from the beginning. In this paper we define syntax and semantics of a query language similar to that of LOOM. Users of description logic systems such as Racer already demonstrated the demand of such a query language for sound and complete DL systems [9], and this paper evaluates the practical relevance of the current implementation for query answering algorithms in Racer-1-7-19 using an application to software engineering problems.

2 The New Racer Query Language - nRQL

In the following we describe the *n*ew *R*acer *Q*uery *L*anguage, also called *nRQL* (pronounce: Nercle). We start with some auxiliary definitions:

Definition 1 (Individuals, Variables, Objects) Let $\mathcal{I}$ and $\mathcal{V}$ be two disjoint sets of *individual names* and *variable names*, respectively. The set $\mathcal{O} =_{def} \mathcal{V} \cup \mathcal{I}$ is the set of *object names*. We denote variable names (or simply variables) with letters $x, y, \ldots$; individuals are named $i, j, \ldots$; and object names $a, b, \ldots$. ■

Query atoms are the basic syntax expressions of nRQL:

Definition 2 (Query Atoms) Let $a, b \in \mathcal{O}$; C be an $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ [4] concept expression, R a role expression, P one of the concrete domain predicates offered by Racer; $f = f_1 \circ \cdots \circ f_n$ and $g = g_1 \circ \cdots \circ g_m$ be feature chains such that f_n and g_m are attributes (whose range is defined to be one of the available *concrete domains* offered by Racer, or f, g are individuals from one of the offered concrete domains which means that $m, n = 1$ and f_1, g_1 are 0-ary attributes). Then, the list of *nRQL atoms* is given as follows:

- Unary concept query atoms: $C(a)$
- Binary role query atoms: $R(a, b)$
- Binary constraint query atoms: $P(f(a), g(b))$
- Unary bind-individual atoms: $bind_individual(i)$
- Unary has-known-successor atoms: $has_known_successor(a, R)$
- Negated atoms: If rqa is a nRQL atom, then so is $\backslash(rqa)$, a so-called *negation as failure atom* or simply *negated atom.* ■

We give some examples of the various atoms and assume throughout the paper that $betty \in \mathcal{I}$. Note that $(woman \sqcap (\neg mother))(betty)$ and $woman(x)$ are unary concept query atoms; $has_child(x, y)$ and $has_child(betty, y)$ are binary role query atoms; `string=`$(has_father \circ has_name(x), has_name(y))$ as well as $\geq(has_age(x), 19)$ are examples of binary constraint query atoms. Finally, $\backslash(woman(x))$ is an example of a negated atom. The rationale for introducing unary bind-individual and has-known-successor atoms will become clear later. Both are related to effects caused by negated atoms. We can now define nRQL queries:

Definition 3 (nRQL Query Bodies, Queries & Answer Sets) A *nRQL Query* has a *head* and a *body*. *Query bodies* are defined inductively as follows:

- Each nRQL atom rqa is a body; and
- If $b_1 \ldots b_n$ are bodies, then the following are also bodies:
  - $b_1 \wedge \cdots \wedge b_n$, $b_1 \vee \cdots \vee b_n$, $\backslash(b_i)$

We use the syntax $body(a_1, \ldots, a_n)$ to indicate that $a_1, \ldots, a_n$ are all the objects ($a_i \in \mathcal{O}$) mentioned in *body*. A *nRQL Query* is then an expression of the form

$$ans(a_{i_1}, \ldots, a_{i_m}) \leftarrow body(a_1, \ldots, a_n),$$

$ans(a_{i_1}, \ldots, a_{i_m})$ is also called the *head*, and $(i_1, \ldots, i_m)$ is an index vector with $i_j \in 1 \ldots n$. A *conjunctive nRQL query* is a query which does not contain any $\vee$ and $\setminus$ operators. ■

Before we consider atoms with variables, we define truth of *ground query atoms.* A ground query atom does not reference any variables. To define truth of ground query atoms, we will need the standard notion of *logical implication* or *logical entailment.* We first start with *positive atoms* – atoms which are not negated:

Definition 4 (Entailment of Positive Ground Query Atoms) Let $\mathcal{K}$ be an $\mathcal{ALCQHI}_{\mathcal{R}^+}(\mathcal{D}^-)$ knowledge base. A knowledge base, KB for short, is simply a T-box/A-box tuple: $\mathcal{K} = (\mathcal{T}, \mathcal{A})$.

