A Query Language for Logic Architectures

Anton Malykh and Andrei Mantsivoda

Irkutsk State University, Irkutsk, 664003, Russia

Abstract. In this paper we consider the impact of the Semantic Web and logical means on a wide range of developers solving traditional tasks on the WWW. How to make the 'elite' logic tools acceptable for ordinary developers? How to incorporate a wide range of users in the space of the Semantic Web? These and some other questions are considered here and certain proposals are made. In particular we are based on the conception of a logic architecture as a stratified description logic system, and introduce an ontology query language working within logic architectures.

1 Introduction

The tools of the Semantic Web are successfully applied to solving a number of problems, which demand sophisticated logical descriptions and strong logical inference 'engines'. On the other hand, there is a wide-spread opinion that due to the complexity and heaviness of underlying logics, ontologies can not be successfully applied to solving 'lightweight' problems consisting mostly of object processing. It is true that the existing ontology systems can not compete with, say, data base management systems on this kind of tasks. And it is a pity, because this does not allow the Semantic Web to have a significant impact on 'everyday' web resources development, though it is very important if to keep in mind the initial aims of the SW. If the overwhelming majority of practical applications have nothing in common with the SW, it is impossible to 'reorganize' the Web by the SW's elegant and strong conceptions and tools.

The high comprehension barrier between conventional developers and logics, on which the SW is heavily based, is also a problem, because the things that are done by the ordinary developers should be at least compatible with the SW principles and add value to the SW environment. This means that while producing new data, the conventional developer makes it in the form, which is compatible with the logical formalisms and can be integrated in the SW context.

In [1] we consider a conception of a logic architecture, which in particular tries to tackle the problem outlined above. The idea here is to stratify the general logical formalism (e.g. a strong description logic like $\mathcal{SHOIN}(D)$) in such a way that (1) each stratum is responsible for a specific kind of tasks and/or users (while the higher layers can be used for sophisticated and advanced knowledge management, the lower layers can be employed by the wide range of users and developers); (2) each stratum is supplied with special interfaces and programming methods, which implement the scenarios of work within the stratum; (3) the architecture is supplied with tools/formalisms, which work at each stratum

A. Pnueli, I. Virbitskaite, and A. Voronkov (Eds.): PSI 2009, LNCS 5947, pp. 294–305, 2010.
© Springer-Verlag Berlin Heidelberg 2010

and 'glue' the strata together. Among the tools, which can work at each stratum, a query language plays a key role. The basic feature of this language is to be acceptable for the conventional developers.

In this paper we introduce a query language (named `BoxQL`), which meets the conditions stated above. `BoxQL` is designed in the XPath-like style at the both syntactic and operational levels, and looks familiar to many people. The idea behind `BoxQL` is that it should have identical behavior at any level of the logic architecture. `BoxQL` is intended for logic architectures, which are based on $\mathcal{SHOIN}(D)$ [2] as a 'maximal' logic. $\mathcal{SHOIN}(D)$ is attractive, because it determines the semantics of the web ontology language OWL DL [3].

2 Preliminaries

The languages and logics we consider in this paper are used to describe the *worlds*, which are habitats for *objects* (individuals like John, planet Jupiter and this paper). Objects can be grouped in *concepts* (or classes like Mammals, Planets and Information resources). Objects are connected with each other through *object properties* (or roles – like hasChild or spouse).

The languages describing the worlds are based on vocabularies. A *vocabulary* is a structure $V = \langle T_C, T_R, Id \rangle$ in which T_C, T_R and Id are finite pairwise disjoint sets. T_C is called the set of concept types, and T_R the set of relations. Id is the set of individual names (identifiers). Each relation $r \in T_R$ is associated with a non-negative integer $\sigma(r)$, which is called the arity of r. T_C and T_R are partially ordered in such a way that T_C has a maximum element $\top$, and the relations with different arities are not comparable in T_R.

Objects can have attributes (e.g. age or price). Attributes are assigned to objects by *datatype properties*. The values of datatype properties are taken in *datatype domains* (e.g. integers or strings). Knowledge about the worlds is stored in various *namespaces*. In order to introduce namespaces and data types we augment the notion of a vocabulary in the following way. Let $NS = \{ns_1, \dots, ns_k\}$ be the set of namespaces and $D = \{d_1, \dots, d_m\}$ the set of datatypes.

Definition 1 (A namespaced vocabulary). *Let $\mathcal{V} = \langle T_C, T_R, Id \rangle$ be a vocabulary. Then $\mathcal{V}_{NS}^{D} = \langle T_C, T_R, Id, NS, D \rangle$ is a namespaced vocabulary with names $L = T_C \cup T_R \cup Id \cup D$, if the following conditions hold:*

1. *L is divided into k pairwise disjoint subsets $L = \bigcup_{i=1}^{k} L^{ns_i}$, such that $L^{ns_i} = T_C^{ns_i} \cup T_R^{ns_i} \cup Id^{ns_i} \cup D^{ns_i}$. To indicate that a name nm belongs to a namespace ns_i (that is, $nm \in L^{ns_i}$), we write $ns_i\!:\!nm$.*
2. *T_R is divided into three pairwise disjoint sets T_R^{o}, T_R^{t}, and T_R^{d} of object properties, datatype properties and domain specific relations, respectively, such that $T_R = T_R^{\mathrm{o}} \cup T_R^{\mathrm{t}} \cup T_R^{\mathrm{d}}$.*
3. *T_C contains a concept type c_{ns} for each $ns \in NS$.*

Informally, a concept type c_{ns} denotes the set of objects with names belonging to the namespace ns. We assume that all c_{ns} belong to the initial vocabulary $\mathcal{V}$. Since we work in the context of $\mathcal{SHOIN}(D)$, we have $\sigma(r) = 2$ for each $r \in T_R^{\mathrm{o}} \cup T_R^{\mathrm{t}}$. The relations of T_R^{d} can have arbitrary arities.

