Research Report

Portable Ontology Query Language (POQL)

Tudor Muresan, Rodica Potolea, Alin Suciu,
Emilia Cimpian, Adrian Mocan, Radu Popovici, Horatiu Tarcea

Computer Science Department of Technical University of Cluj-Napoca, Romania,
Multiparadigm Logic Programming Group http://bavaria.utcluj.ro/~suciu/mlp/

{tmuresan, potolea, suciu}@cs.utcluj.ro
{cemilia, madrian, pandrei, tiustin}@asterix.obs.utcluj.ro

Research supported by
DaimlerChrysler AG grant no. 3969000931-F92

July 25, 2002

Table of Contents

Abstract

This research report presents the definition and implementation of a query language for reusable knowledge bases, which uses the Prolog logical form. The advantage is, along with the complexity and flexibility of the allowed questions, the fact that it constitutes a theoretical interface with user friendly querying systems (i.e. Natural Language Interface (NLI)). Also, it makes use of the Prolog solving mechanism for an extensive search in the solution space, providing the framework for the development of theories for automated merging and alignment of existing ontologies. The goal of this research is achieved by the current implementation of the POQL Kernel.

Keywords:

logical query language, computational model, knowledge base space searching, portable ontology, implementation

1 Introduction

A large number of ontologies have been constructed taking into account the principle of generating reusable knowledge bases by adopting standard representational languages [2], [9] or by achieving portability through a translational approach [8]. The advantage of easy knowledge acquisition [1], [4] of the existing tools becomes a weakness from the querying point of view. Thus, the development of query tools independent of the ontology representation becomes appropriate [16], [17]. Such query tools serve both for the development of user friendly query interfaces (i.e. Natural Language Interfaces) and for the purpose of merging and alignment of the existing ontologies. Furthermore, currently [7], there are yet extremely few theories or methods which facilitate or automate the process of reconciling disparate ontologies.

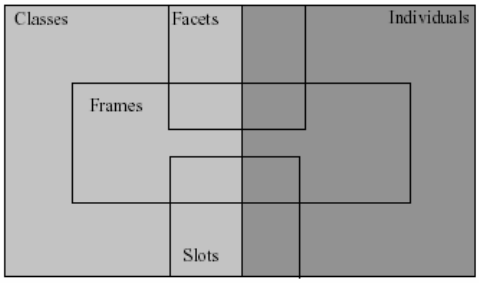
This research report presents the definition and implementation of a query language for reusable knowledge bases, which uses the Prolog logical form. The advantage is, along with the complexity and flexibility of the allowed questions, the fact that it constitutes a theoretical interface with user friendly querying systems (i.e. NLI). Also, it makes use of the Prolog solving mechanism, for generating all the solutions of a specific search, providing the framework for the development of theories for automated merging and alignment of existing ontologies.

In section 2 we give an overview of the theoretical concepts pertaining to a reusable ontology frame which conforms to the OKBC model [6] and also constitutes a description of the syntax, semantics, architecture and computational model of **POQL**. Implementation issues are detailed in the 3rd section. Section 4 gives brief guidelines regarding installation and utilization issues. Experimental results are shown in section 5. We conclude by presenting the conclusions and proposals for further development, in section 6.

2 Theoretical Aspects

2.1 Open Knowledge Base Connectivity (OKBC) Ontology Frame

An ontology is a specification of a representational vocabulary for a shared domain of discourse. *"The OKBC knowledge model defines that classes and individuals form disjoint partitions of a KB. It does not commit to whether classes, individuals, slots and facets are represented as frames. It also does not commit to whether slots and facets should be represented as classes or individuals*.



*OKBC defines an **Assertion Language** (**AL**) for declarative specification of knowledge. The **AL** is a first-order language with conjunction and predicate symbols, **but without disjunction, explicit quantifiers, function symbols, negation, or equality**. The predicate symbols of the OKBC **AL** are class, individual, primitive, instance-of, type-of, subclass-of, slot-of, facet-of, template-slot-of, template-facet-of, own-slot-value, own-facet--value, template-slot-value, and template-facet—value"* [6].

Our research report takes into consideration the most recent version of the Protégé knowledge representation system (KRS), Protégé 2000, which incorporates the OKBC knowledge model. It restricts OKBC to a hierarchy of frames [5], [9], [14].

The standard frame **ontology** consists of a hierarchy of **frames** which are organized, according to their role, into three main categories: classes, slots and facets.

- **Classes** are concepts in the domain of discourse, collections of objects that have similar properties; they are arranged into a subclass-superclass hierarchy and allow multiple inheritance. There are two subcategories for classes: metaclasses - classes which have as instances other classes, and ordinary classes – which have ordinary objects as their materialization.
- **Slots** are named binary relations between a class and either another class or a primitive object in order to describe properties, attributes of classes or relations between classes. Slots attached to a class may be further constrained by facets.
- **Facets** are named ternary relations between a class, a slot, and either another class or a primitive object; they describe properties of slots and may impose additional constraints on a slot attached to a class.
- **Instances** are materializations of classes.

A **knowledge base** includes both the ontology and individual instances of classes with specific values for their slots. The distinction between classes and instances is not an absolute one due to the existence of metaclasses.

The above mentioned restriction of Protégé to a hierarchy of frames has allowed us to further extend semantic of the query language **POQL** as compared to the OKBC Assertion Language, thus eliminating AL's previously mentioned limitations (lack of disjunction, negation, relational operators and so on, see examples in section 5.2).

Meanwhile, limits of traditional OQL and SQL are surpassed in a functionally similar fashion to LOREL, a query language developed by TSIMMIS [2] (see example in section 5.3).

2.2 Query Syntax

The syntax of **POQL** is comprehensive in relation with the specifications of the ongoing project, as stated in the definitions of *Query Language for KRML*, [12]. The extended syntax refers to the constraints' specifications.

A query consists of one or many linked atomic queries. The syntax of such connections follows the Prolog logical form, with conjunctions and disjunctions between expressions and negation.

```
<query> ::= <disjunct-query>

<disjunct-query> ::=
      <conjunct-query> |
      <conjunct-query> <disjunct-op> <disjunct-query>

<conjunct-query> ::= <atomic query> |
      <atomic query> <conjunct-op> <conjunct-query>

<atomic query> ::=
      <term> <poql-op> <term> | <path term> |
      <path term> <relational-op> <path term> |
      <axiom-predicate> |
      '(' <query> ')' | 'not(' <query> ')'

<path term> ::=
      <term> | <term> '.' <path term>

<term> ::=
      '<frame name>' | '?' <Prolog variable> | <Prolog constant>

<poql-op> ::=
      'isa' | 'sub' | ':' | '::'

<relational-op> ::=
       '=' | '>=' | '=<' | '<' | '>' | '\=='
```

```
<logical-op> ::=
      <conjunct-op> | <disjunct-op>

<conjunct-op> ::= ‘,’

<disjunct-op> ::= ‘;’
```

The <**poql**-op> operators correspond to the relations between frames:
isa direct instance-class relation;
: transitive closure of *isa* relation;
sub direct inheritance relation;
:: transitive closure of *sub* relation.

The name of a frame may be simple, referring directly a frame of the knowledge base, or it may be a path. A path is a concatenation of slots s1, s2, ..., sn, written o.s1.s2....sn, where o is the frame slot s1 belongs to (class or instance), o.s1 refers the frame slot s2 belongs to and so forth. Such expression has itself a truth value given by the (non)existence of the path.

2.3 Computational Model

In contrast to [12], the queries' semantic is formally specified by a meta-interpreter [3] for Prolog with freeze [15]. The soundness and safety of negation as failure for this computational model is proven below as in [3].

A SLD refutation procedure for a logic program P and a goal G uses a computation rule and a search strategy (rule). The computation rule chooses a subgoal from the sequence of goals to perform the derivation step. A SLD derivation is (said to be) *fair* if it ensures any subgoal selection in a finite number of steps (the depth first search strategy of Prolog is unfair).

The soundness and completeness of a fair SLD refutation has been proved, that is the equivalence between the logical consequence ($P \models G$) and SLD refutation ($P \vdash G$).

$$(P \models G) \leftrightarrow (P \vdash G)$$

If the negation as failure is taken into consideration, a SLDNF computation rule is said to be *safe* if it selects only ground negative literals and it does not interrupt the corresponding SLDNF finite failure subtree building.

If comp(P) is the Clark completion of a program P, and the SLDNF rule of computation is safe, the soundness and completeness of SLDNF refutation hold true.

$$(comp(P) \models G) \leftrightarrow (P \vdash G)$$

Moreover, the soundness and completeness of a SLD refutation are independent of the chosen computation rule (e.g. the current subgoal selection).

We consider P_f the logic program obtained from P, by enclosing any subgoal G_i of a clause into a `freeze(Var,`G_i`)` predicate, where `Var` belongs to the set of G_i variable, `Var` $\in$`SetVar(`G_i`)`.

If

$$H{:-}\ B_1,..,B_i\ ,..,B_n \in P,$$

then

$$H{:-}B_1,..,\texttt{freeze(Var, }B_i\texttt{)},..,B_n \in P_f\ .$$

A subgoal `freeze(Var,` B_i`)` is not selected as long as `Var` is unbound. On its selection the equivalence

$$\texttt{freeze(Var, }B_i\texttt{)} \leftrightarrow B_i$$

takes place.