A *positive ground query atom rqa (i.e., rqa doesn't reference variables and is not negated)* is logically entailed (or implied) by $\mathcal{K}$ iff every model $\mathcal{I}$ of $\mathcal{K}$ is also a model of *rqa*. In this case we write $\mathcal{K} \models rqa$. Moreover, if $\mathcal{I}$ is a model of $\mathcal{K}$ (rqa) we write $\mathcal{I} \models \mathcal{K}$ ($\mathcal{I} \models rqa$).

We therefore have to specify when $\mathcal{I} \models rqa$ holds. In the following, if *rqa* references individuals i, j, it will always be the case that $i, j \in \mathsf{inds}(\mathcal{A})$. From this it follows that $i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$ and $j^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, for any $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ with $\mathcal{I} \models \mathcal{K}$:

- If $rqa = C(i)$, then $\mathcal{I} \models rqa$ iff $i^{\mathcal{I}} \in C^{\mathcal{I}}$.
- If $rqa = R(i, j)$, then $\mathcal{I} \models rqa$ iff $(i^{\mathcal{I}}, j^{\mathcal{I}}) \in R^{\mathcal{I}}$.
- If $rqa = P(f(i), g(j))$, then $\mathcal{I} \models rqa$ iff
  - $c_i = f^{\mathcal{I}}(i^{\mathcal{I}})$,
  - $c_j = g^{\mathcal{I}}(j^{\mathcal{I}})$,
  - $(c_i, c_j) \in P^{\mathcal{I}}$; moreover,
  - if $f = f_1 \circ \cdots \circ f_n$, then we require that for all $m \in 1 \ldots n-1$ with $k = (f_1 \circ \cdots \circ f_m)^{\mathcal{I}}(i^{\mathcal{I}})$ there is some $j \in \mathsf{inds}(\mathcal{A})$ such that $j^{\mathcal{I}} = k$; and analogously for g.
- If $rqa = (i = j)$, then $\mathcal{I} \models (i = j)$ iff $i^{\mathcal{I}} = j^{\mathcal{I}}$.
- If $rqa = has_known_successor(i, R)$, then $\mathcal{I} \models rqa$ iff for some $j \in \mathsf{inds}(\mathcal{A})$: $\mathcal{I} \models R(i, j)$. ■

It is important to note that the properties of roles and concepts referenced in the query atoms are defined in the knowledge base $\mathcal{K}$. For example, if the role *has_descendant* has been declared as transitive in $\mathcal{K}$, then *has_descendant* will be transitive in the queries as well, since in models of $\mathcal{K}$ $has_descendant^{\mathcal{I}} = (has_descendant^{\mathcal{I}})^+$ must hold. If has_father is declared as a feature, then it will behave as a feature in the queries as well. Also note that, according to Definition 2, atoms of the form $rqa = (i = j)$ are not really query atoms. However, these atoms are used to *replace bind_individual* atoms, see below.

The complex semantic condition enforced on the binary constraint query atoms such as $rqa = P(f(i), g(j))$ with $f = f_1 \circ \cdots \circ f_n$ and $g = g_1 \circ \cdots \circ g_m$ makes it possible

to substitute such an atoms with the conjunction $f_1(i,i_1) \wedge \cdots \wedge f_{n-1}(i_{n-2},i_{n-1}) \wedge g_1(j,j_1) \wedge \cdots \wedge g_{m-1}(j_{m-2},j_{m-1}) \wedge P(f_n(i_{n-1}),g_m(j_{m-1}))$.