Definition 2 (A datatype domain). *A datatype domain* $\mathcal{D} = \langle D_1, , \dots, , D_m; \mathcal{L}_D \rangle$ *is an algebra of the language* $\mathcal{L}_D$*, where each* D_i*,* $1 \leq i \leq m$*, is the set of values of the datatype* d_i*.*

Let $|\mathcal{D}| = D_1 \cup \ldots \cup D_m$. We denote by $\mathrm{Term}_{\mathcal{D}+T}$ the set of all ground terms of the language $\mathcal{L}_D \cup T_R^{\mathrm{t}} \cup \{\,.\,\}$, in which elements of $|\mathcal{D}|$, T_R^{t} and the dot '.' play the role of constants. $\mathrm{Term}^*_{\mathcal{D}+T}$ denotes the set of all finite sequences of elements of $\mathrm{Term}_{\mathcal{D}+T}$, that is, if $v_1, \dots, v_k \in \mathrm{Term}_{\mathcal{D}+T}$ then $(v_1, \dots, v_k) \in \mathrm{Term}^*_{\mathcal{D}+T}$.

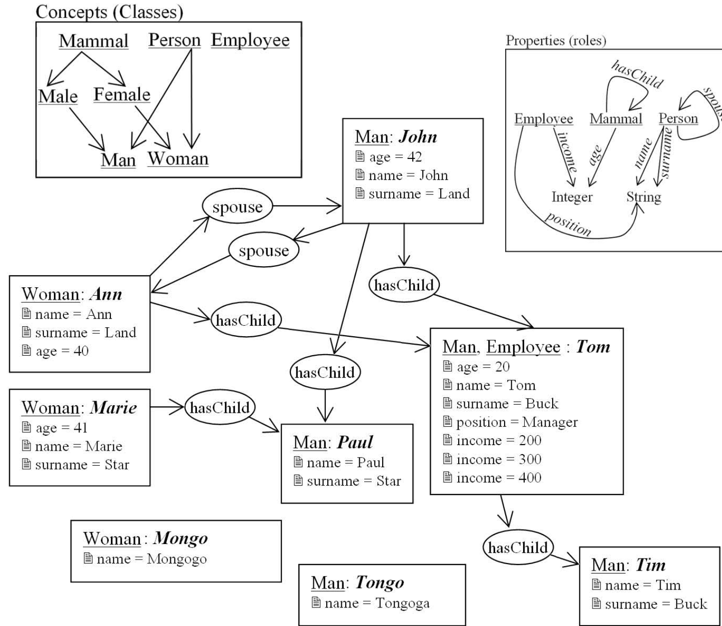

Fig. 1. The world of people $\mathcal{PW}$

Example 1. In figure 1 a simple world of people ($\mathcal{PW}$) is introduced. The vocabulary $\mathcal{V}_{people}$ of this domain consists of

$$T_C = \{\texttt{Mammal, Female, Male, Person, Man, Woman, Employee}\},$$

with the order `Woman` $\leq$ `Female` $\leq$ `Mammal`, `Man` $\leq$ `Male` $\leq$ `Mammal`, `Woman` $\leq$ `Person`, `Man` $\leq$ `Person`.

$$T_R = \{\texttt{spouse, hasChild, age, name, surname, position, income}\}$$

$$Id = \{\texttt{Ann, Marie, Mongo, Tom, Paul, Tim, Tongo, John}\}$$

Let us describe this world in two namespaces `http://people/basic` (in which the basic terminology about people is defined), and `http://people/tribe` (containing only data about two members of a tribe, named Mongogo and Tongoga), plus the datatype namespace `http://www.w3c.org/2001/XMLSchema`. For these namespaces we use the shortcuts (prefixes) `p`, `t` and `x`, respectively. Then, based on $\mathcal{V}_{\mathcal{PW}}$ with augmented $T'_C = T_C \cup \{c_{\texttt{p}}, c_{\texttt{t}}, c_{\texttt{x}}\}$, we can construct $\mathcal{V}^D_{\mathcal{PW}}$:

$NS = \{\texttt{p, t, x}\}$, $D = \{\texttt{x:integer, x:string}\}$
$T^{\mathrm{o}}_R = \{\texttt{p:spouse, p:hasChild}\}$
$T^{\mathrm{t}}_R = \{\texttt{p:age, p:name, p:surname, p:position, p:income}\}$
$T^{\mathrm{d}}_R = \{\texttt{+, > and other rels \& ops of x:integer and x:string}\}$
$Id^p = \{\texttt{p:Ann, p:Marie, p:Tom, p:Paul, p:Tim, p:John}\}$
$Id^t = \{\texttt{t:Mongo, t:Tongo}\}$, $Id^x = \emptyset$

Definition 3 (The language of $\mathcal{SHOIN}(D)$). *If $c \in T_C$ then c is a named concept of $\mathcal{SHOIN}(D)$. Named concepts are concepts. If a, b are concepts, $id \in Id$, $r \in T^{\mathrm{o}}_R \cup T^{\mathrm{t}}_R$, $p \in T^{\mathrm{d}}_R$ then $a \sqcap b$, $a \sqcup b$, $\neg a$, $\exists r.a$, $\forall r.a$, $\leq_n r$, $\geq_n r$, $\exists(x_1, \ldots, x_n).p$, $\forall(x_1, \ldots, x_n).p$, $\{id\}$ are also concepts. If $r \in T^{\mathrm{o}}_R$ then $\exists r'.a$ and $\forall r'.a$ are concepts, where $r' \in \{r^+, r^-, r^\pm\}$.*

