

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Diploma Thesis Assignment

Visualization of Expressive Queries into OWL 2
Ontologies

June 2010

KOSTOV Bogdan

Obsah

| | | |
|----------|--|-----------|
| 1 | Introduction | 9 |
| 2 | Semantick web - Involved Technologies | 11 |
| 2.1 | OWL 2 | 11 |
| 2.2 | Introdiction to OWL 2 Structural Specification | 12 |
| 2.3 | RDF and SPARQL | 16 |
| 2.4 | SPARQL-DL | 20 |
| 3 | Query Graphical Interactive Designing and Visualization | 22 |
| 3.1 | Conjunctive ABox visual graph model | 23 |
| 3.2 | SPARQL-DL Graph Model - mixed ABox, TBox and RBox | 26 |
| 3.2.1 | Basic graph elements | 27 |
| 3.2.2 | Examples | 29 |
| 3.2.3 | Model limitations and proposed improvements | 30 |
| 3.3 | The graph editor | 32 |
| 4 | Query Editor Implementation | 33 |
| 4.1 | Used Technologies | 34 |
| 4.1.1 | OWL2Query | 34 |
| 4.1.2 | JGraph + jgraphaddons | 35 |
| 4.2 | The Query Graph Model | 36 |
| 4.3 | The graph Editor implementation | 39 |
| 4.3.1 | The undo support and the query models bad design characteristics | 39 |
| 4.4 | Upgrading to the new Visual Graph Model | 40 |
| 4.4.1 | The renderers | 42 |
| 5 | The Query Designer in Practice | 44 |
| 6 | Conclusion | 45 |

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Bogdan Kostov
Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný
Obor: Kybernetika a měření, blok KM2 – Umělá inteligence
Název tématu: Vizualizace expresivních dotazů do OWL 2 ontologií

Pokyny pro vypracování:


1. Seznamte se s jazyky rodiny OWL 2, zejména OWL2-DL, a dotazovacími jazyky SPARQL a SPARQL-DL.
2. Navrhněte vhodnou vizualizaci pro dotazy v jazyku SPARQL-DL s ohledem na jejich sémantiku a sémantiku OWL2-DL.
3. Navrhněte a implementujte nástroj pro interaktivní tvorbu SPARQL-DL dotazů s využitím vizualizace navržené v bodě 2. Dbejte na uživatelskou ergonomicitu a využijte interakce s inferenčním strojem pro OWL2-DL pro navigaci uživatele tvorbou dotazu.
4. Porovnejte navrženou vizualizaci se současnými možnostmi tvorby a vizualizace dotazů v existujících systémech, např. Protégé.

Seznam odborné literatury:

- [1] OWL 2 Structural Syntax and Specification (W3C Recommendation), October 2009.
- [2] SPARQL Query Language for RDF (W3C Recommendation), January 2008.
- [3] Sirin E., Parsia B. : SPARQL-DL: SPARQL Query for OWL-DL, OWLED 2007.
- [4] Kremen P., Sirin E. : SPARQL-DL Implementation Experience, OWLED 2008.

Vedoucí diplomové práce: Ing. Petr Křemen

Platnost zadání: do konce letního semestru 2010/2011


prof. Ing. Vladimír Mařík, CSc.
vedoucí katedry




doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 9. 12. 2009

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Bogdan Kostov
Study programme: Electrical Engineering and Information Technology
Specialisation: Cybernetics and Measurement – Artificial Intelligence
Title of Diploma Thesis: Visualization of Expressive Queries into OWL 2 Ontologies

Guidelines:


1. Become familiar with the languages of the OWL 2 family, namely OWL2-DL, and query languages SPARQL and SPARQL-DL.
2. Design a suitable visualization of SPARQL-DL queries with respect to their semantics and semantics of OWL2-DL.
3. Design and implement a tool for interactive SPARQL-DL query design and execution using visualization designed in point 2. Take into account ergonomics of the tool and make use of the interaction with OWL2-DL inference engine to navigate user through query design.
4. Compare designed visualization with current approaches of query creation and visualization in existing systems, e.g. Protégé.

Bibliography/Sources:

- [1] OWL 2 Structural Syntax and Specification (W3C Recommendation), October 2009.
- [2] SPARQL Query Language for RDF (W3C Recommendation), January 2008.
- [3] Sirin E., Parsia B. : SPARQL-DL: SPARQL Query for OWL-DL, OWLED 2007.
- [4] Kremen P., Sirin E. : SPARQL-DL Implementation Experience, OWLED 2008.

Diploma Thesis Supervisor: Ing. Petr Křemen

Valid until: the end of the summer semester of academic year 2010/2011


prof. Ing. Vladimír Mařík, CSc.
Head of Department




doc. Ing. Boris Šimák, CSc.
Head

Prague, December 9, 2009

Abstract

After the World Wide Web Consortium (W3C) introduces the newer version of the Web Ontology Language (OWL 2) in 2009, a demand for a new more expressive query language for OWL 2 DL ontologies was rising which resulted in the creation of the SPARQL-DL. This document describes the designing, development and integration of a graphical visualization and editing query tool for the semantic web based on the new query language SPARQL-DL. This document introduces the current state of the semantic web and explains the need of a query tool that will ease the query building process. I continue with more detailed introduction to the technologies and standards involved in the semantic web, namely OWL2 SPARQL SPARQL-DL. Then I describe two designed graphical models and their basic features after which I discuss their capabilities and suggest feature extensions of the model. Afterwards I discuss the implementation of the tool. Firstly I give an overview of the used technologies and libraries. In subsequent chapters I present the detailed description of the designed API and present some implementation design decisions. Next I shortly discuss the possibilities for integration of the tool in practice. Finally summarize the reached results.

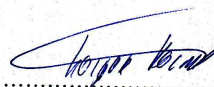
Abstrakt

Poté co World Wide Web Consortium (W3C) uvedly novou verzi standardu Web Ontology Language (OWL 2) v roce 2009, velká část nástrojů pro sémantický web které stavěli na této technologii, začal narůstat požadavek na nový expresivnější dotazovací jazyk pro OWL 2 DL ontologie. Díky tomu vznikl SPARQL-DL. Tento dokument popisuje návrh, vývoj, a integrace nástroje pro grafickou vizualizace a editace dotazu pro sémantický web, který staví na novém dotazovacím jazyku SPARQL-DL. Tento dokument uvádí do aktuálního stavu sémantického webu a vysvětluje potřebu dotazovacího nástroje který usnadní proces tvorby dotazu. Potom pokračuji detailním úvodem do technologii a standardu příslušné sémantickému webu, konkrétně OWL 2, SPARQL a SPARQL-DL. Potom popisuji dva navržené grafické modely a jejich základní vlastnosti. Potom probírám jejich dovedenosti a naznačuji budoucí rozšíření modelu. Nasleduje a rozbor implementace nástroje. Začínám s přehledem použitých technologie a knihovny. V dalších několik kapitol uvádím detailní popis navrženého API a zmiňuji se o několik návrhové rozhodnutí. Potom krátce zmiňuji možnosti pro integrace nástroje v praxe. Na konec shrnují dosažený stav.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v příloženém seznamu.

V Praze dne 14.5.2010



.....
podpis

Poděkování

Chtěl bych poděkovat všem lidem, bez nichž by nemohla tato práce vzniknout, především svému vedoucímu diplomové práce *Ing. Petr Křemen* za vedení, připomínky a poznámky k práci. Dále bych rád poděkoval osobám blízkým, zejména za jejich morální podporu.

1 Introduction

The semantic web community is in the middle of an important step of accepting a new version of one of its most fundamental standards, the web ontology language OWL 2. The OWL 2 as well its older version OWL is a formal or logical language [4]. Some of the advantages that come along with the new language are clearer syntax and semantics. The language has been also enriched with new constructs. In the case of OWL 2 DL the expressivity has been kept under the boundaries of the SROIQ description logic which offer computational advantages for practical reasoning [8]. To gain the advantages over the old OWL, the community started rewriting semantic web programs and tools so that they comply with the new standard. This major event was also a trigger for other changes of aspects and standards. For example the query languages available for OWL 2 DL were either not expressive enough or had too complex syntax which also didn't have much common with the newer semantics. So a new query language was demanded. A new query language SPARQL-DL was developed that started a new chain of events like creating new reasoning and development tools. Its objectives are to have clear syntax and semantics that follow OWL 2 DL and to extend expressivity. Another advantage is that the language is build so that new reasoning tools can be built onto existing ones.

While fast answering a query is important and have been taken care of, query designing is still left for the text editor. A new graphical query development tool is needed. This is common approach in the semantic web community. Developers build ontologies containing hundreds of entities and relations and even more individuals with graphical tools like Protégé and others. Such need of an instrument is not just a luxury, conversely it is essential.

In this work I design a graphical query designer (GQD - DL) for SPARQL-DL query language. Design objectives are: to graphically represent query terms and the relations between them, to ease the search for particular element in the queried ontology, the possibility of integration of the query designer in other environments such as Protégé or in the semantic web application.

For the graphical query representation it is natural to select a graph structure. Simply description of the graph that is used is that nodes of the graph represent terms and the entities to which they belong and edges in the graph represent relations between terms. Since SPARQL-DL queries are build of query atoms which are basically a triple

term, predicate, term

one term can be used more than once, this graph structure will keep the view more simple reducing the query elements. It is common that ontologies are quite big. For a query developer it is mandatory to be able to find the needed entity, relation or individual. This is achieved using context filtering. That is when for example someone wants to add a property to an object, the GQD can filter the available properties to those having their domain as a subclass of the entity to which the object belongs. To illustrate better the power of a graphical representation of a query here I show an example query written in SPARQL:

```

?C rdfs:subClassOf ..:x .
..:x rdf:type owl:Restriction .
..:x owl:onProperty ex:q .
..:x ?p ?C .

```

This query is example from [10] which finds classes $?C$ that are subclasses of a property restriction over the property $ex:q$. The property is restricted to the range of the class that we are looking for $?C$ and in the same time we are asking for the quantifier of the property restriction with the variable $?q$. This query is quite unusual because it uses a variable at the place at the quantifier. To understand what this query is supposed to do, it may take a while. Designing such queries is also as hard. Now I will show a possible graphical representation using a graph. On image 1 we can see a simple graph with three nodes. The shape of the nodes denotes what there are used for what is their semantics. For example the red node, in this particular example graph represents a class node. The ellipse is used to represent a class description node of thy restriction. The rectangle node represents a property. Edges can have can have decorations at their connection points to distinguish between the meaning of different edges. The triangle arrow is used to denote a sub class relation in UML. Labels can be placed on the edges to further specify their semantics. The color may also be used to distinguish semantics. For example the blue is used to denote constants and constructs while the red is used to denote variables.