A fair SLD refutation for a program P_f and a goal G is achieved if the empty clause may be derived in a finite number of steps. This means that all the subgoals `freeze(Var,` B_i`)` have actually been selected. Taking into consideration the independence of the choice of the computation rule and the logic equivalence between `freeze(Var,` B_i`)` and B_i in the moment of the selection, we have:

Lemma :

$$(P_f \vdash G) \rightarrow (P \vdash G)$$

and

Corollary: (Soundness of SLD refutation for programs with freeze.)

$$(P_f \vdash G) \rightarrow (P \models G)$$

The soundness of P_f programs makes possible the use of Prolog with freeze as target language for queries interpretation.

Even if the Prolog strategy is an unfair and incomplete one, freeze does not introduce new exceptions from the theoretical model (comparing with those of standard Prolog). Moreover, freeze may improve the Prolog program's behavior, making safe the negative literals selection (safety of SLDNF refutation). However, the Prolog strategy makes incompatible the use of freeze together with the cut ('!'), without imposing special restrictions.

2.4 Query Semantic

The queries' semantic is formally specified by a meta-interpreter [3] for Prolog with freeze, in a compositional manner [11]. For the `<logical-op>` we define:

`semantic(Q1 <l-op> Q2)` $\rightarrow$ `sem_freeze(Q1) <l-op> sem_freeze(Q2)`

where

`sem_freeze(Q)` $\rightarrow$ `semantic(Q), sem_queue.`

For the atomic query Q_A with `<poql-op>` we define:

`semantic(`Q_A`)` $\rightarrow$ `postpone(`Q_A`)`

where

`postpone(X <poql-op> Y)` $\rightarrow$ `freeze((X,Y), X <poql-op> Y)`

The above definition entails that atomic queries are postponed through the freeze predicate until at least one of its variables becomes instantiated. Postponed atomic queries are resumed by the `sem_queue` predicate.

The following equivalences hold true:

```
a <poql-op> B ↔ forall(X, a <poql-op> X , Lx),
                  member(B, Lx)
```

`forall(X,a <poql-op> X,Lx)` $\leftrightarrow$ `api(<poql-op>)(a,Lx)`

where `api(<poql-op>)(a,Lx)`represents an API call specific to the ontology representation. These equivalences allow the definition of the semantic for the postponed atomic queries through the correspondent ontology program interface:

`semantic(a <poql-op> B )` $\rightarrow$ `api(<poql-op>)(a,Lx),member(B, Lx)`

This renders the Prolog search strategy independent of the actual representation of the queried ontology (Fig. 1).

We shall refer as *acceptable (defined) queries* to all queries which are not *indefinitely postponed*. For such queries, we have demonstrated soundness in section 2.3. Many of the indefinitely postponed queries may be transformed to be acceptable by enriching the specification of the query (see section 5.2).

2.5 Portable Ontology Query Language Architecture

The architecture of **POQL** is shown in Fig.1.

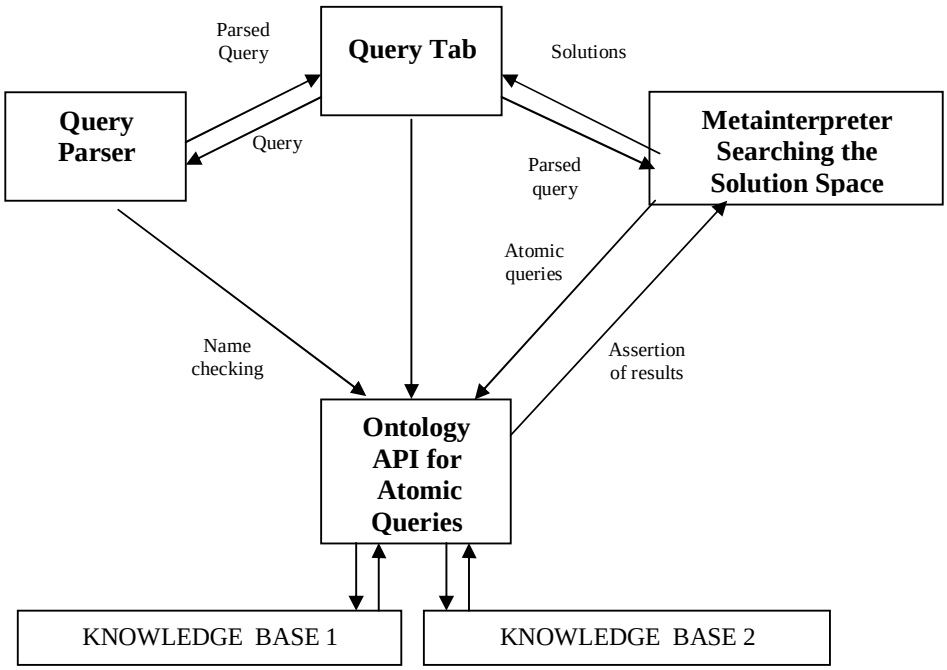


Fig. 1 System Architecture

A query is entered in a Prolog like logical form, using the interface we have developed. Subsequently, a parser performs syntax and name checking, converting the query to a string of our convenience which is further passed to the metainterpreter of Prolog with freeze. The parser ensures, among other, the correct order of execution for the atoms of complex queries. We have used a Prolog like strategy of searching through the entire solution space, thus obtaining all the solutions for our query.

The resolution of the atomic queries is handled by methods specific to the ontology representation (API). The result of each such atomic query is asserted as a Prolog fact and further used by the solving algorithm.

The current implementation of **POQL** ensures queries' independence from the representation of the queried ontologies. Furthermore, the system is subject to future developments, so that it may simultaneously query two distinct ontologies with different representations. This feature will eventually make possible the integration of an ontology merging theory [10]. Meanwhile, the user interaction may be enriched with a Natural Language Interface.

3 Implementation Issues of the POQL Kernel

The implementation is focused on several distinct modules: an interface integrated in *Protégé 2000* or *OntoWorks* [13], a syntactic parser which enforces correctness of syntax and use of names pertaining to the ontology name space, an API which provides access to methods querying the structure of the ontology and a metainterpreter which gives the solving strategy.

For the implementation, we have used two programming languages: Java [20] and XSB Prolog [18]. The reason for using Java is that Protégé is a Java based environment and provides an API for easy access to both the representation of the ontology and interface development.

The syntactic parser is written under JavaCC, a Sun CompilerCompiler [19].

The implementation of the metainterpreter is written in XSB Prolog, which gives a direct mapping between the logical form of the query and the solving strategy. XSB Prolog represents a powerful instrument for solving logical forms; also, it uses a backtracking mechanism that enables us to easily search through the entire solution space.

All communication between Prolog and Java is performed, back and forth, through the Interprolog [18], an easy to use interface which converts Java objects to strings in DCG format and vice versa. This enables Prolog code to call Java methods and Java code to solve Prolog queries.

A query is entered in a Prolog like logical form, using the interface we have developed. Subsequently, a parser performs syntax and name checking, converting the query to a string of our convenience which is further passed to the XSB Prolog resolution mechanism (Fig. 1). Each query is interpreted by a Prolog metainterpreter [3], which ensures, among other, the correct order of execution for the atoms of complex queries. We have used a Prolog like strategy of searching through the entire solution space, thus obtaining all the solutions for the query.

3.1 Protégé integrated interface

Protégé allows easy development through its structure of *Tabs*. Once such a Tab is created, it can be included in the Protégé environment by modifying the `manifest.mf` file.

POQLTab presents a multitude of functionalities:

- A *class hierarchy* panel which gives full browsing control to the classes' taxonomy. Double-clicking on a class reveals/hides its direct subclasses.
- A *direct instances* panel which displays the browser texts for the direct instances of the selected class from the hierarchy panel. Double-clicking on

an instance adds the name of the instance to the query. The *instance name* is not necessarily similar to the *instance browser text*. The former identifies uniquely the frame in the hierarchy of frames, whereas the latter gives a description of the frame, not necessarily unique in the name space.

- A *slots* panel which displays own slots of the selected class or instance. Double-clicking on a slot inserts its name at the current cursor position in the query.
- A text field where the user enters the desired query and an attached button which starts the processing.
- All results of a query (bindings of defined variables) are displayed as a table, below the query text field.

When **POQL**Tab is loaded, method `initialize` is called: it sets the working knowledge base, loads the Prolog engine (if it has not already been loaded), loads the Prolog modules (metainterpreter and resolution clauses), recreates the interface and initializes the installation path and the path to the *plugins* directory.

`runQuery` starts the actual processing. It creates a parser `QP` for the current query, calls `runParser` and creates the Prolog query from the parsed expression and appends to it the list of variables which are of interest to the user (the defined variable – see the next section). Then it calls the Prolog metainterpreter with the newly generated query and further collects all bindings of the defined variables and displays them in a table like format.

The interface also displays the truth value of the query and the execution time.

3.2 Syntactic parser

The syntactic parser performs syntax checking which ensure that the current query conforms to the rules described in section 2.2. Furthermore, the parser checks whether the names which refer to frames of the ontology actually belong to the current name space. Such names are entered between pairs of the special character \'.

The parser also accepts constant values, such as numbers, strings and the empty set (i.e. []). All constants are entered between pairs of the special character \".

In order to ensure correctness of the execution, it is necessary that frame names do not comprise characters other than alphanumeric and blanks.