$\mathcal{SHOIN}(D)$ is a description logic, which is very close to the Web ontology language OWL DL. The first $\mathcal{S}$ stands in its name for the basic logic $\mathcal{ALC}$ augmented with transitive roles. In $\mathcal{ALC}$ the basic description logic constructs (concepts, roles, disjunction $\sqcup$, conjunction $\sqcap$, negation $\neg$ and role quantifiers) are introduced. $\mathcal{H}$ stands for role hierarchies, $\mathcal{O}$ denotes objects, which can be represented explicitly, $\mathcal{I}$ stands for inverse roles, and $\mathcal{N}$ for simple number restrictions. ($\mathcal{D}$) means that datatype properties are also allowed.

3 `BoxQL`: A Query Language

`BoxQL` is designed in an object-oriented style: it considers the collections of individuals in ontologies as a network of interconnected objects (like in Fig. 1). In [1] we introduce the notion of an object-oriented projection, which formally describes such an understanding of an object network. OO-projections are simple sublogics of $\mathcal{SHOIN}(D)$. The nature of `BoxQL` is traversal: its queries are 'walking along' the network of objects and collecting necessary data. This style is familiar, intuitively easy and acceptable for many people, because it resembles XPath, directory paths in file systems, and 'dotted' expressions in the object-oriented languages. On the other hand, `BoxQL` is upward compatible with more sophisticated and elite logic techniques. And the queries in `BoxQL` are actually 'encoded' formulas of $\mathcal{SHOIN}(D)$, though a flavor of logic is concealed in them. Like in XPath the general structure of `BoxQL` queries is

$$\texttt{step}_1\texttt{[pred}_1\texttt{]/.../step}_k\texttt{[pred}_k\texttt{]}$$

where predicates in square brackets are optional. A `BoxQL` query produces a sequence of objects or data values gathered in $\mathcal{KB}$, which satisfy its conditions.

Example 2. Let us take an example from $\mathcal{PW}$. Imagine that we want to find a spouse of some man who is 40 and has a grandson. Here is a BoxQL query

Query: `@man/spouse[age = 40 and hasChild/hasChild[@man]]`
Result: `{Ann}`

The main path `@man/spouse` of this query collects all man's spouses, because the value of a path is always the value of its rightmost step. The names of classes are qualified in BoxQL with '@'. The namespace p is assumed default, thus its prefix is omitted. The predicate in the square brackets allows us to select among the man's spouses those persons who are 40 and for whom the query `hasChild/hasChild[@man]` produces a non-empty sequence of objects. Now let us compare the above query with the following one:

Query: `@man/spouse[age = 40]/hasChild/hasChild[@man]`
Result: `{Tim}`

Here we move along the same path as in the first query. But the result is different, because the main path now is `@man/spouse/hasChild/hasChild` and its rightmost step gives Tim.

Note that the both queries can be expressed by the formulas of $\mathcal{SHOIN}(D)$:

1. $\exists spouse^{-}.Man \sqcap \exists age.\{40\} \sqcap \exists hasChild.\exists hasChild.Man$ and
2. $Man \sqcap \exists hasChild^{-}.\exists hasChild^{-}.(\exists age.\{40\} \sqcap \exists spouse^{-}.Man)$

We can see the fundamental difference between the queries and the corresponding DL formulas. The DL formulas just describe the qualities of a searched object, whereas the traversal queries show how to get it from scratch (this is why we need to use the role inversions in the formulas). In practice, the style of writing queries in BoxQL is closer to navigation over description graphs. BoxQL is based on the idea that the user 'walks along' the ontology objects' network, and points out, which data must be collected during this walk. For collecting necessary data the user can employ the *fields*, like in the following example

Query: `@p:person as Granparent/hasChild/hasChild`
Result: `{(Tim, Grandparent=John), (Tim, Grandparent=Ann)}`

Now the solution contains not only the value of the rightmost step (Tim) but also the value of the field Grandparent corresponding to Tim. The way of formulating queries in BoxQL is also close to writing dotted expressions in OO programming languages, though the dot itself serves in BoxQL as an analog of Java's 'this':

Query: `age[. > 30]`
Result: `{40, 41, 42}`

The result is a datatype sequence, and '¿' $\in T_R^{\mathrm{d}}$. We do not distinguish a constant singleton (x) and the value x, so instead of `[. > (30)]` we can write `[. > 30]`.

An atomic step `ns:*` selects all objects named within the namespace `ns`:

Query: `t:*/name`
Result: `{"Tongoga", "Mongogo"}`

An atomic step −`ns:ro` denotes the inverses of object properties:

Query: −`hasChild`
Result: `{John, Marie, John, Ann, Tom}`

Intuitively '−' means here that we move from children to parents. John occurs twice, because he is the father of two persons. `BoxQL` also allows us to explicitly use the names of objects like in

Query: `&Tom/income`
Result: `{200, 300, 400}`

Using sequences as steps we can combine the sets of elements like in the following example, in which '*' stands for 'all objects':

Query: `(&john, @woman/t:*, *[income])/name`
Result: `{"John", "Mongogo", "Tom"}`