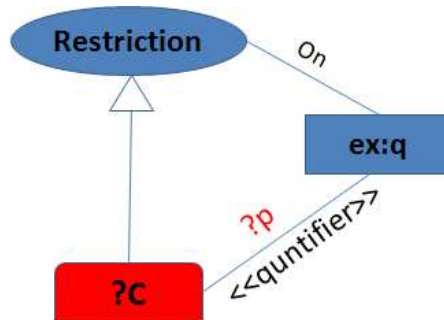


Figure 1: Type Node

In the next chapter I will introduce in more details the technologies on which the GQD-DL is build. I will start with description OWL 2 – DL its syntax and differences over OWL. Next I present SPARQL-DL query language. Then I present the chosen graph model.

2 Semantick web - Involved Technologies

2.1 OWL 2

OWL 2, as its previous version OWL, is based on the formalism of description logics and can have its expressivity reduced to offering better computational properties. These restrictions are subsets of OWL 2 and are called OWL 2 profiles. This introduction to OWL 2 will be targeted mainly to one of its most common profiles OWL 2 DL which happens to be the optimal choice due to the combination of its expressivity and computational capabilities.

OWL 2 is the new ontology language. Its major difference from its previous version OWL is in the extension of OWL vocabulary and expressivity. Moreover the specification introduces functional style syntax which closely follows the abstract structural specification and is used to define OWL 2 ontology semantics. Additionally this syntax is designed so that ontologies can be written in compact form. The functional syntax is tailed for OWL 2 and it can be used as a basis for a compatible API models and tools. One of the reasons the definition of the functional syntax was introduced in OWL 2 is because in OWL which did not specify any specific syntax led to some misusing of OWL and problems between different API models and tool designers. OWL 2 overcomes this issue by creating an abstract structural specification which and defines a functional syntax that follows closely the structural model. Despite the changes of the definition of the abstract structure and its semantics OWL 2, the underling technologies that OWL was build on are kept. The most common syntax that OWL 2 ontologies are serialized and exchanged is RDF/XML (Resource Definition Framework). This underling technology supplies semantics and well designed tools such as reasoners and query languages as SPARQL for RDF. Another aspect of OWL specification is that it provides mapping between the underling semantics of RDF graphs and the new structural specification. This gives another way of interpreting OWL 2 ontologies as RDF graphs as it was with OWL thus making it possible to take advantage of existing RDF reasoning tools. In the case of OWL 2 DL this mapping is not only bidirectional meaning that having any OWL 2 DL ontology can be converted to a RDF graph and the other way around, but also the later conversion will generate the same input ontology. Moreover the RDF graph representing an OWL 2 DL ontology will have the same expressivity and thus its computational advantages. For more detail on the OWL 2 specification see [8].

As mentioned earlier OWL 2 DL language is based on description logics. The concrete informal name of the description logic backing up OWL 2 DL is SROIQ (D) (SROIQ with data types). This is an extension to the SHOIN (D) description logic that was used in OWL 1 DL. The letters in the name of the description logic are abbreviations of groups of constructs that are allowed to be used. Here is a short description of the SROIQ description logic.

S is abbreviation for the ALC with the additional construct for transitive roles. **AL** abbreviation for the Attributive Language which allows concept expressions containing the following constructs: atomic concept negation, concept intersection, universal restrictions and limited existential quantifications.

R stands for: limited complex role inclusion axioms, reflexivity and irreflexivity, role disjointness.

O stands for nominals. (Enumerated classes of object value restrictions - owl:oneOf, owl:hasValue).

I is for inverse properties.

Q is qualified cardinality restrictions.

SROIQ is decidable, meaning that there is an effective method for determining the membership of formulas in the theory. This can be proved by using a tableau based algorithm that decides the consistency of a SROIQ concept w.r.t. a reduced RBox . For more details on the prove of and the extension of the new and more expressive SROIQ (D) and for practical reasoning see [5, 6] .

2.2 Introduction to OWL 2 Structural Specification

The core building blocks of OWL 2 ontologies are entities literals and anonymous individuals. The entities are actually all elements of the ontology that are described by IRI (Internationalized Resource Identifier). Entities are divided into six types which are: class, object property, data property, annotation property, data type and named individual. Literals in the ontology are accompanied with a string representing their data type. Anonymous individuals is new concept but it has been around before but under the notion of a RDF's **blank node**. The most mandatory elements in an ontology are the entities. They represent the domain and given their identifiers, their IRIs, entities are commonly thought as the vocabulary of the ontology. Next I will describe the meaning of each of the entities and will show examples of axioms dealing with them. In following examples I will use a prefixed version of IRIs **a:Person** where the **a:** stands for the abbreviation of the IRI of the fictional ontology and the following part **Person** is the actual entity name or identifier in the referenced ontology.

The class entity interpreted as a set of individuals. There are different constructs or expressions that are provided by the language that manipulate with the class entity.

Expressions are assigned to IRI with the help of axioms. For example one can use an **SubClass(a:Clown a:Person)** to say that **a:Clown** is subclass of the more general class **a:Person**, that is every **Clown** is also a **Person**. The list of available class axioms is:

SubClass(a:C1 a:C2)

Equivalent(a:C1 ... a:C2)

Disjoint(a:C1 ... a:C2)

DisjointUnion(a:C a:C1 a:C2 ...)

The **SubClass(a:C1 a:C2)** axiom states that all individuals in **a:C1** are also contained in **a:C2**. The **Equivalent(a:C1 ... a:C2)** axiom asserts that all individuals in **a:C1** are also contained in **a:C2** and vice versa. For example the class **a:Father** is equivalent to the class **a:MaleParent**. The **Disjoint(a:C1 ... a:C2)** axiom states that non of the individuals in class **a:C1** is contained into the class **a:C2**. An example of disjoint classes is two class **a:Cat** and **a:Dog**. The **DisjointUnion(a:C a:C1 a:C2 ...)** axiom states that all individuals in **a:C** are also contained in the union of the classes **a:C2** , **a:C2** ... which are declared to be disjoint by the same axiom.

For class expressions one can chose from a wide variety of set operations such as intersection, union, object and property restrictions , object and data property cardinality restrictions. A full list including details and interpretation is presented at [8]

The next entity type is the object property which can be thought of as a binary relationship between two individuals. There is only one object property expression and that is:

InverseObjectProperty(ObjectProperty).

This expression states that if two **i1** and **i2** are in an **InverseObjectProperty(ObjectProperty)** relation if and only if **i2** and **i1** are in an **ObjectProperty** relation. In OWL 2 new object property axioms have been added. Here is a list of all supported property axioms:

SubObjectpropertyOf(a:P1)
EquivalentObjectProperties(a:P1 a:P2 ...)
DisjointObjectProperties(a:P1 a:P2 ...)
InverseObjectProperties(a:P1 a:P2)
ObjectPropertyDomain(a:P a:C)
ObjectPropertyRange(a:P a:C)
FunctionalObjectProperty(a:P)
InverseFunctionalObjectProperty(a:P)
ReflexiveObjectProperty(a:P)
IrreflexiveObjectProperty(a:P)
SymmetricObjectProperty(a:P)
AsymmetricObjectProperty(a:P)
TransitiveObjectProperty(a:P)

The first three axioms have the same semantics as the corresponding class axioms mentioned earlier. The **InversObjectProperties(a:P1 a:P2 ...)** axiom is used to declare inverse relations, that is when the pair of individuals, **i1** and **i2** is in a **a:P1** relation, then the reversed ordered pair, **i2** and **i1** is of the relation **a:P2**. The axioms **ObjectPropertyDomain(a:P a:C)**, **ObjectPropertyRange(a:P a:C)** assert the domain and the range of the property **a:P**. The **FunctionalObjectProperty(a:P)** axiom defines the property **a:P** that permits only one value for each argument. For example the **a:hasFather** property has this characteristics because each argument **child** has only one value **parent**. The **InverseFunctionalObjectProperty(a:P)** axiom states that the inverse property of **P** is functional. For example think of the proerty **a:hasChild**, only one person can be the **father** of one childe **childe**. The **ReflexiveObjectProperty(a:P)** axiom declares that eache element is related to itself by the property **a:P**. Example for this construct is the **a:areRelatives** because everyone is a relative to himself. The **IrreflexiveObjectProperty(a:P)** axiom states in a sense the opposite, the property **a:P** any object to itself. In this case a good example is the **a:ParentOf** nobody can be a parent of himself. The **AsymmetricObjectProperty(a:P)** axioim assert that

the property **a:P** is symmetric, that is when a two individuals, **i1** and **i2** are related by the property then the reversed ordered pair **i2** and **i1** is also related by it. An example of symmetric property is **a:areRelatives**. The **AsymmetricObjectProperty(a:P)** axiom does the opposite, that is the property **a:p** do not contains any symmetric individual pairs. For example the **a:hasFather**, **a:olderThan**. The last object property axiom is the **TransitiveObjectProperty(a:P)**. It assumes that if the two pairs (**i1**, **i2**) and (**i2**, **i3**) are included in the property **a:P** then the property contains also the pair (**i1**, **i3**).

Next follows the data properties. Data properties can be thought as of object properties, however their domain is not a OWL 2 class but a data type, which makes the domain and the range of data properties disjoint. For data properties there are not any expressions defined. The axioms available is actually a reduced list of the axioms available for the object properties. This reduction is due to the fact that some of the axioms of object properties assume that the domain and the range of the property are not disjoint, which is not the case of data properties. Here is the list of the data property axioms:

SubDataPropertyOf(a:P1)

EquivalentDataProperties(a:P1 a:P2 ...)

DisjointDataProperties(a:P1 a:P2 ...)

DataPropertyDomain(a:P a:C)

DataPropertyRange(a:P a:C)

FunctionalDataProperty(a:P)

The semantics of these axioms are the same as the semantics of their corresponding object property equivalents with the only difference that the value of the data property is a literal and not an individual.

The last group of axioms of interest are the individual assertions. Here is the list of axioms that involve individuals as their arguments:

SameIndividual(a:i1 a:i2 ...)
DifferentIndividual(a:i1 a:i2 ...)
ClassAssertion(a:C a:i)
PositiveObjectProperty(a:P a:i1 a:i2 ...)
PositiveDataProperty(a:P a:i l)
NegativeObjectProperty(a:P a:i1 a:i2 ...)
NegativeDataProperty(a:P a:i l ...)