A variable which appears in a query may have two distinct forms, established by its prefix: *?* or *?_*. The former is a *defined variable* – a variable which bindings must appear in the final result, whereas the latter is an *undefined variable* – one which has only a linking role between different parts of the query and the user has no interest in its meaning but uses it as a liaison (leant).

There are several steps involved in the processing performed by the parser on the initial query: a new string, `parsedExpression,` is generated from the initial one, replacing

all occurrences of the character \. with \@, due to the way Prolog interprets it. Also values of constants are replaced with their Prolog correspondent, depending whether they are numbers (integer or float), strings, boolean or the empty set.

At the same time, a list of defined variables, `variableList,` is created. It may be accessed by calling the `getVariableList` or `getVariableAssertionList` methods, which return, respectively, an `ArrayList` and a `String`. The list will be appended to the end of the query, in an assertion clause. All the bindings for this list of variables are collected by Prolog, after the resolution of the query has completed. They are then sent from Prolog to Java, as a string, and further interpreted by the **POQL** interface.

All processing takes place for a uniquely generated `StringBufferInputStream`, which is created from a `String` passed to the constructor of the class. In order to process the input stream, the user must call the `runParser` method, which sets the working knowledgebase for name checking, initializes `parsedExpression` and `variableList` and starts the actual parsing by calling the `one_line` method.

As stated in the previous section, the interface is responsible for supervising of the entire process; after the parsed query is generated, `parsedExpression,` it is returned to the **POQL**Tab which further passes it (in a specific logical form) to the metainterpreter of Prolog with freeze for its resolution.

3.3 Prolog module (metainterpreter)

The Prolog module comprises a metainterpreter for Prolog with freeze and the respective Java **POQL**_API calls for the resolution of the atomic queries.

The metainterpreter for Prolog with freeze, extending the one in [3], is given in DCG form, *in extenso,* in Appendix A. It preserves the traditional semantic of the computed answer of the general Prolog queries.

At the same time, the subgoal selection rule is modified, as follows: the leftmost subgoal having all its input arguments bound is selected first for computation. It follows that subgoals with unbound input arguments are postponed (see section 2.4.).

There are three types of postponed subgoals:

- An atomic query: `?X <poql-op> ?Y` with both arguments unbound
- A path term: `?X '.' <path term>` with ?X unbound
- A negated subgoal with unbound inner variables (thus, the safety of negation is ensured)

The compound queries' semantic is defined in a compositional manner [11]: the semantic of the whole is obtained from the semantic of its parts. This is ensured by each DCG metainterpreter rule.

All <**poql**-op> appearing in postponed atomic queries, are mapped by the `rename` predicate to corresponding operators which are directly executed by the Prolog engine on resuming by the `sem_queue` predicate.

Initial Operator	Renamed Operator
`isa`	`isaa`
`:`	`:^`
`::`	`::^`
`sub`	`subb`
`not`	`nott`
`dot3`	`dott3`
`@`	`sdot`

Table. Operators mapping

The `api/3` and `result/2` predicates are used for the Java **POQL** API calls (see Appendix A). The `api/3` predicate calls Java API methods through the Interprolog Interface. The results returned by the Java API are asserted in the Prolog database as facts of the `result/2` predicate.

3.4 Prolog/Java POQL API calls

POQL_API is an abstract class, which ensures a common interface for accessing the current knowledge base. It delegates the responsibility of implementing its methods to the classes which inherit it (in our case Protégé_**POQL**_API and OW_ **POQL**_API – see section 3.4.1.).

POQL atomic query	**Prolog POQL API**	**Java POQL API**
?A **sub** ?B	Postponed	Postponed
?A **sub** 'aClass'	getDirectSubclasses(A, aClass)	getDirectSubclasses(aClass)
'aClass' **sub** ?B	getDirectSuperclasses(aClass, B)	getDirectSuperclasses(aClass)
'aClass' **sub** 'bClass'	directSubClassOf(A,B)	directSubclassOf(aClass,bClass)
?A **sub** 'aSlot'	getDirectSubslots(A, aSlot)	getDirectSubslots(aSlot)
'aSlot' **sub** ?B	getDirectSuperslots(aSlot, B)	getDirectSuperslots(aSlot)
'aSlot' **sub** 'bSlot'	directSubSlotOf(aSlot,bSlot)	directSubslotOf(aSlot, bSlot)
?A :: ?B	Postponed	Postponed
?A :: 'aClass'	getSubclasses(A, aClass)	getSubclasses(aClass)
'aClass' :: ?B	getSuperclasses(aClass, B)	getSuperclasses(aClass)
'aClass' :: 'bClass'	subClassOf(A,B)	subclassOf(aClass,bClass)
?A :: 'aSlot'	getSubslots(A, aSlot)	getSubslots(aSlot)
'aSlot' :: ?B	getSuperslots(aSlot, B)	getSuperslots(aSlot)
'aSlot' :: 'bSlot'	subSlotOf(aSlot,bSlot)	subslotOf(aSlot, bSlot)
?A **isa** ?B	Postponed	Postponed
?A **isa** 'aClass'	getDirectInstances(A,aClass)	getDirectInstances(aClass)
'anInastance' **isa** ?A	getDirectTypeOf(anInstance, B)	getDirectTypeOf(anInstance)
'anInstance' **isa** 'aClass'	directInstance(anInstance,aClass)	directInstanceOf(anInstance,aClass)
?A : ?B	Postponed	Postponed
?A : 'aClass'	getInstances(A,aClass)	getInstances(aClass)
'anInastance' : ?A	getTypeOf(anInstance, B)	getTypeOf(anInstance)
'anInstance' : 'aClass'	instance(anInstance,aClass)	instanceOf(anInstance,aClass)
?A . ?B	Postponed	Postponed
?A . 'aSlot'	Postponed	Postponed
'anInstance' . 'aSlot'	getSlotAtInstanceValue(anInstance,aSlot)	getSlotAtInstanceValue(anInstance,aSlot)
anInstance . ?A	getSlotsAtInstance(anInstance)	getSlotsAtInstance(anInstance)

POQL_API's methods need to be static because of the communication way between Java and Prolog, which is achieved through Interprolog.

This class contains two types of methods: the first kind is used for initialization and the latter is used by Prolog for its solving strategy.

- Initialization methods – they initialize internal attributes, such as the working knowledge base, the Prolog engine and the response list. Also, there are methods which are used to assert in Prolog the return values for methods of the latter kind.
- Actual interface methods: there are two distinct functionalities for methods pertaining to the **POQL**_API:

1. Methods used for access to the ontology – they reflect the relations existing between frames: PART-OF, IS-A and their respective transitive closure. These methods verify the existence of such relations (`isClass, isSlot, directInstanceOf, subclassOf, instanceOf` etc.) or return objects for which such relations apply (`getSubclasses, getDirectInstances, getTypeOf, getSlotsAtInstance` etc.)
2. Methods used to access the values of instances of classes (objects, slots) (`getSlotAtInstanceValue`).

   All these methods have as return value objects of type `Result,` which encapsulate both a truth value and a list of objects.

The above table represents the translation of **POQL** atomic queries to Java calls. The full methods' prototypes are given in Appendix B.

3.5 Porting POQL_API on specific platforms

At the present time, there are two distinct implementations of **POQL**_API, the Protégé_**POQL**_API and OW_ **POQL**_API classes. They use different approaches to gain access to the knowledge base. Thus, the user can choose the mode of interrogation for a working knowledge base: using Protégé API, or, respectively, using OW_API. The former implementation is in final form, whereas the latter is subject to further developments of the OntoWorks API Model. Due to the fact that the current OntoWorks API is not in final form, the latter implementation has several unlisted features, which belong in fact to the API itself. Therefore, this implementation of **POQL**_API mixes methods from both the OntoWorks and Protégé APIs.

Java POQL API	**Protégé_POQL_API** Protégé API calls	**OntoWorks_POQL_API** Protégé API calls	 OntoWorksAPI calls
isClass(String cls)	**getCls** from *KnowledgeBase*	-	**getClass** from *OW_KnowledgeBaseManager*
isSlot(String slot)	**getSlot** from *KnowledgeBase*	**getSlot** from *KnowledgeBase*	**getProtegeObject** from *OW_KnowledgeBaseManager*
isMetaclass(String metaCls)	**getCls** from *KnowledgeBase* **isMetaCls** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **isMetaClass** from *OWI_Class*
getDirectInstances(String cls)	**getCls** from *KnowledgeBase* **getDirectInstances** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getDirectInstances** from *OWI_CollectionOfClasses*
getInstances(String cls)	**getCls** from *KnowledgeBase* **getInstances** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getInstances** from *OWI_CollectionOfClasses*
subclassOf(String son, String parent)	**getCls** from *KnowledgeBase* **hasSuperclass** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getSubclasses** from *OWI_Class*
directSubclassOf(String son, String parent)	**getCls** from *KnowledgeBase* **hasDirectSuperclass** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getDirectSubclasses** from *OWI_Class*
superclassOf(String parent, String son)	**getCls** from *KnowledgeBase* **hasSuperclass** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getDirectSuperclasses** from *OWI_Class*