Now that we have defined truth of of positive ground query atoms, we can define truth of arbitrary ground query atoms:

Definition 5 (Truth of Ground Query Atoms) Let rqa be a ground query atom. Let $\mathcal{K} = (\mathcal{T},\mathcal{A})$ be a knowledge base (T-box/A-box tuple). A ground atom rqa is either TRUE in $\mathcal{K}$ (we write $\mathcal{K} \models_{NF} rqa$) or FALSE in $\mathcal{K}$ (we write $\mathcal{K} \not\models_{NF}$). The relationship $\models_{NF}$ resp. trueness of ground query atoms is inductively defined as follows:

- If rqa is positive (does not contain "\"): $\mathcal{K} \models_{NF} rqa$ iff $\mathcal{K} \models rqa$
- Otherwise: $\mathcal{K} \models_{NF} \backslash(rqa)$ iff $\mathcal{K} \not\models_{NF} rqa$ ■

It is important to note that for each query body or atom q, q is TRUE iff $\backslash(q)$ is FALSE, and vice versa. Note that this does not hold for the usual entailment relationship. For example, consider the A-box $\{woman(betty)\}$. Given $\mathcal{K} =_{def} (\emptyset,\mathcal{A})$, $woman(betty)$ is TRUE, and $mother(betty)$ is FALSE, since we cannot prove that *betty* is a mother. Thus, $\backslash(mother(betty))$ is TRUE. In contrast, $\neg mother(betty)$ is obviously FALSE. Moreover, $(mother \sqcup \neg mother)(betty) = \top(betty)$ is *not* the same as $(mother(betty)) \vee (\neg mother(betty))$.

In order to check whether $\mathcal{K} \models_{NF} rqa$, we can use the basic consistency checking and A-box retrieval methods offered by Racer. The symbol "$\models_{NF}$" shall remind the reader of the employed "Negation as Failure" semantics (i.e., suppose rqa is positive, then $\mathcal{K} \models_{NF} \backslash(rqa)$ iff $\mathcal{K} \not\models rqa$, which means $\backslash(rqa)$ is TRUE in $\mathcal{K}$, see below for examples).

The truth definition of ground atoms can be extended to complex *ground query bodies* in the obvious way (i.e., $\mathcal{K} \models_{NF} b_1 \wedge \cdots \wedge b_n$ iff $\forall b_i : \mathcal{K} \models_{NF} b_i$, and analogously for $\vee$ and $\backslash$).

Having defined truth of ground query atoms and bodies, we can specify the semantics of queries which are not ground, but first we need one more piece of notation. The rationale behind the next definition is best understood with an example: consider the query $ans(betty) \leftarrow woman(betty)$. The answer to this query should either be $\emptyset$ (in case $\mathcal{K} \not\models woman(betty)$), or $\{(betty)\}$ (in case $\mathcal{K} \models woman(betty)$). A reasonable statement is that $ans(betty) \leftarrow \backslash(woman(betty))$ should be the complement query of $ans(betty) \leftarrow woman(betty)$. The latter one should therefore return the set $\{\,(i) \mid i \in \mathsf{inds}(\mathcal{A})\,\}$, probably without $\{(betty)\}$ (note that $\mathsf{inds}(\mathcal{A})$ returns the set of all individuals mentioned in the A-box $\mathcal{A}$). Thus, within $\backslash(woman(betty))$, *betty* behaves in fact like a variable. To capture this behavior, we replace the individuals in the atoms with representative variables and use (in)equality statements as follows:

Definition 6 (α-Substitution) Let rqa be an atom that contains at most one "\" (note that $rqa = \backslash(\backslash(rqa))$). Denote the set of mentioned individuals in rqa as $\mathsf{inds}(rqa)$. Then, $\alpha(rqa)$ is defined as follows:

- If $\mathsf{inds}(rqa) = \emptyset$, then $\alpha(rqa) =_{def} rqa$.

- If $rqa = bind_individual(i)$, then $\alpha(rqa) =_{def} x_i = i$
- If $rqa = \backslash(bind_individual(i))$, then $\alpha(rqa) =_{def} x_i \neq i$
- If rqa is not a bind-individual atom, then
  - If rqa is *positive* and $\mathsf{inds}(rqa) = \{i, j\}$ (possibly $i = j$), then $\alpha(rqa) =_{def} rqa_{[i \leftarrow x_i, j \leftarrow x_j]} \wedge (x_i = i) \wedge (x_j = j)$.
  - If $rqa = \backslash(rqa')$ is *negative* and $\mathsf{inds}(rqa) = \{i, j\}$ (possibly $i = j$), then $\alpha(rqa) =_{def} \backslash(rqa'_{[i \leftarrow x_i, j \leftarrow x_j]}) \vee (x_i \neq i) \vee (x_j \neq j)$. ■