The expression `(h ! [r1])` implements in `BoxQL` the ∀–quantifier of DLs. The next query selects individuals whose children are *all* boys (following the semantics of ∀ in the DLs `!` collects also individuals who do not have children):

Query: `*[hasChild ! [@man]]`
Result: `{John, Ann, Marie, Paul, Tom, Tim, t:Tongo, t:Mongo}`

Since DLs are based on the open-world paradigm, a `BoxQL`-query actually produces a sequence of objects, which *are known* to satisfy the query's conditions. Thus, we should be careful with the negation because its use in open knowledge bases can set dangerous non-monotone traps. For instance, the query `*[not hasChild]` asks to find those *known* objects, which are *unknown* to have children, and further updates in the knowledge base can reduce the resulting sequence. We do not put restrictions on the negation in the general formalism. There are various ways how to tackle such problems, say, with the help of the epistemic operator $\mathcal{K}$ (see [4], section 6.2.3). In particular, we can restrict the negation only to those concepts c, the knowledge about which is complete, i.e. $\neg\mathcal{K}c \sqsubseteq \mathcal{K}\neg c$.

4 The Semantics of `BoxQL`

First, we formally define the `BoxQL` syntax.

Definition 4 (`BoxQL` **syntax).** *Let $\mathcal{V}^D_{NS}$ be a namespaced vocabulary. We define the sets of steps $\mathcal{S}$, paths $\mathcal{P}$ and predicates $\mathcal{R}$ of* `BoxQL` *as follows:*

1. *The atomic steps of* `BoxQL` *are: (a) $\mathtt{*} \in \mathcal{S}$ and $\mathtt{ns{:}*} \in \mathcal{S}$ for each $\mathtt{ns} \in NS$; (b) if $\mathtt{ns{:}c} \in T_C$, then $\mathtt{@ns{:}c} \in \mathcal{S}$; (c) if $\mathtt{ns{:}ro} \in T^{o}_{R}$ then $\mathtt{ns{:}ro}, \mathtt{-ns{:}ro} \in \mathcal{S}$; (d) if $\mathtt{ns{:}rd} \in T^{t}_{R}$ then $\mathtt{ns{:}rd} \in \mathcal{S}$; (e) if $\mathtt{ns{:}id} \in Id$, then $\mathtt{\&ns{:}id} \in \mathcal{S}$.*
2. *$\mathcal{S} \subseteq \mathcal{P}$. If $\mathtt{h} \in \mathcal{P}$ and $\mathtt{s} \in \mathcal{S}$ then $\mathtt{h/s} \in \mathcal{P}$.*
3. *$Term^*_{\mathcal{D}+T} \subseteq \mathcal{S}$. If $\mathtt{h}_1, \ldots, \mathtt{h}_k \in \mathcal{P}$ then the sequence $(\mathtt{h}_1, \ldots, \mathtt{h}_k) \in \mathcal{S}$. A sequence is called constant if all $h_i \in Id$.*

4. *If* $\mathtt{s} \in \mathcal{S}$ *and* $\mathtt{r} \in \mathcal{R}$*, then* $\mathtt{s[r]} \in \mathcal{S}$*.*
5. $\mathcal{P} \subseteq \mathcal{R}$*.* $Rel_{\mathcal{D}+T} \subseteq \mathcal{R}$*.*
6. *If* $\mathtt{h} \in \mathcal{P}$*,* $\mathtt{r1},\ \mathtt{r2} \in \mathcal{R}$ *and* `c` *is a constant sequence, then* `(r1 and r2)`*,* `(r1 or r2)`*,* `(not r1)`*,* `(r1 = c)`*,* `(h ! [r1])` *belong to* $\mathcal{R}$*.*

A `BoxQL`-*query* is any $\mathtt{h} \in \mathcal{P}$. A `BoxQL`-query fetches basic data about objects in the underlying knowledge base $\mathcal{KB}$, in which `BoxQL` works. In other words, $\mathcal{KB}$ is a parameter of `BoxQL`. The strength of $\mathcal{KB}$ depends on the stratum of the logic architecture, in which `BoxQL` works at the moment.

Let $\mathcal{V}^D_{NS}$ be a namespaced vocabulary and $\mathcal{KB} = \langle |\mathcal{KB}|, \models \rangle$ a knowledge base of this vocabulary with the interpretation $I : \mathcal{V}^D_{NS} \mapsto \mathcal{KB}$. $|\mathcal{KB}|$ is the set of objects stored in $\mathcal{KB}$. We assume that $\models$ allows us to check whether $\mathtt{ns{:}c}^I(o_1)$, $\mathtt{ns{:}ro}^I(o_1, o_2)$, $\mathtt{ns{:}rd}^I(o_1, v)$ are true for any $o_1, o_2 \in |\mathcal{KB}|$, $\mathtt{ns{:}c} \in T_C$, $\mathtt{ns{:}ro} \in T^{\mathrm{o}}_R$, $\mathtt{ns{:}rd} \in T^{\mathrm{t}}_R$ and $v \in |\mathcal{D}|$. Predicates from $\mathrm{Rel}_{\mathcal{D}+T}$ are also evaluated by $\models$.

The (naive) procedural semantics of `BoxQL` is defined in the form of calculi. Note that the paths of `BoxQL` focus on collecting objects, while the predicates in square brackets do the opposite work: they filter out the objects not satisfying certain conditions. This means that we need to have separate, though mutually defined, sub-calculi for paths and predicates, which are called the $\star$-calculus and the $\diamond$-calculus, respectively. For technical reasons we treat a path $\mathtt{h}_1/\ldots/\mathtt{h}_k$ as if it is obtained from the empty path ϵ by the multiple applications of the left-associative operator '/': $(\ldots(\epsilon/\mathtt{h}_1)/\mathtt{h}_2)/\ldots/\mathtt{h}_k)$.