Java POQL API	**Protégé_POQL_API** Protégé API calls	**OntoWorks_POQL_API** Protégé API calls	OntoWorksAPI calls
directSuperclassOf(String parent, String son)	**getCls** from *KnowledgeBase* **hasDirectSuperclass** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getDirectSuperclasses** from *OWI_Class*
directSuperslotOf(String parent, String son)	**getSlot** from *KnowledgeBase* **getDirectSuperslots** from *Slot*	**getSlot** from *KnowledgeBase* **getDirectSuperslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*
directSubslotOf(String son, String parent)	**getSlot** from *KnowledgeBase* **getDirectSuperslots** from *Slot*	**getSlot** from *KnowledgeBase* **getDirectSuperslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*
superslotOf(String parent, String son)	**getSlot** from *KnowledgeBase* **getSuperslots** from *Slot*	**getSlot** from *KnowledgeBase* **getSuperslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*
subslotOf(String son, String parent)	**getSlot** from *KnowledgeBase* **getSuperslots** from *Slot*	**getSlot** from *KnowledgeBase* **getSuperslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*
directInstanceOf(String inst, String cls)	**getInstance** from *KnowledgeBase* **getCls** from *KnowledgeBase* **hasDirectType** from *Instance*	-	**getProtegeObject** from *OW_KnowledgeBaseManager* **getDirectInstances** from *OWI_Class* **getName** from *OW_Class*

Java POQL API	**Protégé_POQL_API** Protégé API calls	**OntoWorks_POQL_API** Protégé API calls	OntoWorksAPI calls
instanceOf(String inst, String cls)	**getInstance** from *KnowledgeBase* **getCls** from *KnowledgeBase* **hasType** from *Instance*	-	**getProtegeObject** from *OW_KnowledgeBaseManager* **getInstances** from *OWI_Class* **getName** from *OW_Class*
getDirectTypeOf(String inst)	**getInstance** from *KnowledgeBase* **getDirectType** from *Instance* **getName** from *Cls*	**getInstance** from *KnowledgeBase* **getDirectType** from *Instance* **getName** from *Cls*	**getProtegeObject** from *OW_KnowledgeBaseManager* **getName** from *OW_Class*
getTypeOf(String inst)	**getInstance** from *KnowledgeBase* **getDirectType** from *Instance* **getSuperclasses** from *Cls*	**getInstance** from *KnowledgeBase* **getDirectType** from *Instance*	**getProtegeObject** from *OW_KnowledgeBaseManager* **getSuperclasses** from *OW_Class*
getSuperslots(String slot)	**getSlot** from *KnowledgeBase* **getSuperslots** from *Slot*	**getSlot** from *KnowledgeBase* **getSuperslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*
getDirectSuperslots(String slot)	**getSlot** from *KnowledgeBase* **getDirectSuperslots** from *Slot*	**getSlot** from *KnowledgeBase* **getDirectSuperslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*
getDirectSubslots(String slot)	**getSlot** from *KnowledgeBase* **getDirectSubslots** from *Slot*	**getSlot** from *KnowledgeBase* **getDirectSubslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*

Java POQL API	**Protégé_POQL_API** Protégé API calls	**OntoWorks_POQL_API** Protégé API calls	 OntoWorksAPI calls
getSubslots(String slot)	**getSlot** from *KnowledgeBase* **getSubslots** from *Slot*	**getSlot** from *KnowledgeBase* **getSubslots** from *Slot*	**getProtegeObject** from *OW_KnowledgeBaseManager*
getSuperclasses(String cls)	**getCls** from *KnowledgeBase* **getSuperclasses** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getSuperclasses** from *OW_Class*
getDirectSuperclasses(String cls)	**getCls** from *KnowledgeBase* **getDirectSuperclasses** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getDirectSuperclasses** from *OW_Class*
getSubclasses(String cls)	**getCls** from *KnowledgeBase* **getSubclasses** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getSubclasses** from *OW_Class*
getDirectSubclasses(String cls)	**getCls** from *KnowledgeBase* **getDirectSubclasses** from *Cls*	-	**getClass** from *OW_KnowledgeBaseManager* **getDirectSubclasses** from *OW_Class*

Java POQL API	**Protégé_POQL_API** Protégé API calls	**OntoWorks_POQL_API** Protégé API calls	 OntoWorksAPI calls
getSlotAtInstanceValue(String instS, String slotS)	**getInstance** from *KnowledgeBase* **getSlot** from *KnowledgeBase* **getValueType** from *Slot* **getOwnSlots** from *Instance* **getOwnSlotValues** from *Instance*	**getInstance** from *KnowledgeBase* **getSlot** from *KnowledgeBase* **getValueType** from *Slot* **getOwnSlots** from *Instance* **getOwnSlotValues** from *Instance*	**getProtegeObject** from O*W_KnowledgeBaseManager*
getSlotsAtInstance(String instS)	**getInstance** from *KnowledgeBase* **getOwnSlots** from *Instance*	**getInstance** from *KnowledgeBase* **getOwnSlots** from *Instance*	**getProtegeObject** from O*W_KnowledgeBaseManager*

4 Installation and Utilization Tips

4.1 Installation Guidelines

In order to have a working version of POQL, one must take the following steps:

- Make sure that *OntoWorks* or *Protégé 2000* is installed to a path which contains **no** blank spaces (XSB Prolog is designed for Linux, it won't accept any path that doesn't conform to a standard Unix path).

- Unpack the self-extracting file to the OntoWorks installation directory.
  Make sure that all necessary files are present:
  - User files – in the *plugins* directory:
    - `poql_qsem.P, poql_operators.P, poql_map.P, poql_api_calls.P;`
  - **POQL**`Tab.jar` - in the *plugins* directory:
  - Query history ontology related files – in the *plugins* directory:
    - `*.demo;`
  - XSB files: all XSB release files - in the OntoWorks installation directory.
  - Documentation files – in the OntoWorks directory: `Portable_Ontology_Query_Language.pdf, Portable_Ontology_Query_Language__Research_Report.bat`

- Copy the XSB release file structure to the installation directory of *Protégé 2000* or *Ontoworks* (…*OntoWorks**XSB*). Make sure that, no matter what version of XSB is used, it is installed in a directory named XSB (not XSB_2_5 or others). Set the PATH environment variable to point to the XSB executable binary file (e.g. …*OntoWorks**xsb**config**x86-pc-windows**bin*).

- Start *OntoWorks* or *Protégé 2000* and in the *File->Configure* menu check the box corresponding to the **POQL**Tab.

All necessary Java files (classes and sources) are packed into the `POQLTab.jar` file . Its structure is the following one:

- User interface: com.utcn.poql.ui.
  - `Choose_POQL_API_Dialog.java`
  - `InstanceClsesPanel.java`
  - `InstanceRenderer.java`
  - `InstancesPanel.java`
  - **POQL**`Tab.java`
  - `ResponseTable.java`
  - `RootDirectory.java`
  - `SlotRenderer.java`
  - `SlotsPanel.java`
- Ontology Access: com.utcn.poql.ontologyAccess.

- **POQL**_API.java
- Result.java

com.utcn.poql.ontologyAccess.protégé.

- Protégé_**POQL**_API.java

com.utcn.poql.ontologyAccess.ontoWorks.

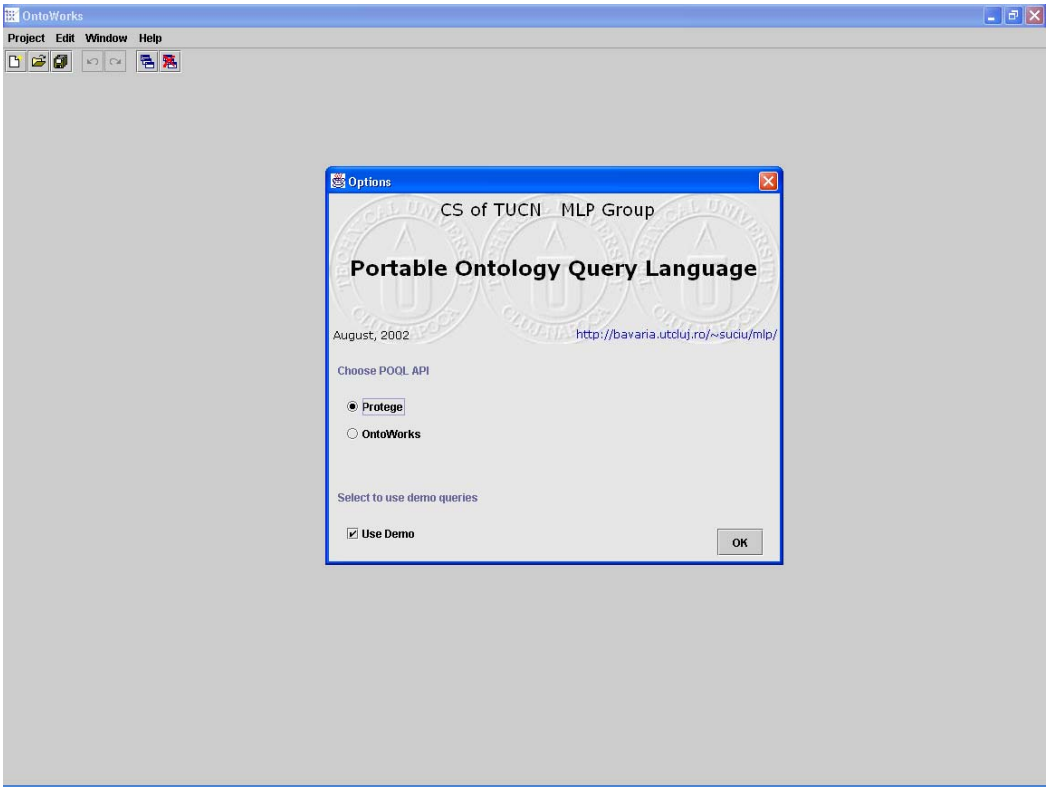
- OntoWorks_**POQL**_API.java

o Syntactic Parser: com.utcn.poql.syntacticParser.