The **SameIndividual(a:i1 a:i2)** states that **a:i1** is the same individual as **a:i2**. The second axiom, **DifferentIndividual(a:i1 a:i2)**, states that the individual **a:i1** is the different from **a:i2**. The next axiom is the class assertion **ClassAssertion(a:C a:i)**, which states that the individual **a:i** is an instance of the class **a:C**. The axiom **PositiveObjectProperty(a:P a:i1 a:i2)** asserts that the individuals **a:i1** and **a:i2** are connected with the object property **a:P**. The axiom **PositiveDataProperty(a:P a:C l)** states that the individual **a:i** is connected to the literal **l**. The next two axioms are new in the OWL 2 language and they have the opposite effect than last two motioned above. The axiom **NegativeObjectProperty(a:P a:i1 a:i2)** states that the two individual are not connected with the object property **a:P**. The axiom **NegativeDataProperty(a:P a:i l)** states that the individual and the literal are not connected with the data property **a:P**.

Apart from the listed axioms and expressions there are also other defined by OWL 2. Like stated earlier the language offers a rich collection of class expressions. However are not very important for the definition of the GQD-DL graph model and so will be not listed here. The whole list of expressions with explanation of their semantics and other details are found in [8].

2.3 RDF and SPARQL

The existence of the OWL language and its newest version OWL 2 is based on the much more abstract conceptual web resource definition framework. This technology has been introduced for the first time by the W3C, along with the XML file format. In this section I will introduce the basics of the RDF frame work and SPARQL for RDF query language. RDF is has formal semantic, and abstract syntax with extensible vocabulary. The interpretation of the RDF data is determined by the interpretation of the vocabulary, a collection of all identifiable of named RDF terms used to describe the data. The source data can be physically stored in a RDF file format or viewed using middleware as RDF format. The recommended syntax for RDF is XML (Extensible

Markup Language). In a RDF file the data is basically stored as a collection of RDF triples of the form.

argument, predicate, value

RDF triples can be view as a binary relations where the *argument* and the *value* are connected or related to each other by the *predicate* relation. For example the sentence “The house has a flat roof” can be written as a RDF triple which will look like this:

The house, hasRoof, flat.

The collection of triples actually defines a graph data structure, where the *argumets* and the *values* of triples represent the nodes of the graph and the *predicates* represents the edges in the graph. Because this framework is created to work in the web environment, there are curtain rules for what can be put in the place of the individual elements of a triple. The main types of elements used in a RDF triple are URI’s (Universal Resource Identifier), literals and blank nodes. In the place of the *argument* one can put only an URI or a blank node. In the place of the *predicate* it is allowed to use only URI’. For the *value* there is no limitation assuming that guessed value is complies the RDF specification [2]. The blank node element is necessary concept for the RDF specification for it is designed to be used on the web. The blank node can be interpreted as a place holder for a resource that is not available or not known, which can happen quite easily on the web. For example, imagine the next situation where the users of a certain page upload poems. To submit a poem the page requires that the user fills in a form where supplying the author’s name, the name of the poem and the poem itself. And internally the data are stored in a RDF format with triples of the form:

Author , wrote , poem_name
poem , hasName , poem_name

If the owner of the web page decides to make all the fields of the form mandatory, then one do not need the use of blank nodes. However if the owner decides that maybe some authors may want to be anonymous or the uploaded does not know the name of the poem, than he may want to enable page to allow visitors to upload poems even though they do not supply the name of the poem or the name of the author. In that case if RDF do not have elements such as blank nodes, the triples that have to be asserted in the RDF graph will be inconsistent. So in this case when a user uploads only the poem itself the two RDF triples with blank nodes will be written as follows:

_:A , wrote , _:PN
poem , hasName , _:PN

As the example shows the triples are consistent even though there is missing information of resources. This example demonstrates only the basic use of blank nodes. However, blank nodes are not only limited for this use, for example one can create multi

arity relationship using a blank nodes. This is called a decomposition. For more details about blank nodes and other RDF elements . . . see [2].

If we have a source RDF graph describing some data we can query this data source using the SPARQL for RDF query language. SPARQL is based on the RDF's graph equivalence defined in the RDF specification [2], which states that two graph G and G' are the same if there exists a bijection mapping such that maps URI's and literals to themselves, blank nodes to blank nodes and a triple is in graph G if and only if the mapping of the same triple is in graph G'. SPARQL defines graph patterns that are RDF graphs containing variables. Next I will introduce the basics of how queries are represented and how variables are resolved in the basic graph patterns. Basic graph patterns are RDF graphs that contain variables for some of their nodes. So in its basic form a query is represented as a collection of RDF triples that may contain variables. As such basic graph patterns can be matched to a sub graph of the data source graph patterns. In the process of matching variables are bind to constants of the input graph. Details about the way basic graph patterns match to input graph is defined in [9]. The basic idea is that when we substitute the variables in the basic graph pattern we will obtain a valid RDF graph. If the graph obtained by substitution is a sub graph of the input graph then the basic graph pattern has found a match. The result for this match is the substitution of the variables. This is only a description of how the matching process should behave. Actual query tools take into account that the RDF graph is actually a list of triples. Then a sub graph is equivalent to a sub list of triples. So a query tool may start with the one of the triple patterns and try to match that pattern to a triple in the source graph. The result of the positive matching is the binding of the variables present in the pattern. All occurrences of the bound variables are replaced by the value they are bound to. Then , the next triple pattern is processed. When no matching triple is found in the whole input tree then a backtracking can be done, this is when we go one step back, unbind variables that were bound in that step and try to find a new matching pattern. Here is an example input graph. In following examples I will omit the prefix from the abbreviated URI, for example if *a.Name* will be written as *Name*.

song1 hasCompouser Compouser1

song10 hasCompouser Compouser1

Here is an example basic graph pattern query and the its result.

```

SELECT ?X, ?Y
WHERE{
    ?X hasCompouser ?Y
}

```

result

| X | Y |
|--------|------------|
| song1 | Compouser1 |
| song10 | Compouser1 |

In this example the query consists of one basic graph pattern containing only one triple pattern with two variables *?X hasCompouser ?Y* that has two variables. So the graph pattern has two variable nodes and one edge. Matching the graph pattern against the input graph we get two results because there were two substitutions available for the graph pattern which were a sub graph of the input graph. The first row of the result represents the first binding of variables. It shows that the variable *?X* is bound to the value *song2* and the variable *Y* is bound to the *Compouser1*. The second represents the second binding of variables.

The syntax of the query begins with a *SELECT* clause that is used to define the variables whose bindings will be listed in the result or the result variables. Next is the *WHERE* clause, this is where the basic graph pattern is placed. In SPARQL the first clause of the query, in this case *SELECT*, determines the form of the query. There are four query form:

SELECT

CONSTRUCT

ASK

DESCRIBE

The *SELECT* form as shown earlier is used to create queries that return the result in a tabular form. The *CONSTRUCT* query form is similar to the *SELECT* with the difference that it returns the result as the RDF graph created after substituting the variables. The *ASK* query form returns as a result a boolean value, that is *yes* or *no* when the query succeeded in finding a matching sub graph. The last query form *DESCRIBE* is used to describe a resource. As mentioned earlier queries are build using graph patterns. For now I have introduced the basic graph pattern which is a set of simple triple patterns with semantics of conjunction that is the graph pattern matches

if all the triple patterns match. There are also other graph patterns that with different semantics that make SPARQL quite flexible. Here is a list of the graph patterns:

Basic Graph Pattern

Group Graph Pattern

Optional Graph Pattern

Alternative Graph Pattern

Patterns on Named Graphs

The basic *basic graph pattern* was already discussed. The next pattern is the *group graph pattern*. This pattern has similar semantics as the *basic graph pattern* with the difference that it is not a set of triple patterns but a set of graph patterns with conjunction semantics. *Group graph pattern* are delimited by { and }. In the example above there was one graph pattern in the *where* clause consisting of one *basic graph pattern*. The *optional graph pattern* is used to mark parts of the query as optional. When an optional part does not match, this does not have an effect on the matching process of the mandatory (normal) part of the query. So is the query matches except for the optional part the binding succeeds. If the optional part matches than the result of the query is enriched by the bindings of the variables in the optional part. The a graph pattern is marked as optional using the syntax *OPTIONAL* { *graph pattern*}. The *alternative graph pattern* is used to make alternative graph patterns so that a query can find a result in more than one ways. This can be used when dealing with multiple URI dictionaries that have URI's with the same meaning. The syntax for *alternative graph pattern* is {*graph pattern*} *UNION* { *graph pattern*}. The last *patterns on named graphs* is used when we are querying multiple graphs.

Apart from this functionality SPARQL offers filters that can be used to filter the result. There are also ordering constructs that can sort or transform the result in various ways. The language also defines informative functions that can be used in filters to extract parts from the URI check whether its URI, blank node or literal etc. Another application is that SPARQL can be transformed into a relational algebra [3] which can take advantage of the capabilities of modern relational databases. For more details on SPARQL see [9].

2.4 SPARQL-DL

SPARQL-DL is a new query language for OWL 2 DL. The language offers new constructs that comply with the abstract functional syntax of OWL 2. SPARQL-DL introduces can handle mixed ABox, TBox and RBox queries. When ABox queries offer the possibility of optimization of computation [11] because its lack of expressivity. How

ever SPARQL-DL tries to push the boundaries of expresivity and still provide reasonable computational properties and possible optimization techniques [7]. The query language is based on SPARQL which supports an extension for a custom entailment regime [9, 10]. SPARQL-DL offers a lot more expressivity than other DL query languages. The basic building block for queries is a query atom. A query consists of collections of these query atoms. Next I will describe the basics of the query atoms are interpreted and then I will go through all the query atoms and their interpretation that are specified in [10, 7]. I will omit the $Annotation(s, p_a, o)$ because it is not relevant. Then I will list the new query atoms which are an extension to that specification.