The derived objects of the $\star$-calculus have the form $\mathtt{h}\langle\mathcal{C}\rangle$, where `h` is a path and $\mathcal{C}$ a finite sequence of elements from $|\mathcal{KB}|$. We say that a sequence $\mathcal{A}$ is the *answer* to a path $\mathtt{h} \in \mathcal{P}$ on a sequence $\mathcal{C}$ (denoted $\mathtt{h}\langle\mathcal{C}\rangle \vdash_\star \mathcal{A}$), if there exists a derivation sequence $\mathtt{h}\langle\mathcal{C}\rangle, \mathtt{h}_1\langle\mathcal{C}_1\rangle, \ldots, \epsilon\langle\,\mathcal{A}\,\rangle$ such that ϵ is the empty path, and every $\mathtt{h}_i\langle\,\mathcal{C}_i\,\rangle$ is obtained from the previous one by the application of some $\star$-rule. There are two possibilities: $\mathcal{A} \subseteq |\mathcal{KB}|$ (the result is a sequence of objects), and $\mathcal{A} \subseteq |\mathcal{D}|$ (the result is a sequence of datatype values).

Let $\mathtt{h} \in \mathcal{P}$, $\mathtt{r} \in \mathcal{R}$, $\mathtt{s} \in \mathcal{S}$, $o \in |\mathcal{KB}|$, $x \in |\mathcal{KB}| \cup |\mathcal{D}|$, $\mathcal{C} \subseteq |\mathcal{KB}|$, $\mathcal{A} \subseteq |\mathcal{KB}|$ or $\mathcal{A} \subseteq |\mathcal{D}|$, $\mathtt{ns{:}ro} \in T^{\mathrm{o}}_R$, $\mathtt{ns{:}rr} \in T^{\mathrm{o}}_R \cup T^{\mathrm{t}}_R$. For a ground formula f, $\models f$ means that $\mathcal{KB}$ *'knows'* that f is true, and $\not\models f$ means that $\mathcal{KB}$ *'knows'* that f is false. Here are the rules of the $\star$-calculus:

$$\star\ \frac{\mathtt{*/h}\langle\,\mathcal{C}\,\rangle}{\mathtt{h}\langle\,\mathcal{C}\,\rangle} \qquad \star\ \frac{\mathtt{ns{:}*/h}\langle\,\mathcal{C}\,\rangle}{\mathtt{h}\langle\,\mathcal{C} \cap \{o \mid\ \models \mathtt{c}^I_{ns}(o)\}\,\rangle} \qquad \star\ \frac{\mathtt{ns{:}c/h}\langle\,\mathcal{C}\,\rangle}{\mathtt{h}\langle\,\mathcal{C} \cap \{o \mid\ \models \mathtt{ns{:}c}^I(o)\}\,\rangle}$$

$$\star\ \frac{\mathtt{ns{:}rr/h}\langle\,\mathcal{C}\,\rangle}{\mathtt{h}\langle\,\{x \mid \exists o \in \mathcal{C} : \models \mathtt{ns{:}rr}^I(o, x)\}\,\rangle} \qquad \star\ \frac{\mathtt{-ns{:}ro/h}\langle\,\mathcal{C}\,\rangle}{\mathtt{h}\langle\,\{o \mid \exists o' \in \mathcal{C} : \models \mathtt{ns{:}ro}^I(o, o')\}\,\rangle}$$

$$\star\ \frac{\mathtt{s[r]/h}\langle\,\mathcal{C}\,\rangle}{\mathtt{h}\langle\,\{o \mid o \in \mathcal{C}' : \mathtt{s}\langle\mathcal{C}\rangle \vdash_\star \mathcal{C}' \text{ and } \mathtt{r}(o) \vdash_\diamond \mathtt{true}\}\,\rangle} \qquad \star\ \frac{(x_1, \ldots, x_k)\mathtt{/h}\langle\,\mathcal{A}\,\rangle}{\mathtt{h}\langle\,\mathcal{A} \cap \{x^I_1, \ldots, x^I_k\}\,\rangle}$$

In the next $\diamond$-calculus the derived objects are `true` and `false`, where `true` denotes any non-empty set of elements and `false` is represented by $\emptyset$. To prove that a predicate $\mathtt{r} \in \mathcal{R}$ holds on an element (an object or a datatype value) x, we need to derive $\mathtt{r}(x) \vdash_\diamond \mathtt{true}$ in the $\diamond$-calculus, and to derive $\mathtt{r}(x) \vdash_\diamond \mathtt{false}$, if we want to refute it. The $\diamond$-rules are:

$$\diamond\ \frac{\mathtt{h}\langle\{x\}\rangle\vdash_\star\{x_1,\ldots,x_k\}\quad \forall i:\mathtt{r}(x_i)\vdash_\diamond \mathtt{true}}{\mathtt{h}\ !\ [\mathtt{r}](x)\vdash_\diamond \mathtt{true}} \qquad \diamond\ \frac{\mathtt{r}_1(x)\vdash_\diamond res_1\quad \mathtt{r}_2(x)\vdash_\diamond res_2}{(\mathtt{r}_1\ \mathtt{and}\ \mathtt{r}_2)(x)\vdash_\diamond res_1\wedge res_2}$$