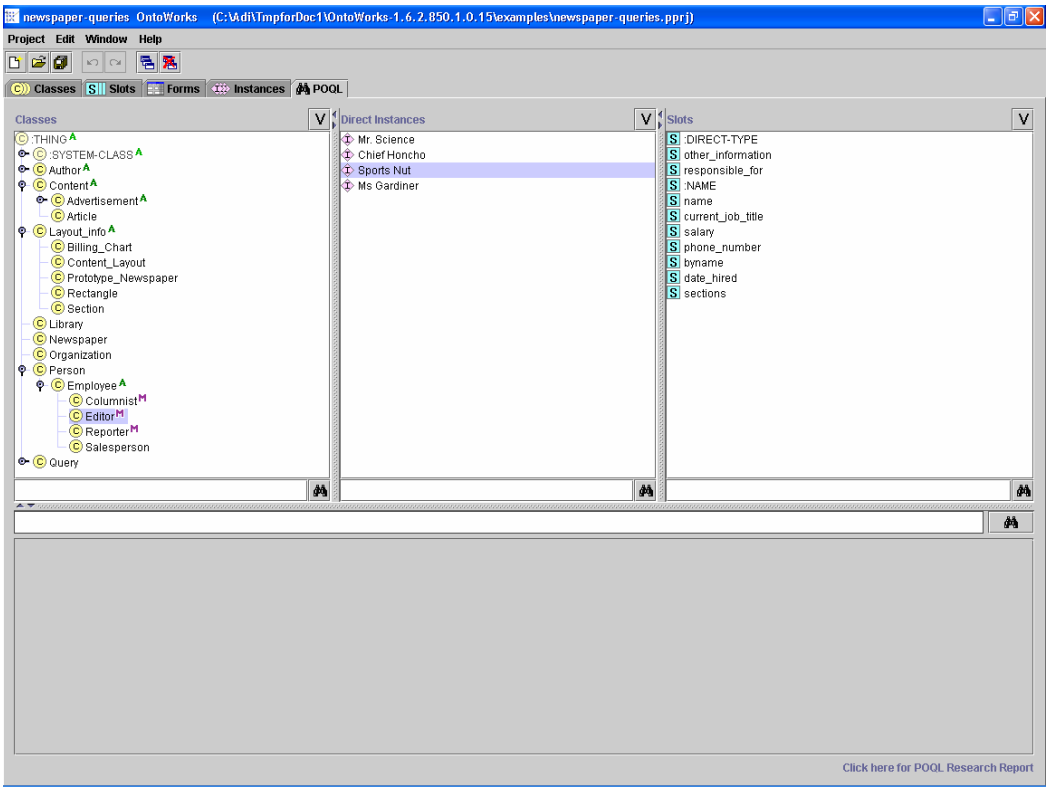
- QP.jj
- ParseException.java
- QP.java
- QPConstants.java
- QPTokenManager.java
- SimpleCharStream.java
- Token.java
- TokenMgrError.java

4.2 Utilization Guidelines

Before the initialization of the tab takes place, the user is asked which of the *Protégé_API* or the *OntoWorks_API* is to be used. By checking the *Use Demo CheckBox*, a file containing queries is appended to the *queries' history* (a list of queries which have already been run; it may be used for easy selection of a previous query).
This is how the POQL Tab should look like, if you have followed the above mentioned installation steps.



After the selections corresponding to the initial dialog box have been made, the tab is initialized:



The upper three panels correspond, respectively, to the class hierarchy, direct class' instances and own slots of the selected class or instance (depending on the order of selection of items in the previous two panels). By double clicking in one of the latter two panels, the user may append the *instance name* (as opposed to the *instance browser name* – see section 3.1) or the *slot name* to the cursor position in the *query text field.*

Queries are entered in a text entry field. They should obey the syntax (as is presented in section 3.1). In order to ensure correctness of the execution, it is necessary that frame names do not comprise characters other than alphanumeric and blanks.
A query may contain several types of Prolog constants. They should be entered according to the following rules:

- Strings are contained between pairs of the character \".
- The empty set is represented as "[]".
- The allowed numerical constants are integers and floats.
- Possible boolean constants are `true` and `false`.

Also, all frame names should be input in-between pairs of the character \'.

Results of the input query – all possible bindings of the *defined variables* are displayed in the panel below the query input field. If there are unbound variables with no bindings and still the query constraints are satisfied, such results are displayed using the `\*` character. When the value to which a variable is bound is undefined, it is displayed as the empty set `[]`.

Queries' history is saved in a list which allows access to previous queries by simply right-clicking in the query text entry area. Also, a file which has the same name with the opened ontology and the extension `.demo` is loaded at the initialization of the tab. It contains a list of queries, which are thus loaded in the queries' history and accessible for easy future access. This feature is enabled upon the selection of the API to be used.

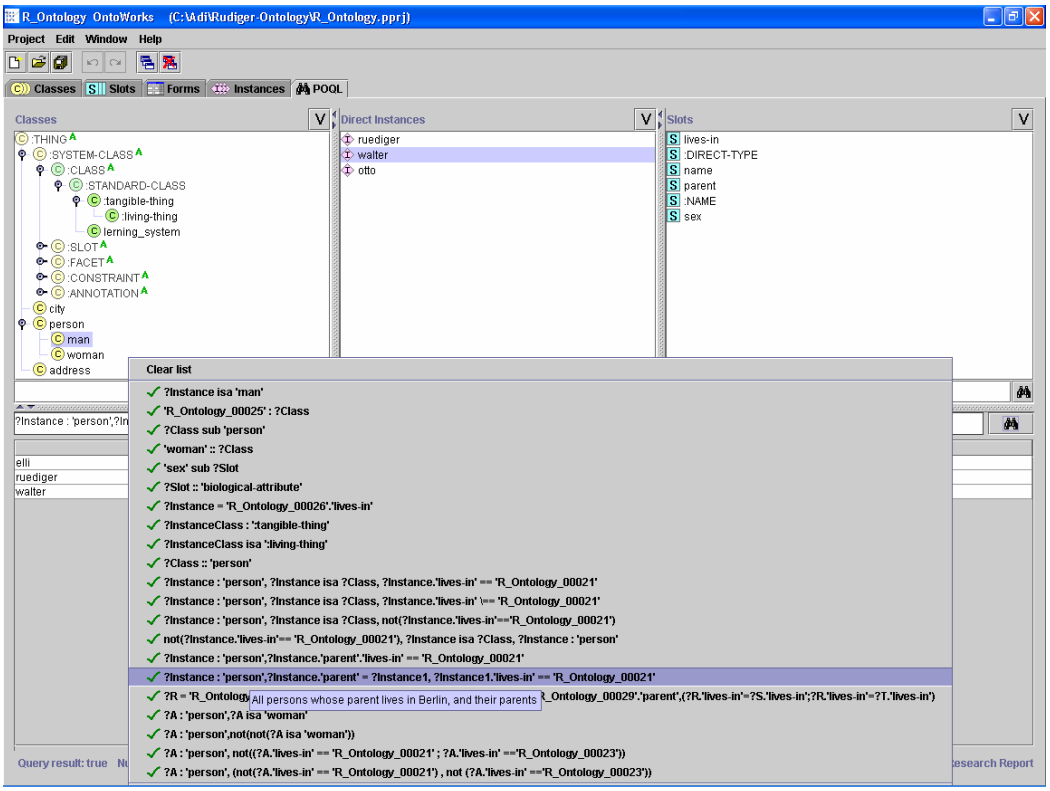
NOTE: Due to Interprolog, one may experience difficulties when loading an ontology once another has already been loaded. In this situation the application should be restarted.

5 Experimental Results

We chose to test **POQL** with ontologies built in Protégé 2000 [5], [9] and OntoWorks [13] knowledge base creational environments. Our decision was based on the fact that, among other, they present the advantage of integrating the OKBC model. We specially created an ontology structure inspired from the one presented in *Query Language for KRML* [12]. The results are shown for the above-mentioned ontologies and queries with various complexities, obtained on an Athlon XP 1800+, using the OntoWorks platform (although POQL may just as well be integrated in Protégé 2000).

5.1 Comprehensive Query Language for KRML

In order to show the results we have obtained in regards to the specifications in *Query Language for KRML* [12], we created an ontology structure with 82 frames, `R_Ontology`, inspired from the one presented in the above mentioned paper.



In order to have an easy way to query this ontology, we have generated the `R_Ontology.demo` file, which comprises queries which comprehensively include the use cases presented in *Query Language for KRML*. All these queries were executed both

using the OntoWorks API and the Protégé API, presenting the response time in both situations. The results are shown in the following table.