Query atoms are defined over the OWL ontology vocabulary $V_O = (V_{cls}, V_{op}, V_{dp}, V_{ap}, V_{ind}, V_D, V_{lit})$ and the sets V_{bnode} and V_{var} . The ontology vocabulary components are interpreted as follows in the same order : classes, object properties, data properties, annotation properties, individuals, data types and literal. These sets together with the newly added sets for variables and blank nodes are pair wais disjoint. The semantics of the query atoms are based on the vocabulary V_O of the ontology O and the interpretation $\mathcal{I} = (\cdot^{\mathcal{I}}, \cdot^{\mathcal{I}})$. The query atoms can be compatible with the vocabulary if it satisfies conditions on the arguments of the query atoms. For example **ObjectProperty(P)** is compatible with the vocabulary V_O if $\mathbf{P} \in V_{op}$ and so on. There is another condition that must be satisfied for some other query atoms so that they can entail with the ontology vocabulary. For example the **Type(a, C)** query atom is not entailed just by being compatible with the ontology vocabulary V_O . There is also the requirement that the interpretation of \mathbf{a} must be a member in the interpretation of the class \mathbf{C} or formally $a^{\mathcal{I}} \in C^{\mathcal{I}}$. To make this work there are two more concepts defined in[7, 10]. First the semi-ground query atom which is the an atom with all of its variables bound except for maybe some blank nodes and second the mapping $\sigma : V_{ind} \cup V_{lit} \cup V_{bnode} \rightarrow \Delta^{\mathcal{I}}$. The mapping is defined so that it will interpret constants that is *individuals* and *literals* to themselves but it will provide a mapping for the blank nodes. This mapping is needed because of the blank nodes and their scope. The basic goal of blank nodes is to represent a missing resource as stated earlier and described in detail in [2, 9]. In the case of SPARQL-DL we talk blank nodes have the meaning of not asserted but inferred resource. Internally, in inference engines, we can think that the blank node or the inferred resources are identified by an internal id with scope that is not accessible from the outside. Using a blank node in the query atom we are trying to infer a resource and so the only way that match the id used in the query and the id in the inference engine is through the σ mapping. So that's why the domain of the sigma mapping is $V_{ind} \cup V_{lit} \cup V_{bnode}$, because it is allowed to use inferred resources in the query that are of type *individuals* or *literals*. Next I will list the query atoms and their entailment semantics including the compatibility of the semi-ground query atom. The tab.?? has three columns first for the query atom type, the second for the compatibility constraints and the last for the specific query atom semantics. The semantics of the query nodes proposed in SPARQL-DL are very similar to OWL 2 DL semantic [10, 7].

There are also a few more axiom which I will describe only informally. The *Direct-Type(a, C)* is used to state that the most general that the class C is the most general class of the individual. Query atom *DirectSubClassOf(C₁, C₂)* states that C_1 is the most

| Query Atom | Compatibility | Specific Semantics |
|----------------------------------|---|--|
| Type(a,C) | $a \in V_{ind}, C \in V_{cls}$ | $\sigma(a) \in C^I$ |
| PropertyValue(a, p, v) | $a \in V_{ind}, v \in V_{ind} \cup V_{lit}, p \in V_{dp} \cup V_{dp}$ | $\langle \sigma(a), \sigma(v) \rangle \in p^I$ |
| SameAs(a,b) | $a, b \in V_{ind}$ | $\sigma(a) = \sigma(b)$ |
| DifferntFrom(a,b) | $a, b \in V_{ind}$ | $\sigma(a) \neq \sigma(b)$ |
| SubClassOf(C_1, C_2) | $C_1, C_2 \in V_{cls}$ | $C_1^I \subseteq C_2^I$ |
| EquivalentClass(C_1, C_2) | $C_1, C_2 \in V_{cls}$ | $C_1^I = C_2^I$ |
| DisjointWith(C_1, C_2) | $C_1, C_2 \in V_{cls}$ | $C_1^I \cap C_2^I = \emptyset$ |
| ComplementOf(C_1, C_2) | $C_1, C_2 \in V_{cls}$ | $C_1^I = \Delta^I \setminus C_2^I$ |
| SubProperty(p_1, p_2) | $p_1, p_2 \in V_{op}$ or $p_1, p_2 \in V_{dp}$ | $p_1^I \subseteq p_2^I$ |
| EquivalentProperty(p_1, p_2) | $p_1, p_2 \in V_{op}$ or $p_1, p_2 \in V_{dp}$ | $p_1^I = p_2^I$ |
| EquivalentProperty(p_1, p_2) | $p_1, p_2 \in V_{op}$ or $p_1, p_2 \in V_{dp}$ | $p_1^I = p_2^I$ |
| ObjectProperty(p) | $p \in V_{op}$ | |
| DataProperty(p) | $p \in V_{dp}$ | |
| Functional(p) | $p \in V_{op}$ or $p \in V_{dp}$ | $\langle a, b \rangle \in p^I$ and $\langle a, c \rangle \in p^I \Rightarrow b = c$ |
| InverseFunctional(p) | $p \in V_{op}$ or $p \in V_{dp}$ | $\langle a, c \rangle \in p^I$ and $\langle b, c \rangle \in p^I \Rightarrow a = b$ |
| Transitive(p) | $p \in V_{op}$ or $p \in V_{dp}$ | $\langle a, b \rangle \in p^I$ and $\langle b, c \rangle \in p^I \Rightarrow \langle a, c \rangle \in p^I$ |
| Symmetric(p) | $p \in V_{op}$ or $p \in V_{dp}$ | $\langle a, b \rangle \in p^I \Rightarrow \langle b, a \rangle \in p^I$ |

Table 1: Query atoms and their entailment semantics

general class that is still a subclass of c_2 . The *StrictSubClassOf* C_1, C_2 query atom states that the class is C_1 is subclass of C_2 but it cannot be equivalent to it. For properties there are two more axioms that have the same semantics over the property sets of the ontology vocabulary as the last two class query atoms. For properties there are also defined two more query atoms *Asymmetric*(p), *Irreflexive*(p).

3 Query Graphical Interactive Designing and Visualization

This section begins with the description of the basic ABox visual graph model followed by examples. Next I describe some of the design issues that occurred when developing the graph model. In the following section I present the basic graphical elements and the semantics that they present, that is how they represent SPARQL-DL query atoms. Then a show some query examples both with their serialized form and as a graph which

is a screen shot from the prototype implementation of the model. I discuss objectively properties of the model. Then I try to make conclusions on the model and how it can be improved. The last sub section is dedicated to the graph creation and editing.

3.1 Conjunctive ABox visual graph model

The GQD DL is based on a graph model visualization. In this section I will describe the first graph model that was chosen in the first stage of the project. The first graph model was designed to be able to express conjunctive ABox queries. This graph mod has three element types, one node type and two edge types.

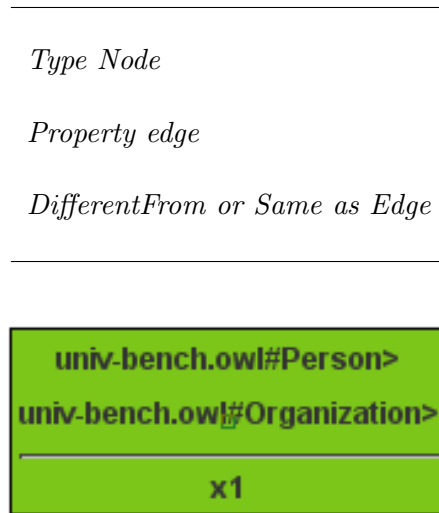
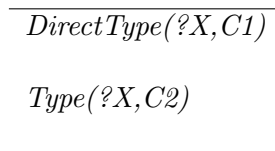


Figure 2: Type Node

On fig.3 is the graphical representation of the *Type Node* element. The node is represented as a bounded rectangle and contains several lines of labels. Bottom label denotes a variable name, individual name or a literal. Above the bottom label there is a separating line and other labels that denote ontology classes. The *Type Node* can represent one or more *Type* or *DirectType* query atoms for the same individual or variable. In the example above, the node represent the two *Type* query atoms with the object variable *x1* a the classes *univ-bench.owl#Person* and *univ-bench.owl#Organization*. More enhanced model should be able to handle the *DirectType* query atom and to show the difference between the two query atoms. This can be done with a dot or a square in front of the label that represents the class that is an argument of the *DirectType* query atom. Here is an example of one *Type* and one *DirectType* query atoms and the node that represents them.



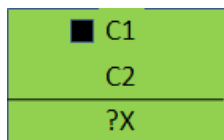


Figure 3: Type Node distinguishing a direct type query atom

The *node* element is the only element that represents a variable which is located in the bottom label. The model supports both types of SPARQL-DL variables *distinguished* and *undistinguished* filling the background of the node with a different color. The only constraint defined for the color is that *undistinguished* nodes, that is nodes that contain *undistinguished* variable, should look more neglectable than the other nodes in the graph. In the case when the node's bottom label is a literal, then the node represents a single literal term. In this case the node has only one type label which is the IRI or the abbreviated prefixed IRI of a data type. Here is an example of a literal node:

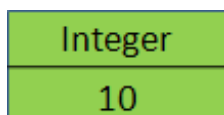


Figure 4: Literal node

10 nteger

The next elemnt is the *property* edge. This element represents one or more *PropertyValue* atoms with the same argument and the same value. The edge itself represents only the propertiy list the same way the node represents a type list with the difference that labels are drawn along the edge in a comma separated list. The other information, the argument and value of the *PropertyValue* query atom is represented by the nodes it edge is connected to, more specifically by their bottom labels. One end of the edge has an arrow head to mark the value node. The edge itself is drawn as a continues line. This element is used to represents both data and object properties the same way as the query atom it represents is used regardless of the type of the property. Here is an example of the query terms and their graph

PropertyValue(arg, prop1, val

PropertyValue(arg, prop2, val

The last graphical element in the graph model is the edge representing the *SameAs* and *DifferentFrom* query atoms. The edge is dashed and connects two nodes having

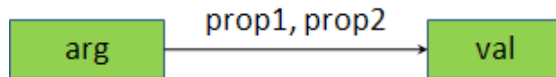


Figure 5: Property representation

individual IRI or variable in their bottom label. The edge has no arrows and has label that can be *SameAs* or *DifferentFrom*. Again as with the property edge, the arguments of the query atoms are the bottom labels of the nodes that the edge connects to. It is obvious that there is no meaning putting the two when one puts both labels at the same time, that's why the model permits only one label to be added between unique pair of nodes. Here is an example of this element and the query atom it represents.

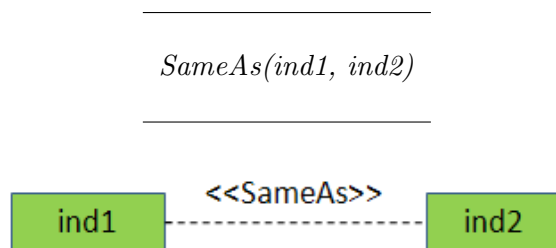


Figure 6: SameAs relation between individuals

The graph model representing conjunctive ABox queries covers this set of query atoms:

Type(ind1, C)

PropertyValue(arg, P, val)

SameAs(ind1, ind2)

DifferentFrom(ind1, ind2)

Here I defined the model for simple conjunctive ABox queries. This definition of this model was underestimated and was created after the implementation of the first version of the GQD-DL was. This implementation was based on the programmatic model that was not well designed to support the graphical model. For instance it did not supported the *DirectType*, *SameAs* and *DifferentFrom* query atoms. Another issue was that the model could only show a node as a simple node object without a type which can be quite common in a SPARQL-DL query.