$$\diamond\ \frac{\mathtt{h}\langle\{x\}\rangle\vdash_\star\{x_1,\ldots,x_k\}\quad \exists i:\mathtt{r}(x_i)\vdash_\diamond \mathtt{false}}{\mathtt{h}\ !\ [\mathtt{r}](x)\vdash_\diamond \mathtt{false}} \qquad \diamond\ \frac{\mathtt{r}_1(x)\vdash_\diamond res_1\quad \mathtt{r}_2(x)\vdash_\diamond res_2}{(\mathtt{r}_1\ \mathtt{or}\ \mathtt{r}_2)(x)\vdash_\diamond res_1\vee res_2}$$

$$\diamond\ \frac{\mathtt{r}(x)\vdash_\diamond res}{(\mathtt{not}\ \mathtt{r})(x)\vdash_\diamond \neg res} \qquad \diamond\ \frac{\mathtt{h}\langle\{x\}\rangle\vdash_\star \mathcal{A}}{(\mathtt{h}=\mathtt{c})(x)\vdash_\diamond \mathcal{A}\cap c^I} \qquad \diamond\ \frac{\mathtt{h}\langle\{x\}\rangle\vdash_\star \mathcal{A}}{\mathtt{h}(x)\vdash_\diamond \mathcal{A}}$$

$$\diamond\ \frac{\exists\ \mathtt{p}_e\!\uparrow\! x:\ \models \mathtt{p}_e\!\uparrow\! x}{\mathtt{p}_e(x)\vdash_\diamond \mathtt{true}} \qquad \diamond\ \frac{\forall\ \mathtt{p}_e\!\uparrow\! x:\ \not\models \mathtt{p}_e\!\uparrow\! x}{\mathtt{p}_e(x)\vdash_\diamond \mathtt{false}}$$

Here $res, res_i \in \{\mathtt{true}, \mathtt{false}\}$ and $p_e \in \mathrm{Rel}_{\mathcal{D}+T}$. The first rule also applies if $k = 0$. $\mathtt{p}_e\!\uparrow\! x$ is obtained from p_e by the substitution of all occurrences of the dot '.' by x, and if x is an object, by the substitution of property name occurrences in $\mathtt{p}_e$ with the values of these properties in x. Note that if x contains several values of some property, there can be several such substitutions. $\exists\ \mathtt{p}_e\!\uparrow\! x :\models \mathtt{p}_e\!\uparrow\! x$ means that there exists a substitution, which makes $\mathtt{p}_e$ true.

Thus, we see that BoxQL is indifferent to the logical mechanisms working in the knowledge base $\mathcal{KB}$, because the procedural semantics of BoxQL uses its checking tool $\models$ as an oracle in a black box. This means that $\models$ can be defined in various ways: as an inference machine for $\mathcal{SHOIN}(D)$, $\mathcal{ALC}$ or whatever we want. In section 6 we consider an experimental implementation, in which BoxQL is used for queries in an object data base, in which $\models$ is implemented as simple check of explicit data. This means that BoxQL can work at any layer of the logic architecture: 'lifting' BoxQL through its layers preserves the compatibility and semantics of the language. In particular this means that, if to be careful enough, the queries that are asked on the lower layer of an object DB, are still valid on the higher logical layers, which subsume this ODB.

Note that the last two $\diamond$-rules show how to handle ordinary relations (=, ¡, ¿ etc.) with sequences as arguments. For instance, `*[income = 300]` selects persons who have the income of 300. Tom has, but he has also incomes of 200 and 400. Thus, we have to check if `{200, 300, 400} = 300`. In such situations $\diamond$-rules check if *there exist* equal members in the both parts. Of course, sometimes such behavior looks tricky, so BoxQL has special built-ins to treat the sequences in different ways.

The computational complexity of BoxQL depends on the complexity of the underlying $\mathcal{KB}$: by adjusting the checking tools of $\mathcal{KB}$ we can find the necessary ratio of the efficiency and expressiveness. We hope that this can make BoxQL useful in various and very different situations.

5 Translation to DL

In this section we investigate the soundness of BoxQL. First, we show that each BoxQL-query can be translated into a formula of $\mathcal{SHOIN}(D)$ (on the other hand, not any $\mathcal{SHOIN}(D)$ formula can be translated into BoxQL).

To translate BoxQL-queries we augment $\mathcal{SHOIN}(D)$ with a concept non-emptiness construct $\exists c$ [2], which holds if the concept c is non-empty. Also to

h/s	$\mathbb{P}$(h/s)	r	$\mathbb{R}$(r)
ϵ (empty)	$\top$	r1 and r2	$\mathbb{R}(\mathtt{r1}) \sqcap \mathbb{R}(\mathtt{r2})$
h/*	$\mathbb{P}(\mathtt{h})$	r1 or r2	$\mathbb{R}(\mathtt{r1}) \sqcup \mathbb{R}(\mathtt{r2})$
h/ns:*	$c_{ns} \sqcap \mathbb{P}(\mathtt{h})$	not r	$\neg\mathbb{R}(\mathtt{r})$
h/c ($\mathtt{c} \in T_C$)	$c \sqcap \mathbb{P}(\mathtt{h})$	h ! [r]	$\forall\mathbb{P}(\mathtt{h}).\mathbb{R}(\mathtt{r})$
h/ro ($\mathtt{ro} \in T^{\mathrm{o}}_R$)	$\exists ro^{-}.\mathbb{P}(\mathtt{h})$	$\mathtt{h} \in \mathcal{P}$	$\exists\mathbb{P}(\mathtt{h})$
h/-ro ($\mathtt{ro} \in T^{\mathrm{o}}_R$)	$\exists ro.\mathbb{P}(\mathtt{h})$	$\mathtt{p}_e \in \mathrm{Rel}_{\mathcal{D}+T}$	$\exists(x_1,\ldots,x_k)p_e$
h/rt ($\mathtt{rt} \in T^{\mathrm{t}}_R$)	$rt^* \sqcap \mathbb{P}(\mathtt{h})$	h = c	$\mathbb{P}(\mathtt{h/c})$
h/id ($\mathtt{id} \in Id$)	$\{id\} \sqcap \mathbb{P}(\mathtt{h})$		
h/($\mathtt{h}_1,\ldots,\mathtt{h}_k$)	$\mathbb{P}(\mathtt{h}) \sqcap (\bigsqcup_{i=1}^{k} \mathbb{P}(\mathtt{h}_i))$		
h[r]	$\mathbb{P}(\mathtt{h}) \sqcap \mathbb{R}(\mathtt{r})$		
h/v ($\mathtt{v} \in \lvert\mathcal{D}\rvert$)	$\{v\} \sqcap \mathbb{P}(\mathtt{h})$		