No. crt.	POQL Query for R_Ontology [12] (no. of frames = 82)	Protégé API (seconds)	OntoWorks API (seconds)
1.	?Instance isa 'man' ***?Instance*** ruediger walter otto	0.040	0.030
2.	'R_Ontology_00025' : ?Class ***?Class*** man person :THING	0.040	0.030
3.	?Class sub 'person' ***?Class*** man woman	0.041	0.040
4.	'woman' :: ?Class ***?Class*** person :THING	0.040	0.040
5.	'sex' sub ?Slot ***?Slot*** biological-attribute	0.030	0.050
6.	?Slot :: 'biological-attribute' ***?Slot*** sex	0.030	0.030
7.	?Instance = 'R_Ontology_00026'.'lives-in' ***?Instance*** muenchen	0.030	0.020
8.	?InstanceClass isa ':living-thing' ***?InstanceClass*** man person woman	0.040	0.030
9.	?Class :: 'person' ***?Class*** man woman	0.040	0.040
10.	?Instance : 'person', ?Instance isa ?Class, ?Instance.'lives-in' == 'R_Ontology_00021' ***?Instance*** ***?Class*** ruediger man nina woman	0.110	0.120
11.	?Instance : 'person', ?Instance isa ?Class, ?Instance.'lives-in' \== 'R_Ontology_00021' ***?Instance*** ***?Class*** walter man otto man elli woman luise woman	0.130	0.121

No. crt.	POQL Query for R_Ontology [12] (no. of frames = 82)	Protégé API (seconds)	OntoWorks API (seconds)
12.	?Instance : 'person', ?Instance isa ?Class, not(?Instance.'lives-in'=='R_Ontology_00021') ***?Instance*** ***?Class*** walter man otto man elli woman luise woman	0.131	0.110
13.	not(?Instance.'lives-in'== 'R_Ontology_00021'), ?Instance isa ?Class, ?Instance : 'person' ***?Instance*** ***?Class*** walter man otto man elli woman luise woman	0.161	0.140
14.	?Instance : 'person', ?Instance.'parent'.'lives-in' == 'R_Ontology_00021' ***?Instance*** ruediger walter elli	0.120	0.100
15.	?Instance : 'person',?Instance.'parent' = ?Instance1, ?Instance1.'lives-in' == 'R_Ontology_00021' ***?Instance*** ***?Instance1*** ruediger nina walter ruediger elli ruediger	0.120	0.140
16.	?R = 'R_Ontology_00025'.'parent' , ?S = 'R_Ontology_00024'.'parent', ?T = 'R_Ontology_00029'.'parent',(?R.'lives-in' = ?S.'lives-in'; ?R.'lives-in' = ?T.'lives-in') ***?R*** ***?S*** ***?T*** ruediger nina []	0.050	0.060
17.	?A : 'person',?A isa 'woman' ***?A*** elli luise nina	0.100	0.100
18.	?A : 'person',not(not(?A isa 'woman')) ***?A*** elli luise nina	0.130	0.130
19.	?A : 'person', not((?A.'lives-in' == 'R_Ontology_00021' ; ?A.'lives-in' =='R_Ontology_00023')) ***?A*** walter elli luise	0.090	0.100

No. crt.	POQL Query for R_Ontology [12] (no. of frames = 82)	Protégé API (seconds)	OntoWorks API (seconds)
20.	?A : 'person', (not(?A.'lives-in' == 'R_Ontology_00021') , not (?A.'lives-in' =='R_Ontology_00023')) ***?A*** walter elli luise	0.090	0.090

5.2 Extending the OKBC Assertion Language

Due to the inner incompleteness of the Prolog with freeze computational model, one might phrase queries which are *indefinitely postponed* (*undefined*) (e.g. **`?A sub ?B`**). For such queries, an error message is issued: "Unbound Variables Exception".

We shall refer as *acceptable (defined) queries* to all other queries (i.e. – queries which are not indefinitely postponed). For these queries, we have demonstrated their soundness in section 2.3. Many of the indefinitely postponed queries may be transformed to be acceptable by enriching the specification of the query (e.g. for the above example, a transformed acceptable form is **`?A :: ':THING', ?A sub ?B`**).

All extensions of POQL as compared to the OKBC Assertion Language [6] (the use of disjunction, negation and so on) refer implicitly to *acceptable queries* and ensure the return of the correct answer (see section 2.3).

All queries in the above table are *acceptable queries*. The use of logical operators allows the construction of syntactically different queries although semantically equivalent. For instance, the queries 11, 12, and 13 in the above table, though phrased in a different form, have the same semantic and therefore return the same answer. Their semantic negation is given as query 10, and the answer is negated as well.

As an emphasis, for acceptable queries, **double negation** and the applying of **DeMorgan's Laws** both work, due to the safety of negation as failure ensured by the Prolog with freeze, as illustrated in the queries 17, 18 for double negation (**`not(not(A))`** $\leftrightarrow$ **`A`**), respectively 19, 20 for DeMorgan's Laws (**`not(A;B)`** $\leftrightarrow$ **`not(A), not(B)`**).

5.3 Beyond the "peculiarities" of SQL

In [2] on page 129, H. Garcia-Molina, reports "*the peculiar way in which SQL handles OR operations in where-clauses. Supposing we have three unary relations R, S and T, and we wish to compute R* $\cap$ *(S* $\cup$ *T). Supposing each of these relations has a single attribute A, we might expect the following SQL query to do the trick.*

```
select R.A
from R, S, T
where R.A = S.A or R.A = T.A;
```

Unfortunately, if T is empty, the SQL result is empty, even if there are elements in R∩S".

Fortunately, **POQL** completely eliminates these shortcomings, in a similar fashion to the LOREL language, also presented by Molina in [2].

As an illustration of this, the very same example is shown in POQL form in query 16 in the above table, where we obtain the expected correct answer, although T is empty.

5.4 Checking Ontology Consistency

POQL was tested on a number of four ontologies, with complex queries representing various consistency checking. All queries were run both using the Protégé and the OntoWorks API. The following tables show the number of results along with the execution time. All queries may be found in the associated files: `Level8-withConstr-Restriction.demo`, `Newspaper-queries.demo`, `Organizational_Model.demo` and are available for consulting if the demo check box is checked. The consistency tests refer to the multiple inheritance property(query1), the list of classes which have at least two distinct subclasses(query 2), common boundaries of trees rooted in different classes(query 3) and so on. The tests show both the possibility of querying different ontologies and the complexity of the queries themselves. The large values for the response time correspond to complex queries which search thoroughly the ontology space.

Level8-withConstr-Restriction – 331 frames

	Query	Protégé API (seconds)	OntoWorks API (seconds)	No. of results
1	?ClassA :: ':THING' , ?ClassA sub ?SuperA1, ?ClassA sub ?SuperA2, ?SuperA1 \==?SuperA2	13.850	7.551	6
2	?ClassA :: ':THING' , ?SubA1 :: ?ClassA, ?SubA2 :: ?ClassA, ?SubA1 \== ?SubA2	111.080	114.134	8900
3	?ClassA sub ':THING', ? ClassB sub ':THING', ? ClassB \== ?ClassA, ?IstCls :: ? ClassA, ?IstCls : ? ClassB	11.066	21.040	37
4	? ClassA sub 'EveryThing', 'smallDieselMotor-1' : ? ClassA	0.050	0.030	1
5	? ClassA :: ':DCX_CF_SYSTEM_CLASS', ? ClassB :: ':DCX_CF_SYSTEM_CLASS', ? ClassA \==?ClassB, ?SubA sub ? ClassA, ?SubB sub ? ClassB, ? ClassA sub ?SuperAB, ? ClassB sub ?SuperAB	6.299	6.198	2

Newspaper-queries – 189 Frames;
(http://protege.stanford.edu/ontologies.html)

	Query	Protégé API (seconds)	OntoWorks API (seconds)	No. of results
1	?ClassA :: ':THING' , ?ClassA sub ?SuperA1, ?ClassA sub ?SuperA2, ?SuperA1 \==?SuperA2	4.300	4.100	6
2	?ClassA :: ':THING' , ?SubA1 :: ?ClassA, ?SubA2 :: ?ClassA, ?SubA1 \== ?SubA2	5.047	5.377	192
3	?ClassA sub ':THING', ?ClassB sub ':THING', ?ClassB \== ?ClassA, ?IstCls :: ?ClassA, ?IstCls : ?ClassB	9.614	13.990	20
4	?InstanceAB isa ?_ClassA , ?InstanceAB isa ?_ClassB, ?_ClassA :: 'Author' , ?_ClassB :: 'Person'	0.350	0.320	7
5	(?InstanceA isa 'Personals_Ad'; (?InstanceB isa 'Article', ?InstanceB.'containing_section'=?InstanceValueB, ?InstanceValueB.'section_name'\=="Lifestyle")), ?InstanceB.'published_in'=?InstanceA, ?InstanceA.'number_of_pages' =?InstanceValueA, not(?InstanceValueA<35)	0.300	0.281	4

Organizational_Model – 163 Frames;
(http://protege.stanford.edu/ontologies.html)