Here is an example of the graph of the SPARQL-DL query:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
```

```

SELECT ?X ?Y1 ?Y2 ?Y3
WHERE
{
  ?X rdf:type ub:Professor .
  ?X ub:worksFor <http://www.Department0.University0.edu> .
  ?X ub:name ?Y1 .
  ?X ub:emailAddress ?Y2 .
  ?X ub:telephone ?Y3 .
}

```

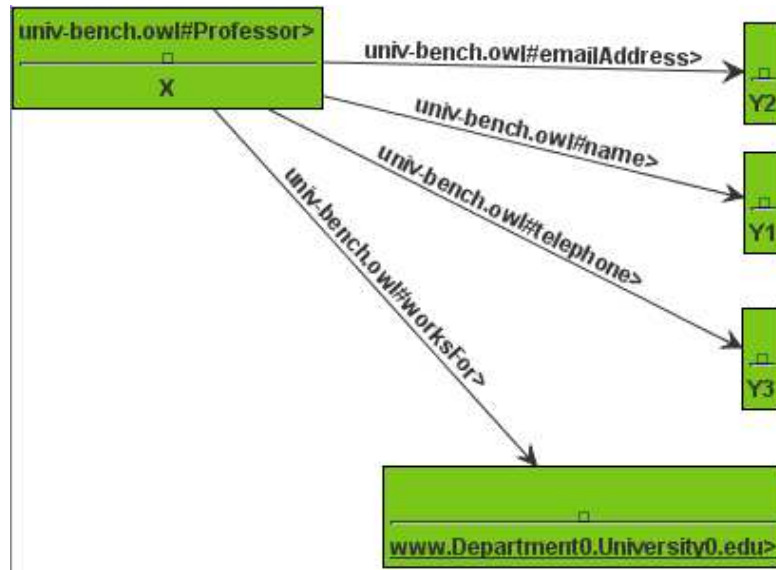


Figure 7: A conjunctive ABox query example

On fig.7 we can see the visual graph elements in action. This is a snap shot from the implementation which was in the process of adopting to the new model so there are numerous bugs, like for example there is no color difference for variable and constant nodes. When we look at the graph we see that object *X* of type *Professor*. The property *worksFor* at the bottom shows that the *Professor X* is working at *www.Department0.University0.edu*. The rest of the properties have as their value a variable meaning that the query is looking for the *Professors name, address and telephone*. So this query finds all the professors from the given university and retrieves their *name, address and telephone*.

3.2 SPARQL-DL Graph Model - mixed ABox, TBox and RBox

As mentioned in the last section the model had to be extended to represent all possible query atoms available in SPARQL-DL. Another extension of the model is to allow variables to be placed in the place of classes and properties so that the query graph model can support mixed ABox, TBox and RBox queries. The first problem that arises when

allowing variables to be placed everywhere is the fact that class type and property terms can be used in more than one node in the model defined in the previous section. There is no problem if the type or the property is not involved in any other query atoms than the ones already defined. However the problem arises when we are using the rest of the query atoms most of which represented with an edge. The arguments of the edge element are defined to be the nodes that the edge connects but one node can have more than one variable class. Another problem that arises with the fact mentioned earlier that we can use one class variable at more than one node. In that case which node we should chose to use to represent the represent the new query atoms? In the case of properties the graphical representation is defined to be an edge. Should the edge we connect edge with edge to represent some of the property query axioms? And as with nodes classes, there are also the problem with the uniqueness of the edge (there can be more than one variable for an edge) and the variable (one variable can be used in more than one edges). The answering of these and other questions lead to a definition of new query model.

First of all it was decided that the type nodes will be represented as a round rectangle nodes. In the case that the node do not contains a type the node representation do not render an extra *OWLClass* type. The new model supports nodes without types. The graphical representation of such node is with two round corners at the bottom and two sharp (normal) corners at the top. This node should look like the bottom part of a normal type node with its upper part removed leaving only the separating line. This way there is a reasonable way to create a distinguishable class node that represents a single class to which class axioms represented by edges will be directed. The graphical representation is acquired by doing the opposite that we did to get classless term node, that is, we remove the bottom part of the type node. The result is a node with two round upper corners and two sharp (normal) corners at the bottom.

Issues with the representation of the property are resolved by changing the graphical representation to one node and two edges. The node contains the property list while the two edges denote the argument and the object of the property with an arrow head at the object side. This way I was able to create a new term node, the property term node that can be used to represent property axioms.

3.2.1 Basic graph elements

The new graph model has four shapes and three colors that are used to create a variety of node types. In addition the individuals are being underlined to be able distinguished it from literals. However the images in the examples are snapshots taken from the program which does not support this feature correctly. In this model the color is used in slightly different way the in the conjunctive ABox query model. In this graph model each label in the node or each line in the node has its own color. The yellow color is used to denote constants that is IRI's and literal. The red color is used to denote distinguished variables. Finally I use a pale color for undistinguished variables. Here is a list with the new types of shapes designed for nodes present in the model:

Round Corners

Down Round Corners

Up Round Corners

Simple Rectangle

The *Round Corners* node is used to denote *Type* and *DirectType* atoms. The node's semantics remains the same, only the shape has changed to a round rectangle. The *Down Round Corners* node is used to denote ABox terms that are used only in query atoms different from *Type* and *DirectType*. This node can have only one label the term that the node represents. This two nodes are the only nodes that can contain undistinguished variables. Also the union of this two node types is the set of all ABox terms (individuals, literals and variables), each of this node which is unique and it is identified by its bottom label. The *Up Round Corners* node is the shape that denotes class terms and TBox or class variables. These nodes are called TBox nodes. A tbox node can have only one element and no other tbox element can contain the same label, term. This node can have only two colors yellow and red because it can be a variable or a class, but it can not be a undistinguished variable [8, 9, 10]. The last shape is the *Simple Rectangle* node. This node is used to represent RBox variables and property terms. The node can have only one label and this label is unique for all the rbox nodes. There are also only two colors used in this node, yellow for IRI's and red for variables. However there are a lot of query atoms as *Transitive* or *Functional* describing the characteristics of a single properties. These query atoms are denoted in the unique RBox node. For now representation is a trailing string containing first one or two letters for the query atoms used on the property. This can representation can be extended to view symbols instead of the syntactic abbreviation used in this model.

The new graph model also introduces new edges. For example the for the subclass or sub property query atom the model uses the UML subclass edge with a triangle arrow head. As mentioned earlier the property node was introduced an so property edge now in the extended model is divide in two parts. The first part begins in the argument of the property and connects to the property node. This edge does not have any arrows or decoration, neither text. The second edge used when representing properties starts at the property node and connects to the value node with a simple arrow.

The last basic graphical element used in the model is a dashed edge with no arrows. This edge is used to represent the query atoms between the individual entities used in the query, that is, classes, properties and objects (variables or individuals). The role the edge is determined by a stereotype. The name of the stereotype matches the name of the query atom that it represents.

3.2.2 Examples

Here are few example screen shots taken from the implementation of the new model. The first query describes is the query in the example in the previous chapter in figure 7conjunctive query On fig.8 we can see the advantages that the extended model over

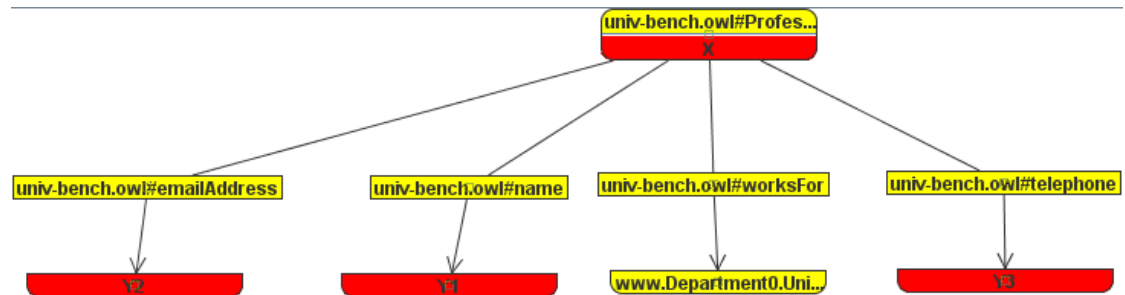


Figure 8: A conjunctive ABox query example presented in the new model

the older ABox model. For example the *Professor* in the root node in the graph is in yellow color meaning that it is a constant, in this case a class IRI. This is more complex than the older model because there we had only one color for each node which was not a problem because the model assumed that there are no variables in the places of classes and properties. Now that the model allows this the nodes should be able to have more colors like in the discussed example. The Fig.?? also is a very good example, showing that the color can very easily help the eye to distinguished which node is a variable and which a constant. And thus we can orient in the query semantics much more easier than when query is in serialized form. However, this example also shows the new model in a so good way. It is fact that the number of nodes increased and thus the complexity of the representation rose.

In the next example I demonstrate a SPARQL-DL query.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX ub: <http://www.lehigh.edu/~zhp2/2004/0401/univ-bench.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
```

```
SELECT ?X ?C
WHERE {
    ?X rdf:type ub:Student .
    ?X rdf:type ?C .
    ?C rdfs:subClassOf ub:Employee .
}
```

In this query we have a mixed ABox and TBox. The variable *X* will be bound to an individual and the variable *C* will be bound to a class that it is also evolved in one TBox query atom. On fig.9 is the a snapshot of the graphical representation model implementation.

Here again it is very easy to distinguished variables from constants. So the first thing we notice looking at the query graph is that we are searching for the variables X and C an individual and a class respectively. The *Type* node on the left shows that there are two query atoms so we know that we are looking for such X that are both of type C and of type *Student*. When we discover a TBox variable in *Type* node it means that there may be some TBox query atoms that that variable is involved with. Looking to the right part in fig. 9 we can see TBox node with the same label C as one in the middle in the *Type* node. This means that this node TBox node represents node that is involved in the description or the constraints of the variable. The label in the *Type* node only represent a term tha is used by the *Type* query atom. So the query atom in which the variable C is involved is *SubClassOf*. So next we look at the last node, which is a constant TBox node, also involved in the *SubClassOf* query atom as the super class. So this query looks for students that are also employees and what type of employees they are. This query shows some limitations of the model. First the variable C is located in two places one where the variable is defined and constrained and another zero or more places where it is used as argument in a *Type* query atom. This separation can be misleading if the viewer is not well informed about the model.