Fig. 2. The translation operators $\mathbb{P}(\cdot)$ and $\mathbb{R}(\cdot)$

handle queries like */age resulting in sequences of datatype values, we introduce a construct rt^* for each datatype property $\mathtt{rt} \in T^{\mathrm{t}}_R$. For any value $v \in |\mathcal{D}|$, $\models rt^*(v)$ iff $\exists o :\models \mathtt{rt}(o, v)$. The connectives $\sqcap, \sqcup, \neg$ behave on 'datatype' propositions as propositional conjunction, discjunction and negation, respectively. Figure 2 defines operators $\mathbb{P}$ and $\mathbb{R}$, which translate paths and predicates, respectively, into the formulas of $\mathcal{SHOIN}(D)$ augmented with these two constructs.

Let $\mathcal{T}$ be a $\mathcal{SHOIN}(D)$-description of some world, and $\mathcal{KB}_\mathcal{T}$ a knowledge base, in which $\models$ is interpreted as $\mathcal{SHOIN}(D)$-satisfiability in $\mathcal{T}$. Then the following proposition holds:

Proposition 1 (soundness). *For any* $\mathtt{h} \in \mathcal{P}$*, if* $\mathtt{h}\langle|\mathcal{KB}_\mathcal{T}|\rangle \vdash_\star \mathcal{A}$ *in* $\mathcal{KB}_\mathcal{T}$ *then for each* $x \in \mathcal{A}$*,* $\mathbb{P}(\mathtt{h})(x)$ *is satisfiable in* $\mathcal{KB}_\mathcal{T}$*.*

The proof of this proposition is established by induction on the length of a derivation in $\star-$ and $\diamond-$calculi.

6 Implementation and Evaluation

In this section we consider an implementation and evaluation of BoxQL based on an experimental OntoBox module, which we are developing now in Java. In OntoBox, $\models$ is interpreted as an object DB explicit checker. BoxQL is implemented in OntoBox in a naive style based on the $\star$- and $\diamond$-calculi. Also we verified manually some ideas for compilation of BoxQL-queries.

To evaluate the approach, we checked (1) if BoxQL was adequate and reliable for inexperienced developers, and (2) if it could compete with DB management systems on the lower levels of object processing.

To achieve the first goal we had a number of experiments and questionnaires. E.g. we worked with a group of 24 students. The tasks were to develop (after one introductory lecture) reference systems for LaTeX, CSS, HTML, DOM, etc. The students developed ontologies and the corresponding interfaces. 4 advanced

tasks were offered to the best students. 14 students were successful, 5 had minor problems with interfaces, 3 had minor problems with BoxQL, 2 had problems with both, 1 failed to solve his task. The questionary showed that in general BoxQL was considered by students as simple and natural. 18 students think that BoxQL is easier than SQL. We also asked the students to write the same queries in $\mathcal{SHOIN}(D)$ and then compare the two styles. All of the students confirmed that writing in BoxQL had been much easier (and more familiar) for them than writing in $\mathcal{SHOIN}(D)$.

As a benchmark for the second goal we took the NCBI taxonomy database [5], which describes the names of all organisms that are represented in the genetic databases with at least one nucleotide or protein sequence. This taxonomy contains 482960 objects. The taxonomy established in a database has been converted into an ontology in which every name is an object of the class `node`, and the tree structure is represented by the object property `parent`. In the experiments we asked queries of the form $\underbrace{\texttt{parent/.../parent}}_{n}$ searching for the chains of nodes. Concurrently we asked the equivalent SQL-queries in the original database (in MySQL):

```
select * from nodes where parent_id in          \
  (select id from nodes where parent_id in      | n
      ... ... ...                               |
    (select id from nodes))...);                /
```

with indexed columns `id` and `parent_id` of taxonomy nodes. Here are the results for Apple Mac OS X 10.5.6, Java 1.6.0_07 (64-Bit Server VM), and MySQL 5.0.51a (MySQL does not allow nestings for $n > 32$):

$n =$	1	5	10	20	30	40
number of collected chains	482959	471762	301503	135297	30736	280
MySQL 5.0.51a (sec)	5.02	42.34	83.25	130.24	148.56	n/a
BoxQL(naive, sec)	1.56	6.62	11.93	17.08	18.94	19.2
BoxQL(compiled, sec)	0.16	0.99	1.54	2.16	2.36	2.41

What we want to say by this example is that on the lower levels of the logic architecture we can develop the tools, which are quite good for 'simple' but efficient knowledge management (especially in object models), while staying compatible (e.g. via the query language BoxQL) with much more expressive methods and tools of the Semantic Web.