Note that $rqa_{[i \leftarrow x_i, j \leftarrow x_j]}$ means "substitute i with x_i, and j with x_j". For example, $\alpha(R(i,j)) = R(x_i, x_j) \wedge (x_i = i) \wedge (x_j = j)$, but $\alpha(\backslash(R(i,j))) = \backslash(R(x_i, x_j)) \vee (x_i \neq i) \vee (x_j \neq j)$. We extend the definition of α to query bodies in the obvious way. However, we need to bring the bodies into *negation normal form (NNF)* first, such that "\" appears only in front of atoms. This is simply done by applying *DeMorgan's Law* to the query body (from the given semantics it follows that $\backslash(A \wedge B) \equiv \backslash(A) \vee \backslash(B)$, $\backslash(A \vee B) \equiv \backslash(A) \wedge \backslash(B)$, $\backslash(\backslash(A)) \equiv A$). The semantics of a nRQL query can now be paraphrased as follows:

Definition 7 (Semantics of a Query) Let $ans(a_{i_1}, \ldots, a_{i_m}) \leftarrow body(a_1, \ldots, a_n)$ be a nRQL query q such that *body* is in NNF. Let $\beta(a_i) =_{def} x_{a_i}$ if $a_i \in \mathcal{I}$, and a_i otherwise; i.e., if a_i is an individual we replace it with its representative unique variable which we denote by x_{a_i}. Let $\mathcal{K}$ be the knowledge base to be queried, and $\mathcal{A}$ be its A-box. The *answer set* of the query q is then the following set of tuples:

$$\{ (j_{i_1}, \ldots, j_{i_m}) \mid \exists j_1, \ldots, j_n \in \mathsf{inds}(\mathcal{A}), \forall m, n, m \neq n : j_m \neq j_n, \\ \mathcal{K} \models_{NF} \alpha(body)_{[\beta(a_1) \leftarrow j_1, \ldots, \beta(a_n) \leftarrow j_n]} \}$$

Finally, we state that $\{()\} =_{def}$ TRUE and $\{\} =_{def}$ FALSE. ■

Note that we assume the *unique name assumption (UNA)* for the variables here. However, the implemented query processing engine also offers non-UNA variables (originally meant for breaking up feature chains). For reasons of brevity we decided not to include them in the formal definition here.

Let us briefly discuss some "pathological examples" which explain why we included bind-individual and has-known-successor atoms into nRQL.

Suppose we want to know for which individuals we have explicitly modeled children in the A-box. For this purpose, the query $ans(x) \leftarrow has_know_successor(has_child, x)$ can be used, but also the query $ans(x) \leftarrow has_child(x, y)$. However, now suppose we want to retrieve the A-box individuals which *do not* have a child. The query $ans(x) \leftarrow \backslash(has_child(x, y))$ cannot be used, since first the complement of $has_child(x, y)$ is computed, and then the projection to x is carried out. Thus, $ans(x) \leftarrow \backslash(has_known_successor(x, has_child))$ must be used. Please note that this query is not equivalent to $ans(x) \leftarrow \backslash(\exists has_child.\top(x))$. To see why, suppose we want to query for *mothers* not having any explicitly modeled children in the A-box. Obviously, these mothers cannot be retrieved with $ans(x) \leftarrow \backslash(\exists has_child.\top(x))$, since motherhood implies having a child. But this child need not be explicitly modeled in the

A-box. Thus, the query $ans(x) \leftarrow mother(x) \wedge \backslash(has_known_successor(x, has_child))$ must be used. The syntax $ans(x) \leftarrow mother(x) \wedge has_child(x, \texttt{NIL})$, which we borrowed from the query language of the LOOM system, is also understood by Racer.

We already mentioned that individuals appearing within negated query atoms turn into variables. Suppose $ans(eve) \leftarrow mother(eve)$ returns $\emptyset$. Thus, the query $ans(eve) \leftarrow \backslash(mother(eve))$ will return the complement set w.r.t. all mentioned A-box individuals, e.g. $\{(eve)(doris)(charles)(betty)(alice)\}$. But sometimes, this behavior is unwanted: in this case we can add the additional conjunct $bind_individual(eve)$. We then get $\{(eve)\}$ for $ans(eve) \leftarrow bind_individual(eve) \wedge \backslash(mother(eve))$.