3.2.3 Model limitations and proposed improvements

There were several unwanted features of the model that were observed in the examples. First, the increase of the number of nodes, due to the introduction of the property node. Another aspect of the model, also increases the number of nodes in the graph. This is when the graph represents the TBox and RBox query atoms , there is an additional node inserted that is used to for the description of itself and other TBox or ABox variable nodes. However the second reason for the increase of nodes is not that bad, after all we add only one node per term. This approach can be used to implement the programmatic model that stores all the necessary information. Apart from this the other problem with Rbox and Tbox nodes is that it is not graphically emphasized that for example in fig.9 that variable C in the *Type* node is actually the same thing as the TBox node on the right. For a person that have used the model few times that won't be a problem. An-

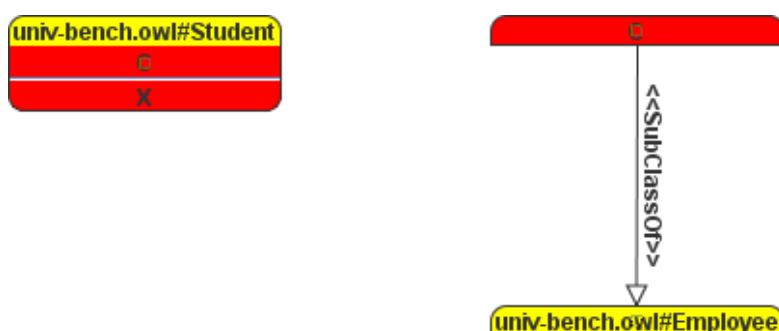


Figure 9: Example of a mixed ABox TBox SPARQL-DL query

other problem is that the model does not handle the separation of ABox and RBox, TBox query atoms. This may be hard to find the TBox node in a bigger query.// So there are three approaches that can be assumed for now that solve the issues discussed above. First, concerning the property nodes which are creating too much detail. Of course the shape of the property node suggests that there is something that is very similar to other elements in the graph such as *Type* nodes. I think the model should keep the property nodes because this way they are consistent in their shape as the RBox nodes. The problem may be solved if the property node and its edges that connect it to its argument and value can be viewed as one edge. By this I mean that when there is a node between the edges the node can move around and so the edge move as well, thus creating the impression of that the property node is independent of its argument and value. So I need to make the property node non movable. Its position will be determined from the edge that connects the argument and the value. To create even better impression of that the property node is part of the edge, I can rotate the node so that it is aligned with the edge, as it was before with the conjunctive ABox graph model.

The next issue was the separation of the TBox or RBox description of a variable and then using it in a *Type* or *PropertyValue* query atoms. I will discuss the solution with respect to the TBox variables first and then I outline specifics in the case of the RBox variables. This issue can be resolved very elegantly when there is only one use of a TBox variable in the *Type* query atom. In that case I can substitute the TBox node with the Node that uses the variable. This way all the extra TBox nodes will disappear and the model will become liter. However this approach is very limited because if for example the *Type* node has more than one class variables or classes IRIs that are used in other TBox atoms there must be a way to distinguish which edge corresponds to which term in the type list of the node. Another obstacle is the fact that it may quite often that one variable is used in more than one *Type* nodes. So combining all those cases it may be optimal if this functionality is optional. So there might be a menu item that will switch between the mode where the TBox nodes are independent and the other mode, where the TBox node is incorporated in to the *Type* node. To emphasize which to which variable in the node the edge is actually connecting we can make a constraint on the connection point on the node, for example the edge can connect only on the row where the term to which it connects is located. At last the problem with multiple places where the TBox can be incorporated, we can determine a candidate and then let the user decide and give him the ability to choose where to place the TBox node. In the case of RBox, there is the only difference that the property can have its characteristics as *Transitivity* or *Symmetry*. This means that the property node should change its representation to be able to show this characteristic when one tries to incorporate the RBox in the property node. Everything else stated for the TBox is also true for RBox nodes.

The last issue discussed here is to be able to distinguish a TBox atoms the way variables are distinguished using the color. We can separate the plain of the graph in to two. One part of the plain will be used only for ABox and the other part of the plain can be used for both TBox and RBox. What about mixed elements such as the *Type* node in fig.9 on the right. One approach is to order the type nodes' and property

nodes' labels so that all the TBox variables are on top and all the class IRIs are at the bottom of the node of course above the individual. This way mixed ABox, TBox and RBox nodes can be aligned on a horizontal line so that above the line there is TBox and RBox expressions and below that line there are ABox expressions. This approach is in small conflict with the solution of the first issue. This approach has other obstacles that concerns the layout and how edge will be placed but it is a very attractive idea that should be at least prototyped and tested.

3.3 The graph editor

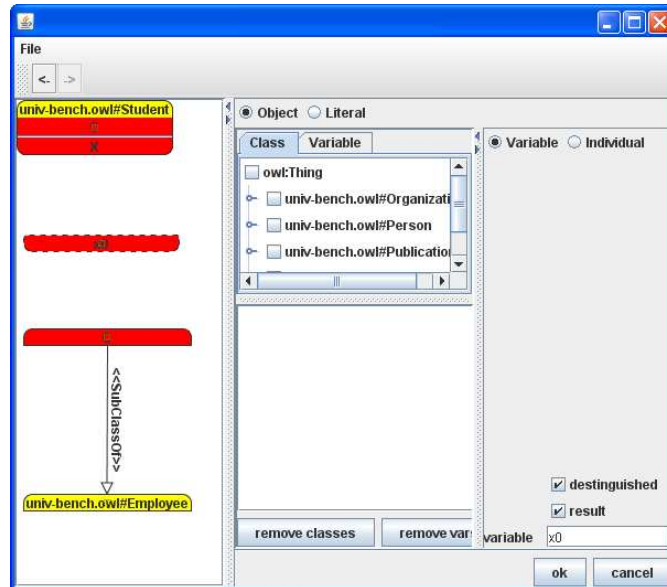


Figure 10: Screen shot of the query editor designer after creating an ABox node or a Type node

The Graph editor is designed in two main parts, the graph pane and the editor panel. On fig.?? edits a screenshot example of the prototype implementation of the editor. The graph pane is located on the left and it is used to view the graph and capture mouse and key events that trigger editing actions like adding a node removing or deleting a node or a multiple selection of nodes, creating edges . . . The graph pane is only limited to insert and remove edits. The editor panel on the right is used to edit the nodes and edges contents, that is the classes included in the type node whether the variable is of direct type or not . . . All the edits in the editor are being stored in the undo manager and can be undone using the action buttons on the up left corner under the *file* menu. On the fig.10 we can see part of the class subsumtion witch is useful for finding types more efficiently. Another helpful feature of the editor is the Individual editor panel is interconnected with the class view and is filtered by selecting a class in the class view. There are two lists in the Individual editor to list that is on top is the list of individuals

that are part of the class selected on the class tree. The bottom list consists of all the individuals from the ontology. Both lists are sorted alphabetically. This is visualized on the fig.11. There is also possible to see that the selection of the two lists is synchronized. The other editor panels responsible for editing the TBox, RBox and Property nodes and

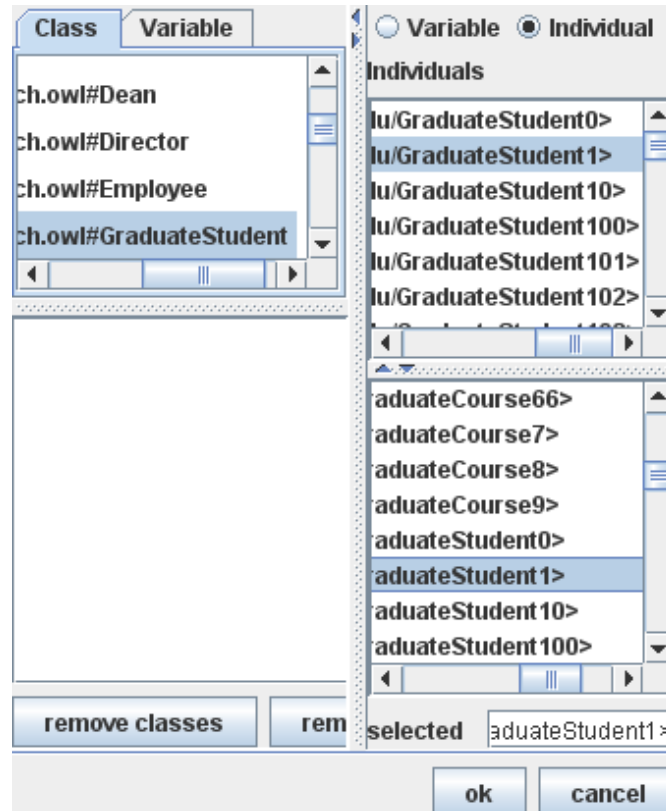


Figure 11: Screen shot of the editor panel

the editors for the dashed edges are not yet implemented. This is because the change to the new model started recently and there were major changes in the whole program code. The editor panels will be implemented and functional in the near future.

4 Query Editor Implementation

The implementation of the editor is based on several libraries. These are JGrapph + JGraphaddons, OWL2Query, OWLAPI, Pellet and libraries that come with it. In the next chapter I will describe the functionality and usage of all of the libraries listed here. Then I will start with description of the implementation itself. First I will start describing the main components of which the program consists. I will discuss the implementation of each of the components over a simplified UML class diagram. After words I will try to explain some of the reasons why I've chosen this implementation. Then

I will shortly talk about the integration of the software in other environments as web applications a developer tools such as Protege. At the a end I will conclude the state of the implementation and how the project might proceed into development.

4.1 Used Technologies

4.1.1 OWL2Query

OWL2Query is a programmatic data query model for SPARQL-DL. The tool is very useful because it provides a complete interface for all the processes that are involved in reading a query and running it into an inference engine. The two most used interfaces in this library are the OWL2Query and OWL2Ontology. The way this library is use is pretty straight forward. The first step is to create an OWL2Ontology. There are several ways to do so. There is the possibility to choose between three possible ways. Using owlapi, Pellet or factplusplus. In my case I am using Pellet with OWLAPI. The pellet is a reasoner that implements a tableau algorithm [1].When doing things through OWLAPI one can also incorporate a reasoner such as Pellet that implements OWLAPI's reasoner interface. Here is a sample code for creating the OWL2Ontology interface with an incorporated pellet reasoner:

```
OWLReasoner r = null;
OWLAPILoader oal = null;
OWLAPIv3OWL2Ontology o = null;
oal = new OWLAPILoader();
oal.setIgnoreImports(false);
oal.parse(filename);
oal.load();
oal.getKB().realize();
PelletReasoner rr = oal.getReasoner();
rr.getKB().printClassTree();
rr.prepareReasoner();
o = new OWLAPIv3OWL2Ontology(oal.getManager(), oal.getOntology(), rr);
```

Here I use OWLAPILoader to load the owlapi ontology and manger, then I prepare the reasoner and the knowledge base. At the end I instantiate an instance of the OWL2Ontology with its implementing class OWLAPIv3OWL2Ontology.