Query	Protégé API (seconds)	OntoWorks API (seconds)	No. of results
?ClassA :: ':THING' , ?ClassA sub ?SuperA1, ?ClassA sub ?SuperA2, ?SuperA1 \==?SuperA2	3.975	12.388	4
?ClassA ***?SuperA1*** ***?SuperA2***			
Organizational_Model_Diagram Network Organizational_Model_Entity			
Organizational_Model_Diagram Organizational_Model_Entity Network			
Organizational_Model_connector Connector Organizational_Model_Entity			
Organizational_Model_connector Organizational_Model_Entity Connector			
?ClassA :: ':THING' , ?SubA1 :: ?ClassA, ?SubA2 :: ?ClassA, ?SubA1 \== ?SubA2	7.451	11.418	780
?ClassA sub ':THING', ?ClassB sub ':THING', ?ClassB \== ?ClassA, ?IstCls :: ?ClassA, ?IstCls : ?ClassB	1.142	1.231	33
?ClassA :: 'Diagram_Entity', ?ClassB ::'Organizational_Model_Entity', ?SubAB sub ?ClassA, ?SubAB sub ?ClassB	21.591	31.385	7
?InstanceA isa 'Point', ?InstanceA.'x'=130, ?InstanceA.'y'=?InstanceValueA, ?InstanceB isa 'ObjectLocation', ?InstanceB.'location'=?InstanceValueB, (?InstanceValueB.'lower_right_corner'=?InstanceA ; ?InstanceValueB.'upper_left_corner' =?InstanceA)	0.350	0.280	1
?InstanceA : ?Class, ?InstanceB : ?Class, ?InstanceB.'x' \== ?InstanceA. 'x', ?InstanceA = 'INSTANCE_00016'	10.828	4.846	48

6 Conclusions and Possible Further Developments

We have described in this paper our approach regarding the development, implementation and usage of **POQL**, a new language for solving investigations in a portable knowledge base. It presents several advantages over other querying tools:

- it allows a query syntax that follows the Prolog logical form, therefore enabling the **further development** of interfaces that would communicate with our tool (i.e. Natural Language Interfaces, which make possible for users that are not familiar with the given ontologies to ask them queries.);
- the use of a Prolog with freeze meta-interpreter brings the possibility of generating complex queries; the solutions are computed taking advantage of the backtracking mechanism and the postponing technique. As a possible **future development**, one might extend the Prolog constraints with general expressions, also making use of a postponing technique;
- queries are introduced in the interface in a Prolog like manner, without being restricted to certain patterns, which leads to increased flexibility in searching the solution space;
- queries are parsed before sending them to the Prolog resolution mechanism, thus eliminating syntactic errors and ensuring the use of frame names belonging to the knowledge base space, before calling the Prolog solver;
- it contains a user friendly interface, integrated in both the Protégé 2000 and OntoWorks environments, which take full advantage of the possibilities of the described query language.

POQL conforms to the Prolog logical form, thus being independent towards any specific knowledge base creational environment. Its independence renders it fit for usage in other such environments and for future developments of interfaces that would communicate with **POQL**. Our implementation allows **further developments** for ontologies merging theory support (by using axiom-predicate as defined in section 2.2).

As stated in section 2.1., **POQL** overcomes the limitations of the OKBC Assertion Language [6] by introduction of disjunction, negation, relational operators, and also the limitations of OQL and SQL mentioned in [2]. The above statements are sustained by the examples presented in sections 5.2 and 5.3. Also, as stated in the same section, the distinction between classes and instances is not an absolute one due to the existence of metaclasses. Still, the current implementation of the POQL Kernel provides methods concerning metaclasses, which are not used in the present version. They might represent **future development** starting points for KRSs used for ontologies in which the distinction between classes and metaclasses is functionally relevant.

7 Acknowledgements

The authors wish to thank Kalman Pusztai form TUCN and Mugur Tatar from DC AG for the initiation of this contract, and also to Ruediger Klein for the professional collaboration during the development of this research project.

8 References

[1] Eriksson, H., Fergerson, R.W., Shahar, Y. & Musen M. A. (1999). Automatic Generation of Ontology Editors. *Twelfth Banff Knowledge Acquisition for Knowledge-based systems Workshop*, Banff, Alberta, Canada.

[2] Molina, H. G., Papakonstantinou, Y., Quass, D., Rajaraman, A., Sagiv, Y., Ullman, J., Vassalos, V. & Widom, J. (1997). The TSIMMIS approach to mediation: Data models and Languages. *Journal of Intelligent Information Systems*, 8(2), pag. 117-132.

[3] Muresan, T., Potolea, R., Muresan, S. (1998). Amalgamating CCP with Prolog, *Scientific Journal of Technical University Timisoara, Vol.43,57, no.4, 1998, Special Issue Dedicated to Third International Conference on Technical Informatics, CONTI'98*, Romania, pag.47 - 58 .

[4] Musen, M.A., Fergerson, R.W., Grosso, W.E., Noy, N.F., Crubezy, M. & Gennari, J.H. (2000). Component-Based Support for Building Knowledge-Acquisition Systems. *Conference on Intelligent Information Processing (IIP 2000) of the International Federation for Information Processing World Computer Congress (WCC 2000)*, Beijing.

[5] Noy, N.F., Fergerson, R.W. & Musen M.A. (2000). The knowledge model of Protege-2000: Combining interoperability and flexibility. *2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)*, Juan-les-Pins, France.

[6] Chaudhri, V.K., Farquhar, A., Fikes, R., Karp, P.D. and Rice, J.P. (1998). OKBC: A Programmatic Foundation for Knowledge Base Interoperability. *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI – 98)*, Madison, Wisconsin, AAAI Press.

[7] Noy, N.F., Musen, M.A. (1999). An Algorithm for Merging and Aligning Ontologies: Automation and Tool Support. *Sixteenth National Conference on Artificial Intelligence (AAAI-99), Workshop on Ontology Management*, Orlando, FL.

[8] Gruber, T.R. (1991). *A* translation approach to portable ontology specifications. *Knowledge Acquisition, **5***, pag. 199-220.

[9] Grosso, W., Eriksson, H., Fergerson, R., Gennari, J., Tu S., and Musen, M. (2000). Knowledge Modeling at the Millenium (The Design and Evolution of Protégé-2000), Stanford University.

[10] Noy, N.F., Musen, M.A. (2001). Anchor-PROMPT: Using Non-Local Context for Semantic Matching. *Proceedings of the Workshop on Ontologies and Information Sharing at the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, Seattle, WA.

[11] Janssen T. (1997). Compositionality. *J. van Benthem and A. ter Meulen, editors, Handbook of Logic and Language,and Linguistics*, pages 417-473. Elsevier Science.

[12] Rudiger K. A query constraint language for KRML, Daimler Chrysler AG.

[13] Hildebrandt, H., Lukibanov, O.Y., Keutgen, I., *OntoWorks Tutorial, Daimler Chrysler AG.*

[14] Noy, N.F., Musen, M.A. (2000). PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment, *Proceedings of the Seventeenth Conference on Artificial Intelligence (AAAI 2000)*, Austin, Texas.

[15] Saraswat, V.A. Concurrent Constraint Programming 1993, *MIT Press, ACM Doctoral Dissertatio Award and Logic Programming Series*.

[16] Cimpian, E., Mocan, A., Popovici, R., Tarcea H. (2002). KBQL: A Query Language for Protégé Based Ontologie, *International Conference on Automation, Quality and Testing, Robotics*, Cluj-Napoca, Romania .

[17] Sintek M. The Flora Query Tab, Querying Protégé 2000 with F-Logic. 2002. *http://dfki.uni-kl.de/~sintek/FloraTab*.