3 An Example from Software Engineering

In [9], we plead for state-of-the-art DL tools having an extensive query language to be able to maintain consistency between object-oriented design models.

The *de facto* modeling language for the analysis and design of object-oriented software applications is UML [7]. The visual representation of this language consists of several diagram types. Those diagrams represent different views on the system under study. We deliberately confine ourselves to three kinds of UML diagrams: class diagrams representing the static structure of the software application, sequence diagrams representing the behavior of the software application in terms of the collaboration between different objects, and state diagrams modeling the behavior of one single object.

State-of-the-art CASE tools have little support for maintaining the consistency between those different diagrams within the same version of a model or between different versions of a model.

Based on a detailed analysis of all the UML concepts appearing in class, sequence and state diagrams, several consistency conflicts are identified and classified. For an overview of this classification, we refer to [8].

In our approach the relevant subset of the UML metamodel, defining class, state and sequence diagrams, is translated into T-box axioms. As such the different user-defined UML diagrams are translated into A-box assertions. Based on two illustrative consistency conflicts, we argued in [9] that checking for inconsistencies demands an extensive query language. This would allow us to specify UML models and consistency rules in a straightforward and generic way.

The *classless instance* conflict, described in [9] is repeated here and the *infinite containment* conflict is introduced.

3.1 Classless instances

The first conflict appears if there are instances in a sequence diagram which do not have any associated class. An example of this conflict is shown in Figure 1, where the object *anATM* is an instance of *ATM* in the sequence diagram on the right side of Figure 1 but this class does not appear in the class diagram on the left side of the same figure.

Classes in a class diagram are represented by the concept *class* in our T-box and an

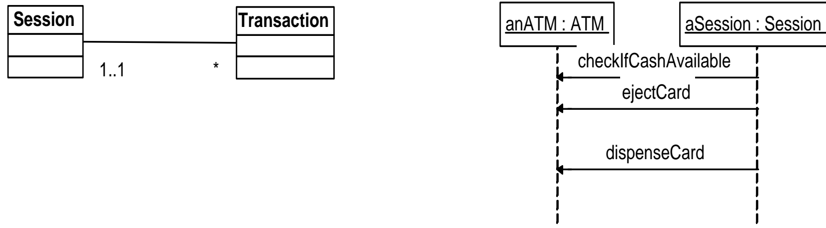

Figure 1: Classless instances conflict

instance by the concept *object*. *instance_of* specifies that an object is an instance of a certain class. *has_classmodel* is a role that contains the associated class diagram of a class. The query language of Racer can now be used to find all the classes that have no related class diagram:

$$\begin{aligned} & ans(x) \\ & \quad \leftarrow \quad class(x) \wedge object(y) \wedge instance_of(y, x) \wedge \\ & \qquad \quad \backslash(has_known_successor(x, has_classmodel)) \end{aligned}$$

This yields the correct result, i.e. the individual ATM bound to the variable x: $\{(ATM)\}$

3.2 Infinite containment

This conflict arises when the composition and inheritance relationships between classes in class diagrams form a cycle and combined with a composition relation, define a class whose instances will, directly or indirectly, be forced to contain at least one instance of the same class, causing an infinite chain of instances.

An example of this conflict is shown in Figure 2, where the class *ASCIIPrintingATM* is transitively a subclass of *ATM* and there exists a composition relation *controls* between those two classes. This composition indicates that every instance of *ATM* controls at least one instance of *ASCIIPrintingATM*. However, an instance of *ASCIIPrintingATM* is also an instance of *ATM* and as such must again control a different instance of *ASCIIPrintingATM* due to the antisymmetric property of a composition relation. (The composition relation is indicated by a black diamond.)

The direct subclass relationship is represented by the *direct_subclass* role which is a sub role of a transitive *subclass* role. A class involved in an association, is linked to this association by the role *has_association* through an association end. An association end has an aggregation kind which can be empty or an aggregation or a composite. This knowledge is represented by the role *has_aggregation* and by the concepts *aggregation* and *composition*. An association has two or more association ends, the role *ends* links the association to its ends. Each association end has a multiplicity, this is expressed by the *has_multiplicity* role. A multiplicity has a range (*has_range*) and this range has a lower and upper bound. These bounds are represented by concrete domain attributes *lower* and *upper* which have type *integer*.