After creating the OWL2Ontology object depends what the user wants to d, to load a query from a file or to create a new query using the editor. The program uses the services of OWL2Ontology in two ways. The firs way is while editing is in progress. The editor has a reference to an object that contains the OWL2Ontology and creates an adapter for the rest of the application. The class name of this object is QuerySymbolDomain. It delegates tasks such as asking for the subclasses of a particular class to the underling OWL2Ontology implementation to. The other way I am using this library is when I read a SPARQL query file or when I want to run the query. In those cases there are two major steps that are performed. When reading a query from a file, I first load the query using the ARQParser to parse the query file using the already loaded OWL2Ontology.

```
SparqlARQParser p = new SparqlARQParser();
query = p.parse(new FileInputStream(f.getSelectedFile()), ontology);
```

The next step is to convert the OWL2Query model into the internal query graph model. The reason why I convert the model to another internal model is because it is more comfortable to use my own model which is used for visualizing the graph and editing at the same time. Another very strong reason to do so is that my model is serializable. And yet third reason is that changing the OWL2Query is not entirely a public API. That's why I created a convertor class in the package containing the package scoped OWL2query API responsible for creating and editing new queries programmatically. When the user is done with the design of the query, one may choose to execute the query then the reverse conversion from my internal model to OWL2query. Afterwards it is straight forward to execute the query. Here is a code example:

```
QueryResult result = OWL2QueryEngine.exec(query);
```

The result is a collection or iterator of query bindings.

4.1.2 JGraph + jgraphaddons

JGraph is an open source that implements a graph component compatible with swing. The component is less or more uses some has something in common with the JTree component in swing but its flexibility is impressive. The JGraph is based on the model, view, renderer pattern. It has a graph model it has model for the cells used in the graph that are compatible with that model. The view of each cell is caring all the necessary information for the cell to be viewed, including the renderer who is the one that actually draws the cell using the visual parameters stored in the view. The work type of work that is involved with JGraph is almost the same as with any customizable JComponent. The JGraph should be configured with its standard parameters for example that allow the editing of cells their resizability the background of the graph and so on. To make JGraph visualize different shapes one has to implement a install renderer or using the suggested way written in examples in the documentation of JGraph to over write the cell view factory. The implementation of the renderer or the cell view is not done by returning the renderer component as with the rest of swing components. In JGraph one has to take into account the fact that the edges might connect to the node. If the shape of the node is different than the connecting edges will not connect correctly on the border of the shape. This is resolved by implementing the getPerimeterPoit() which has as an input the cell that we have to calculate the connection point, and two points that represent the direction of the edge that is going to connect to the cell. In this project JGraph along with JGraphaddons which is a library for layout of the graph. The library is outdated and is not compatible with the more new version of JGraph. The compatibility issues are that not all of the jgraphaddons implemented algorithms for layout are save to run with the newer version of JGraph. The reason why I do not use a newer version of JGraphaddons is because its support as a open source layout library was ended. Now days JGraph Layout Pro is available as the commercial up to date version.

4.2 The Query Graph Model

In this section I describe the model used to represent SPARQL-DL queries and how it is used in the program. On fig.12 is the simplified class diagram of the model. On the left there is one single interface *IQueryGraph* that represents the query itself. This interface defines the methods for accessing the query graphs data which is distributed within the *IGraphElements* hierarchy which is the basic interface for all query graph elements. The main role of this interface is to provide access methods to the query of type *IQueryGraph*. This is done so that graph element knows to which query belongs. In my case this is mainly used to message any changes in the graph element to the query that it is contained in. This way I can refresh the *JGraph* component whenever a graph element has changed some of its properties due to an edit that occurred. So that's about the *IGraphElement*. The next two interfaces that extends the *IGraphElement*, are the *INamedElement* and *IEdge*. The name of the interface *INamedElement* tells that this graph element has a name, it is identifiable. The identifier is of type *Term* which represents any term that can occurred into the query graph. This identifier term is called common term. The interface is used for elements that are identifiable with one term. Such graph elements happen to be nodes. The *INamedInterface* defines methods for accessing the term that identifies the graph element. There is also methods that informative methods that are accessing information about the term, for example there are the getter and setter methods for the *distinguished* and *result* properties of the graph elements common term or identifier. These methods should return true only if the implementation of the interface has a reference to common term which is variable then the graph element can decide on an internal state whether it will return true or falls for any of these two properties. The other interface that implements the *IGraphElement* is the *IEdge*. The *IEdge* interface has extends the base interface by adding four accessor methods and one test method:

```
INamedElement getArguments();
INamedElement getValue();
void setArgument(INamedElement arg);
void setValue(INamedElement value);
boolean equalsEdge(Term arg, Term val);
```

These methods are tailed for the needs of a property or other edge elements. The getters and the setter methods give access the argument and the value of the edge. The *INamedElement* is used as the type of these properties. This is done this way because it is allowed to be only one edge of a particular type between the same two nodes. Another useful feature is that if one changes the argument's term or other properties of the graph element, the edge does not loses a reference to its arguments. This is also used in the *jgraph* component so that the *jgraph* knows between which nodes an edge should connect. This works because the nodes are assigned a value of the *INamedInterface*. The fifth method in the interface is used to compare whether this edge is connected between the two supplied terms. This is useful for example when converting a *OWL2Query*

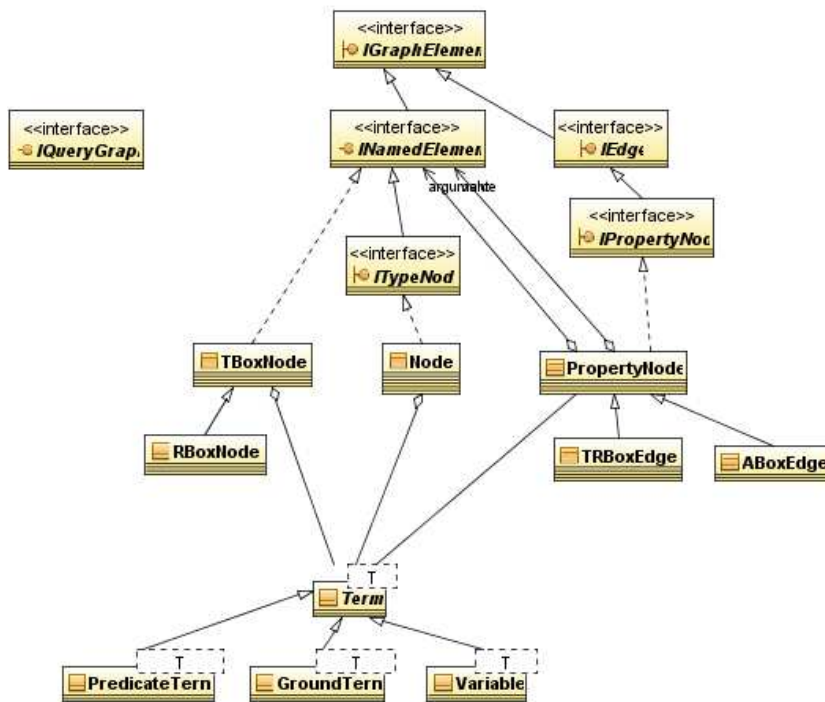


Figure 12: Simplified UML class diagram of the Query Graph model

IQuery.

The next two extensions to the next two interface not shown because it is not important. The only important thing that misses in this UML class diagram is the ICompoundAssertionElement interface. The name is not correctly used. It is better to call the ICompoundTermElement interface. As the first word suggests that it is an interface that supports accessor methods for a collection that is represented by this graph element. This interface is useful because some of the graph elements can have more terms as the arguments of the query atoms of the query atom type that the element type of the element represents. Here are the methods defined by the ICompoundAssertionElement interface:

```

boolean removeAll(Collection<? > c);
boolean remove(Term t);
boolean add(Term t);
boolean addAll(Collection<?extendsTerm c);
List<?extendsTerm getList();

```

This methods are used for example by the override JGraph cell renderer to acquire the elements in the list or the list itself. This interface is extended by another two interfaces which the base type in the model for representing the type node and the property

node. These interfaces are `ITypeNode` and `IPropertyNode`. The `ITypeNode` in addition to the `ICompoundAssertionElement` it extends the also `INamedElement` interface. That's what we want for the type node one element identifying the node in the set of all type nodes and having the ability to store multiple elements for the classes or the data type of the object that is being typed. On the other hand the `IPropertyNode` extends the `IEdge` interface, thus becoming a type suitable for representing property nodes. The next step into the structure of the model is implementation of the `IPropertyNode` by the class `PropertyNode`. The class is shown on fig.12 where there are two aggregation relations to the `INamedProperty`, which actually the argument and value aggregations in the property node. The property node is then extended to `TRBoxEdge` and `ABoxEdge`. These edges represent query atoms like `SubClassOf` for the `TRBoxEdge` or `SameAs` for `ABoxEdge`. Since they extend the `PropertyNode` class they also have aggregated argument and value of type `INamedElement`. The difference between the property nodes and its subclasses is that they keep a collection of `PredicateTerms` which are used to represent `TBox` and `RBox` query atoms.

Moving to the left on the fig.?? we can see the implementation of `ITypeNode` interface. the `Node` class. This contains the identifier term, the common term and a collection of class terms. To represent different the query atoms `Type` and `DirectType` the `Node` uses a set in which contained in the terms of the types that are selected to be `DeirectType`. The accessor methods are defined in the `ITypeNode` interface. There are two more elements to be covered by java classes and these are `TBoxNode` and `RBoxNode`. The `TBoxNode` class implements directly the `INamedInterface` and do not extend the `ICompoundAssertion` interface because this node do not represent any query atom but a single term used in `TBox` axioms. There is separate `RBoxNode` class because needs to store information for the description of the properties, that is for example *Transitive ObjectProperty* ...

All of the graph elements have been covered with a java class type that represents the state of the graph element during the design of the query. Another important element is the `Term` abstract class and its implementations `Variable`, `GroundTerm` and `PredicateTerm`. These are wrapper objects that store the IRIs the variable names or the predicate terms. They are used to represent any term that occurs in the query graph and its query graph elements.

Next I will explain how the `QueryGraph` class, implementation of the `IQueryGraph` interface, handles graph elements. The `QueryGraph` class contains three hash maps that map the three identifiable types of nodes, that is, the `Node`, `TBoxNode` and `RBoxNode` classes. Hash maps to keep the graph consistent. This is done by checking before doing an edit whether there is already such node or whether the property that the user tries to add is already there. `QueryGraph` class stores property nodes `ABoxEdge` and `TRBoxEdge` elements in sets.