[18] XSB Prolog Reference. *http://xsb.sourceforge.net*

[19] JavaCC, *http://www.webgain.com/products/java_cc/*

[20] Java, *http://java.sun.com/j2se/*

9 APPENDIX A - Prolog Metainterpreter

```
%*************************************************************************
%
%      QUERIES SEMANTIC    /      Meta-interpreter of Prolog with freeze
%
%*************************************************************************

semantic(X, S):-!,sem_freeze(X, T/T, S).

sem_freeze(X) --> {!}, semantic(X) ,sem_queue.

sem_freeze(X,Val) --> {!},semantic(X,Val), sem_queue.

semantic((G1,G2)) --> {!},sem_freeze(G1),sem_freeze(G2).

semantic((G1;G2)) --> {!},(sem_freeze(G1);sem_freeze(G2)).

semantic(not(Goal)) --> {!}, postpone_n(Goal).

semantic((nott(Goal))) --> sem_freeze(Goal), {!, fail}.

semantic((nott(GOal))) --> {!}.

semantic(Goal) --> {functor(Goal,F,_), poql_op(F), !}, postpone(Goal).

semantic(Goal) --> {Goal=..[F,X,Y], eq_op(F), !}, sem_freeze(X,VX),
             sem_freeze(Y,VY), {Goal1 =..[F,VX,VY]}, system(Goal1).

semantic(Goal) --> {Goal=..[F,X,Y], math_op(F), !},sem_freeze(X,VX),
             sem_freeze(Y,VY), {number(VX), number(VY), Goal1 =..[F,VX,VY]},
             system(Goal1).

semantic(X,X) --> {(var(X);atomic(X)),!}.


semantic(A@B) --> {var(B), !}, postpone_p(dot3(A,B)).
semantic(A@B@C) --> {!}, sem_freeze(dot3(A,B), V), sem_freeze(V@C).
semantic(A@B) --> {!}, postpone_p(dot3(A,B)).

postpone_p(dot3(A,B)) --> {!}, freeze((A+A), sdot(A,B)).

semantic(A@B,V) --> {var(B),!}, postpone_p(dot3(A,B),V).
semantic(A@B@C,Val) --> {!}, sem_freeze(dot3(A,B),V), sem_freeze(V@C, Val).
semantic(A@B,V) --> {!}, postpone_p(dot3(A,B),V).
semantic(dot3(A,B),V) --> {!}, postpone_p(dot3(A,B),V).

postpone_p(dot3(A,B),V) --> {!}, freeze((A+A), dott3(A,B,V)).

semantic(Goal) --> {axiom_predicate(Goal), !}, user(Goal).

semantic(Goal) --> {!}, system(Goal).

user(Goal) --> {clause(Goal,Body)}, sem_freeze(Body).

system(true) --> {!}.
system(Goal) --> {!,Goal}.

postpone_n(Goal) --> {var_set(Goal, SetVar)}, freeze_n(SetVar, Goal).
```

```
postpone(Goal) --> {Goal=..[Op,X,Y], rename(Op,Op1), Goal1 =..[Op1,X,Y]},
            freeze((X+Y),Goal1).

freeze_n(SetVar, Goal) --> {ground_set(SetVar), !}, sem_freeze(nott(Goal)).
freeze_n(C,Goal,Ti/ (freeze_n(C,Goal),To),Ti/To):-!.

freeze((X+Y),Goal) --> {(nonvar(X);nonvar(Y)), !}, sem_freeze(Goal).
freeze(C,Goal,Ti/ (freeze(C,Goal),To),Ti/To):-!.

resuspend(X,Ti/To,(X,Ti)/To).

sem_queue(S/S,S/S) :- var(S),!.
sem_queue((freeze((X+Y),Goal),Ti)/To,So):-var(X),var(Y),!,
            sem_queue(Ti/To,S),!,resuspend(freeze((X+Y),Goal),S,So).
sem_queue((freeze(_,(Goal)),Ti)/To,So):-!,sem_freeze(Goal,Ti/To,So).

sem_queue((freeze_n(SetVar,Goal),Ti)/To,So):-not(ground_set(SetVar)),!,
            sem_queue(Ti/To,S),!,resuspend(freeze_n(SetVar,Goal),S,So).
sem_queue((freeze_n(_,(Goal)),Ti)/To,So):-!, sem_freeze(nott(Goal),Ti/To,So).

axiom_predicate(G):- functor(G,F,_), axiom_pred_list(L),member(F,L).

axiom_pred_list(L) :- L = [].

poql_op(Op):- member(Op,[:, ::, isa, sub]).

eq_op(Op) :- member(Op,[=, ==, \==]).

math_op(Op) :- member(Op, [<, >, >=, =<]).

rename(isa, isaa).
rename(:, :^).
rename(::, ::^).
rename(sub, subb).


%************************************************************************
%
%      ATOMIC QUERIES SEMANTIC
%
%************************************************************************
%
%  The following equivalences hold true:
%
%      poql-op(a,B) <=> forall(X, poql-op(a,X), Lx), member(B, Lx)
%
%      forall(X, poql-op(a,X), Lx) <=> api(a, _, poql-op), result(yes, Lx)
%
%  where api(a, _, poql-op), result(yes, Lx)   represents a Java API call
%
%************************************************************************

%************** ::        - subclass (direct or indirect) *****************
:- op(300, xfx, [::,::^]).
::^(A, B) :- var(A),!,api(_, B, ::), result(yes,Lx), member(A, Lx).
::^(A, B) :- var(B),!,api(A, _, ::), result(yes,Lx), member(B, Lx).
::^(A, B) :- api(A, B, ::), result(yes, _).


%*************** sub - direct subclass ***********************************
:- op(300, xfx, [sub, subb]).
subb(A, B) :- var(A),!,api(_, B, sub), result(yes, Lx), member(A, Lx).
subb(A, B) :- var(B),!,api(A, _, sub), result(yes, Lx), member(B, Lx).
subb(A, B) :- api(A, B, sub), result(yes, _).
```

```
%*************** :        - instances(direct or indirect)*******************
:- op(300, xfx, [:, :^]).
:^(A, B) :- var(A),!,api(_, B, :), result(yes, Lx), member(A, Lx).
:^(A, B) :- var(B),!,api(A, _, :), result(yes, Lx), member(B, Lx).
:^(A, B) :- api(A, B, :), result(yes, _).


%*************** isa   - direct instances ******************************
:- op(300, xfx, [isa, isaa]).
isaa(A, B) :- var(A),!,api(_, B, isa), result(yes, Lx), member(A, Lx).
isaa(A, B) :- var(B),!,api(A, _, isa), result(yes, Lx), member(B, Lx).
isaa(A, B) :- api(A, B, isa), result(yes, _).


%*************** . <=> @    - attributes' values (dott3, sdot)*************
:- op(150, xfy, @).

dott3(A,B,V) :- nonvar(A), nonvar(B), !, api(A, B, @), result(yes,LV),
             (LV == [] -> V = []; member(V, LV)).
dott3(A,B,V) :- var(B), !, api(A, _, @), result(yes, LB),
             (LB == [] -> B = []; member(B, LB)), dott3(A,B,V).

sdot(A, B) :- nonvar(A), nonvar(B), !, api(A, B, @), result(yes, _).
sdot(A, B) :- var(B), !, api(A, _, @), result(yes, LB),
             (LB == [] -> B = []; member(B, LB)).
```

10 **APPENDIX B -** Knowledge Base Accesss Methods

We give here a list of **POQL**_API methods one may use to access a knowledge base.

```
      /**
       * sets new Protege Knowledge Base
       * @param kb edu.stanford.smi.protege.model.KnowledgeBase
       */
public static void setKnowledgeBase(kb)

      /**
       * sets new OntoWorks Knowledge Base
       * @param kb edu.stanford.smi.protege.model.KnowledgeBase
      */
 public static void setOWI_KnowledgeBaseManager(kb)

      /**
       * sets new Prolog Engine
       * @param prologEngine com.declarativa.interprolog.PrologEngine
       */
 public static void setPrologEngine(prologEngine)

      /**
       * checks if cls is the name of a class
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
 public static Result isClass(cls)

      /**
       * checks if slot is the name of a slot
       * @param slot java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
 public static Result isSlot(slot)

      /**
       * checks if metaCls is the name of a metaclass
       * @param metaCls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
 public static Result isMetaclass(metaCls)

      /**
       * gets list of names for direct Instances of the class cls
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
 public static Result getDirectInstances(cls)

      /**
       * gets list of names for all Instances (direct or indirect)of the
      class cls
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
 public static Result getInstances(cls)

      /**
       * checks if son is a subclass of parent
```

```
       * @param son java.lang.String
       * @param parent java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result subclassOf(son, parent)

      /**
       * checks if son is a direct subclass of parent
       * @param son java.lang.String
       * @param parent java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result directSubclassOf(son, parent)

      /**
       * checks if parent is a superclass of son
       * @param parent java.lang.String
       * @param son java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result superclassOf(parent, son)

      /**
       * checks if parent is a direct superclass of son
       * @param parent java.lang.String
       * @param son java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result directSuperclassOf(parent, son)

      /**
       * checks if parent is a direct superslot of son
       * @param parent java.lang.String
       * @param son java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result directSuperslotOf(parent, son)

      /**
       * checks if son is a direct subslot of parent
       * @param son java.lang.String
       * @param parent java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result directSubslotOf(son, parent)

      /**
       * checks if parent is a superslot of son
       * @param parent java.lang.String
       * @param son java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result superslotOf(parent, son)

      /**
       * checks if son is a subslot of parent
       * @param son java.lang.String
       * @param parent java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result subslotOf(son, parent)

      /**
       * checks if inst is a direct instance of cls
       * @param inst java.lang.String
```

```
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result directInstanceOf(inst, cls)

      /**
       * checks if inst is a instance of cls
       * @param inst java.lang.String
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result instanceOf(inst, cls)

      /**
       * gets the direct type(class) of instance inst
       * @param inst java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getDirectTypeOf(inst)

      /**
       * gets the type(class) of instance inst
       * @param inst java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getTypeOf(inst)

      /**
       * gets list of names for all superslots of slot slot
       * @param slot java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getSuperslots(slot)

      /**
       * gets list of names for all direct superslots of slot slot
       * @param slot java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getDirectSuperslots(slot)

      /**
       * gets list of names for all direct subslots of slot slot
       * @param slot java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getDirectSubslots(slot)

      /**
       * gets list of names for all subslots of slot slot
       * @param slot java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getSubslots(slot)

      /**
       * gets list of names for all superclasses of class cls
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getSuperclasses(cls)

      /**
       * gets list of names for all direct superclasses of class cls
       * @param cls java.lang.String
```

```
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getDirectSuperclasses(cls)

      /**
       * gets list of names for all subclasses of class cls
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getSubclasses(cls)

      /**
       * gets list of names for all direct subclasses of class cls
       * @param cls java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getDirectSubclasses(cls)

      /**
       * gets value of slot at an instance
       * @param instS java.lang.String
       * @param slotS java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getSlotAtInstanceValue(instS, slotS)

      /**
       * gets own slots of instance
       * @param instS java.lang.String
       * @return com.utcn.POQL.ontologyAccess.Result
       */
public static Result getSlotsAtInstance(instS)
```

11 APPENDIX C - JavaDoc Files