7 Related Work and Conclusion

In this paper a new approach to knowledge and data management is introduced, which is targeted at conventional web developers, and based on (1) a new query language designed in the XPath style and compatible with object oriented models; (2) a fast non-memory based implementation of this language.

There are a lot of works, which are focused on the management of large amounts of simple data in the context of the SW. Many researchers consider incorporating the style of relational DBs as the basic way to efficiently handle simple data within SW applications (e.g. see [6][7] etc.). We are convinced that the SW itself can provide quite reliable tools, and 'hybridization' with DBs can be avoided in many cases. The solutions within the SW could be more elegant, coherent and profitable. Concerning the interactions between the object oriented approach and the SW, paper [8] gives the informal case study of interaction between an OO programming language (represented by Java) and OWL. We develop an OO-style query language based on a strictly formal approach, which represents object oriented means as a sublogic of general DLs. In [9] in order to represent structured objects (which are analogous to finite networks of individuals), DL is augmented with description graphs. The basic difference between the approaches is that our aim is to handle with `BoxQL` the networks of concrete data, whereas in [9] the problem of graph-like object representation is considered on the level of TBoxes. And for this the strength of DL is not sufficient due to the well known tree model property [10].

A lot of query languages have been developed in the context of the SW (see [11]–[18] etc). The basic features, which distinguish `BoxQL` from them is that it is a traversal language based on the object oriented paradigm, in which triple employing is hidden. E.g. in SPARQL [11] the query to find the capitals of all countries in Africa looks as follows:

```
SELECT ?capital ?country
WHERE {
   ?x :cityname ?capital ;
      :isCapitalOf ?y .
   ?y :countryname ?country ;
      :isInContinent :Africa .
}
```

In `BoxQL` we have:

```
*[cityname as capital]/
  isCapitalOf[countryname as country and isInContinent = &Africa]
```

The difference is clear. In SPARQL we have to divide the query into a number of triples with auxiliary variables. In `BoxQL` we just determine a two-step walk along the knowledge graph.

The approach considered in this paper raises a number of questions. Is it possible to use logic architectures and `BoxQL` for developing SW-technologies, which can compete (and cooperate with) the standard methods of data management (like DBs) on lower levels, but enjoy on the higher levels the full strength of logic? Will such technologies be really interesting to a wider range of users? Shall we manage to preserve compatibility between the lower and higher layers of logical architectures in procedural data management environments? There are no answers yet, but the initial steps look promising.

References

1. Malykh, A., Mantsivoda, A., Ulyanov, V.: Logic Architectures and the Object Oriented Approach. Technical Report (2009)
2. Horrocks, I., Patel-Schneider, P.F.: Reducing OWL entailment to description logic satisfiability. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 17–29. Springer, Heidelberg (2003)
3. Horrocks, I., Hayes, P., Patel-Schneider, P.F.: OWL Web Ontology Language Semantics and Abstract Syntax, `http://www.w3.org/TR/owl-semantics/`
4. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
5. The NCBI Entrez Taxonomy, `http://www.ncbi.nlm.nih.gov/sites/entrez?db=taxonomy`
6. Hustadt, U., Motik, B., Sattler, U.: Reasoning in Description Logics by a Reduction to Disjunctive Datalog. Journal of Automated Reasoning 39(3), 351–384 (2007)
7. Haarslev, V., Möller, R.: On the scalability of description logic instance retrieval. Journal of Automated Reasoning 41(2), 99–142 (2008)
8. Puleston, C., Parsia, B., Cunningham, J., Rector, A.L.: Integrating Object-Oriented and Ontological Representations. A Case Study in Java and OWL, pp. 130–145
9. Motik, B., Grau, B.C., Horrocks, I., Sattler, U.: Representing Structured Objects using Description Graphs. In: KR 2008, pp. 296–306 (2008)
10. Vardi, M.Y.: Why Is Modal Logic So Robustly Decidable? In: Proc. DIMACS Workshop. DIMACS Series, vol. 31, pp. 149–184.
11. Prudhommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Recommendation (2008)
12. Seaborne, A.: RDQL - A Query Language for RDF. W3C Member Submission (2004)
13. Ortiz, M., Calvanese, D., Eiter, T.: Data Complexity of Query Answering in Expressive Description Logics via Tableaux. Journal of Automated Reasoning 41, 61–98 (2008)
14. Bry, F., Furche, T., Linse, B.: Data Model and Query Constructs for Versatile Web Query Languages: State-of-the-Art and Challenges for Xcerpt. In: Alferes, J.J., Bailey, J., May, W., Schwertel, U. (eds.) PPSWR 2006. LNCS, vol. 4187, pp. 90–104. Springer, Heidelberg (2006)
15. Kaplunova, A., Möller, R.: DIG 2.0 Concrete Domain Interface Proposal., `http://www.sts.tu-harburg.de/~al.kaplunova/dig-cd-interface.html`
16. Frasincar, F., Houben, G.-J., Vdovjak, R., Barna, P.: RAL: An Algebra for Querying RDF. World Wide Web: Internet and Web Information Systems 7, 83–109 (2004)
17. Noy, N., Musen, M.A.: Specifying ontology views by traversal. In: McIlraith, S.A., Plexousakis, D., van Harmelen, F. (eds.) ISWC 2004. LNCS, vol. 3298, pp. 713–725. Springer, Heidelberg (2004)
18. Ogbuji, C.: Versa: Path-Based RDF Query Language, `http://www.xml.com/pub/a/2005/07/20/versa.html?page=1`