The following query, expressed in nRQL, returns classes which are related by

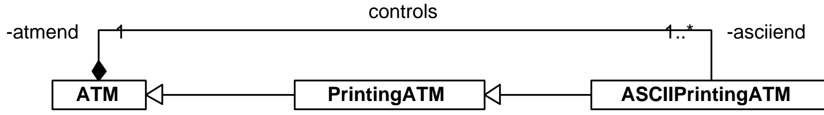

Figure 2: Infinite containment conflict

inheritance and by a composition relation introducing an infinite containment conflict:

$$\begin{aligned} & ans(x,y) \\ & \leftarrow \quad subclass(x,y) \wedge has_association(y, end1) \wedge \\ & \quad has_aggregation(end1, aggreg) \wedge composition(aggreg) \wedge \\ & \quad ends(assoc, end1) \wedge ends(assoc, end2) \wedge has_association(x, end2) \wedge \\ & \quad has_multiplicity(end2, m2) \wedge has_range(m2, r2) \wedge (\exists(lower).\geq_1)(r2) \end{aligned}$$

The result of this query asked to the A-box containing the example of Figure 2 is the answerset $\{(ASCIIPRINTINGATM, ATM)\}$.

With the current nRQL implementation, the answer sets of the queries are correct and delivered within reasonable time limits. Remark however, that this is not a mass data application, which makes this query facility suitable for our purposes.

4 Related Work, Discussion & Conclusion

For querying OWL semantic web repositories, the query language OWL-QL [2] has been proposed, which is the successor of the DAML+OIL query language DQL. Since OWL is basically a very expressive description logic, the proposed query language is relevant in our context as well.

An OWL-QL query is basically a full OWL KB together with a specification which of the referenced URIs in the query "body" (called query pattern in OWL terminology) are to be interpreted as variables. Variables come in three forms: *must-bind, may-bind, and do-not bind variables.* OWL-QL uses the standard notion of logical entailment: query *answers* can be seen as *logically entailed sentences* of the queried KB. Unlike in nRQL, variables cannot only be bound to constants resp. explicitly modeled A-box individuals, but also to complex *OWL terms* which are meant to denote the logically implied domain individual(s) from $\Delta^{\mathcal{I}}$. Thus, if variables in the query patterns are substituted with answer bindings, the resulting sentences are logically entailed by the queried KB. For must-bind variables, bindings *have* to be provided. May-bind variables may provide bindings or not, and *do-not-bind variables* are purely existentially quantified ("existential blanks in the query"). Moreover, OWL-QL queries are full OWL KBs, and this implies that not only extensional queries like in nRQL must be answered, but also "structural queries" are possible, such as "retrieve the subsuming concept names of the concept name father". Similar functions are also offered by Racer's API, but are not available in nRQL.

However, nRQL is not really a subset of OWL-QL. In OWL-QL, neither negation as failure nor *disjunctive A-boxes* can be expressed. Moreover, binary constraint query

atoms of nRQL as well as negated has-known-successor query atoms "are missing" in OWL-QL. The latter ones have in fact been requested by the first users of the nRQL implementation. This clearly indicates that a limited kind of *autoepistemic or closed-world query facilities* should be present in a DL query language. Negation as failure atoms are useful to measure the completeness of the current modeling in an A-box, and this is demanded by users. Thus, it might be more convincing to use a different semantics for "logical implication of queries" in the first place. Such a notion has been given in terms of the so-called K-operator [1], which has been used for the query language $\mathcal{ALCK}$. Roughly speaking, one could state that already *concept_instances* uses the K-operator in front of the concept whose instances are to be retrieved. A more detailed analysis of these relationships is left for future work.

Horrocks and Tessaris [5] also consider conjunctive queries for DL systems. They use two kinds of variables. Must-bind variables are called *distinguished variables*; they are bound to explicitly mentioned A-box individuals, like in nRQL. Similarly to the don't-bind variables in OWL-QL, the *non-distinguished variables* are treated as "existential blanks". Only bindings of distinguished variables are listed in the answerset of a query. Considering DLs of less expressivity than OWL, they observe that the non-distinguished variables of a query can in fact be *removed* by using a *rolling-up* technique without affecting logical entailment. For example, the query $ans(x) \leftarrow R(x,y) \wedge C(y)$ (were only x is distinguished) would be rolled-up into the query $ans(x) \leftarrow \exists R.C(x)$. They also observe that for a DL which has the tree-model property and which does not offer inverse roles, a variable participating in a *join* must in fact be a distinguished variable; e.g. the variable y in $ans(x) \leftarrow R(x,y) \wedge R(z,y)$.

In nRQL, the variables are always distinguished. The query $ans(x,y) \leftarrow R(x,y) \wedge C(y)$ yields $\emptyset$ over the A-box $\{\exists R.C(k)\}$. However, $ans(x) \leftarrow (\exists R.C)(x)$ could be used instead. Obviously, a rolling-up procedure could be implemented as an additional front-end processor for nRQL as well. However, another side-effect caused by the distinction of two kinds of variables is that the generation of answer tuples can no longer be understood as a simple projection. For example, the query $ans(x) \leftarrow R(x,y) \wedge C(y)$ returns $\{(k)\}$ if x is distinguished and y is non-distinguished, but its result is, un-intuitively, not the projection of $ans(x,y) \leftarrow R(x,y) \wedge C(y)$ to x, since this yields $\emptyset$ (note that x, y must be both distinguished, since they appear in ans). Even $ans(x) \leftarrow C(x)$ returns $\emptyset$ (like *concept_instances*(C)), and this holds for the query language in [5] as well. For OWL-QL, however, a possible binding for x might be the *concept* $C \sqcap \exists R^{-1}.\{k\}$. It should be noted that the nRQL query processing engine as well as the whole pragmatic approach for querying DL A-boxes is not dependent on any specific DL system. The same query processing engine would in fact also run with any other DL system offering A-boxes, probably without binary constraint query atoms in case the DL system does not provide concrete domains. The nRQL engine is implemented using the documented API of Racer only, and it is therefore a true "add on" whose implementation does not require any reference to the internal data structures or reasoning algorithms of Racer.

Since the beginning of the Racer endeavor, users were asking for more expressive querying facilities. The presented nRQL is a first step towards satisfying these needs. We substantiated this thesis by presenting an application from the realm of model-

based software engineering, for which it is crucial that expressive query languages are available. The new query language is an integral part since Racer-1-7-16.

References

[1] Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, and Andrea Schaerf. Adding epistemic operators to concept languages. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 342–353. Morgan Kaufmann, San Mateo, California, 1992.

[2] R. Fikes, P. Hayes, and I. Horrocks. OWL-QL - a language for deductive query answering on the semantic web. Technical Report KSL-03-14, Knowledge Systems Lab, Stanford University, CA, USA, 2003.

[3] V. Haarslev and R. Möller. Racer system description. In *International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy.*, 2001.

[4] V. Haarslev, R. Möller, and M. Wessel. The description logic $\mathcal{ALCNH}_{R^+}$ extended with concrete domains: A practically motivated approach. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings of the International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy*, Lecture Notes in Computer Science, pages 29–44. Springer-Verlag, June 2001.

[5] Ian Horrocks and Sergio Tessaris. Querying the semantic web: a formal approach. In Ian Horrocks and James Hendler, editors, *Proc. of the 13th Int. Semantic Web Conf. (ISWC 2002)*, number 2342 in Lecture Notes in Computer Science, pages 177–191. Springer-Verlag, 2002.

[6] Robert MacGregor and David Brill. Recognition algorithms for the LOOM classifier. In *Proc. of the 10th Nat. Conf. on Artificial Intelligence (AAAI'92)*, pages 774–779. AAAI Press/The MIT Press, 1992.

[7] Object Management Group. Unified Modeling Language specification version 1.5. formal/2003-03-01, March 2003.

[8] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between UML models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 326–340. Springer, 2003.

[9] Ragnhild Van Der Straeten, Jocelyn Simmonds, and Tom Mens. Detecting inconsistencies between UML models using description logic. In Diego Calvanese, Giuseppe De Giacomo, and Enrico Franconi, editors, *Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy September 5-7, 2003*, volume 81 of *CEUR Workshop Proceedings*, pages 260–264, 2003.