In addition to the visual graph model described in the previous chapter this implementation tries to facilitate and represent a structure that would be useful when the extension about the incorporated `TBox` and `RBox` in `ABox` and `Property` nodes respectively. This is accomplished by a list of `IGraphElements` contained in the `TBoxNode` and `RBoxNode`

classes and a single IGraphElement variable. In the future this would be used as follows. I am going to explain this first only about TBoxNode class and then I will point out specifics for RBoxNode. The list in the TBoxNode will contain all the type nodes that are using this concrete TBoxNode's common term in their type list. This way the TBoxNode will have to track its references. The single variable of IGraphElement would be used to store the current node's that is chosen to serve as the source and target for all TBox query atoms. There are actually no differences in the case of the RBoxNode class, except for that it will store PropertyNode elements in the list and the single IGraphElement variable will be of the same type.

This is the current implementation of the programmatic query graph model. It stores the needed information to be able to store correctly the state of the edited query.

4.3 The graph Editor implementation

4.3.1 The undo support and the query models bad design characteristics

First I will start with the basic components and models that I created to use later to incorporate into the panel editor. Firstly I implemented a JTree cell renderer to as check box. This was seems a good way of how the selection looks like how it behaves. Here in fig.?? is an example screenshot of the older version of the editor. Here I can demonstrate the functionality of the editor panel that I was after. Apart from the check box selection, which will initialize with the current nodes type list, there is another feature that shades some of the nodes of the tree. In this example the shaded nodes are the super classes of the most specific class selected. In this version there is a small bug that shades also the most specific type. This small frame work can be used to show useful sets in the tree to ease further the query development.

In the next fig.13 we see the JTree element was initialized with the nodes class list. Notice that the class list selection on the right has its members also shaded. This way one can instantly see whether he chose a type that it is actually useless to choose. There are other ideas around the list and the representing the same selection, I mean normal mouse selection not the selection denoted by the checkboxes and the list on the right. Fig.13 is illustrating also the synchronization of the list and the tree. In this example the user clicked on a class in the selected classes in the list which was not visible in the JTree component. This event triggered an action to find the class that was acted upon and to localize it in the JTree component so that it will be able to scroll to it show change the selection to it. The change of the selection happens also in the normal case when the node is visible, but that's not an very interesting feature, but at least it can remind users that this feature exists or it can interest beginners and they may learn its use without looking in the help. Another useful feature of the list is that when you double click it the class was under the mouse pointer will be removed from the list and of course the removal will be also reflected in the tree. This is useful when we want to delete some of the classes that we selected and it is much quicker looking the class in the tree.

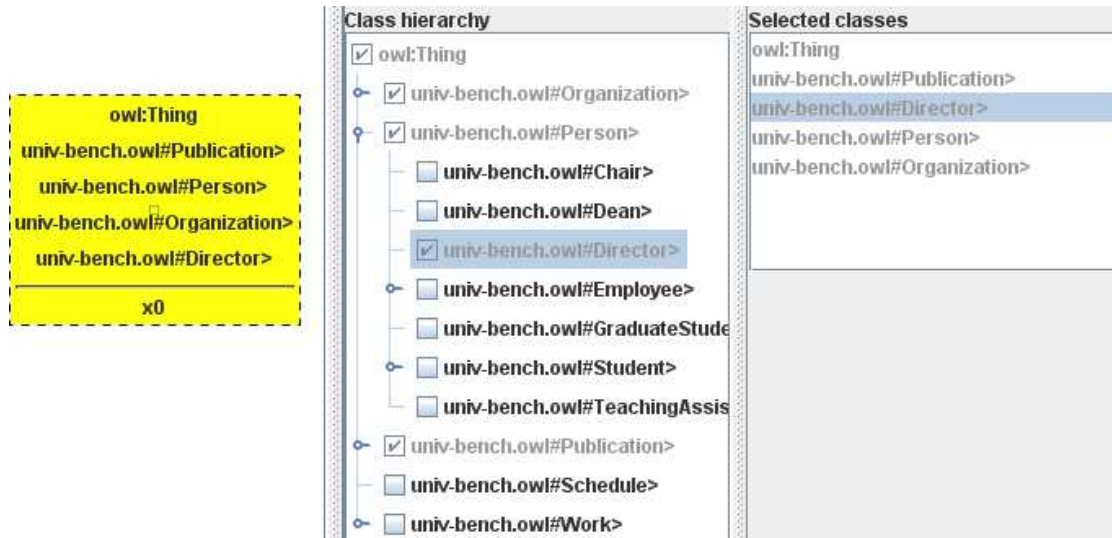


Figure 13: A screen shot showing the initialized selection of the JTree and the selected classes list

This behavior is going to be used also in the new editor panel. The migration from the old the new version of the model and the editor is going to be not so hard because this functionalities are build in way that they can be reused.

4.4 Upgrading to the new Visual Graph Model

In the moment the query designer went through quite a few major changes in the code. For example the extension of the programmatic query graph model lead to some issues about the implementation of the model. For example before the QueryGraph class kept the list of all used variables in the query. It also had a set of all distinguished variables and a set of all result variables. This was one of the reasons why the undoable edits where not implemented very practically. After an edit was applied in the panel editor the panel editor ran a remarkably long function that had the goal to find the differences between the edited element and the editor form. After each property and aspect in which the element and form can be different a new undoable edit was generated which was then added to the list of the compound edit that was going to represent the change. However this approach was impossible in the case of the new query model because there will be so many things that each editor should be able to figure out on its own. That's why I thought to make much more reliable undoable edits. The idea is based on the simple two functions that are present in the IGraphElement interface. Those two functions were:

```
public void copyTo(IGraphElement n);
public boolean compareTo(IGraphElement n);
```

Those two functions will do two relatively simple for the model tasks. The first will be able to copy its own contents in to a new node so that the two nodes will be identical with

regards to terms and properties. The second function will compare with the input node and it will return true if the nodes are the same and false if there is even the smallest difference for example the variable has changed from distinguished to undistinguished. The implementation of this functionality was quite straight forward because the model is built on few layers. This mean that the comparison of the Node class took care only of the *common term*, the *result* and *undistinguished* properties and the rest was left to be done by the super class, in this case that is CompoundAssertionElement. Here is how the code looks like for the comparison of the two Node objects.

```
public boolean compareTo(IGraphElement n) {
    if(!super.compareTo(n) || !(n instanceof ITypeNode))return false;
    ITypeNode tn = (ITypeNode)n;
    if(tn.directSize() != direct.size())return false;

    for(Term t : direct){
        if(!((ITypeNode)n).isDirectType(t)){
            return false;
        }
    }

    return equals(tn) &&
        !((isDistinguished() ^ tn.isDistinguished()) ||
        (isResultVariable() ^ tn.isResultVariable()));
}
```

If the super class succeeded in the comparison, than the Node's task is to check the elements that he is responsible for. In this code first I check whether the number of *DirectType* classes the same. If it is different it means that there is at least one different element in both *direc sets* so if that is the case return false. Otherwise check if one of the elements of our set is not contained in the direct set in the input Node object. If there was a difference in the set return false. Otherwise true if all of the three variables: the common term, the distinguished boolean variable and the result boolean variable. If there at least one difference return false else return true. Next I will show the code for the compareTo method in the CompoundAssertionElement which the super class of the of the class Node

```
public boolean compareTo(IGraphElement n) {
    if(!super.compareTo(n) || !(n instanceof ICompoundAssertionElement))
        return false;

    List<? extends Term> list = ((ICompoundAssertionElement)n).getList();
    if(list.size() != predicates.size())return false;

    for(Term t : ((ICompoundAssertionElement)n).getList()){
        if(!predicates.contains(t))
```

```

        return false;
    }

    return true;
}

```

Here we are checking for the equivalence in the class sets. First we check whether the super class finds something wrong if this is the case than the return false. The super class is actually the implementation of the GraphElement class that actually does nothing else than checking whether the input node has the same query. When we are back and the first if statement succeeds then I check the list size and the a check whether every element our list is contained in the inputs node graph element. If lists are the same return true else return false.

This is how easily is implemented also in the other few classes of the model. The code for the copyTo method is almost the same when we are talking about the length a the structure of the code.

The whole idea with the new undoable edit was that create two helper graph elements after an edit was applied. This means that I have to check the whether the form of the editor panel is correctly filed and then to fill one of these helper graph elements with data from the form. This would be new elements state. The other than I will compare using the original graph element and will call the compare method with its input the new state. If the two nodes are the same the method will return true and this means that there is no need of an edit. In the case when compareTo method returns false, than we call the original elements copyTo method with the other state which I call the old state. After I have an old state and the new state of the element and I have the element that is being edited I can always call copyTo from the new state with argument the original element, which will result in the change of the original node change its state to its new state. Then I can do the same with the old state and original element. The result will be that the original element will go back to its former state. This is how it is possible to implement undoable edits. However for this to work as easy as it looks, the condition was to move all the data about the graph elements inside the graph elements themselves. Keeping the result variables and distinguished variables in the query itself was causing some serious issues with the graphs model consistency. This also involved rewriting owl API method calls with the new ones. In some cases I was even pushed to make minor changes the API of the render classes.

This way the editor panel code cleaned dramatically and was only about keeping the form synchronized to the node and the user actions. The next important task of the editor panel is to validate the form.

4.4.1 The renderers

Simultaneously with the change of the model and the API of the undoable edits I needed to be able to render the shapes defined by the model specification. However the problem was not only in the shapes of the nodes but also the way I was rendering nodes. Before

I was using the fact that, swing has the ability to render simple HTML code. In the last version of the designer I was using HTML to render the nodes. This was also an alternative but it is quite disturbing that one uses java and inside the java code he generates HTML code to render something in the java program. So I decided that this has to go as well. For illustration here is the code that I was using to render the HTML code:

```
public String nodeValueToString(ITypeNode n) {
    String body = "";
    Collection<Term> predicates = n.getTypes();
    for (Term t : n.getTypes()) {
        if(t.isGround()){
            body += "<tr><td align=\"center\">" +
StringRenderer.owl2String(t.asGroundTerm().getWrappedObject()) +
"</td></tr>";
        }else{
            body += "<tr><td align=\"center\">" +
StringRenderer.owl2String(t.asVariable().getName()) +
"</td></tr>";
        }
    }
    if (predicates.size() == 0) {
        body += "<tr><td></td></tr>";
    }
    body = "<table rules=groups><tbody>" +
        body.substring(0, (body.length() - 2 < 0) ? 0 :
        body.length() - 2) + "</tbody>";

    body = body + "<tr><td align=\"center\"><hr size=1>";
    if (n.isVariable()) {
        body = body + n.getCommonTerm() + "</td></tr></table>";
    } else {
        body = body + "<u>" + n.getCommonTerm() +
        "</u></td></tr></table>";
    }

    return "<html><body>" + body + "</body></html>";
}
```

As I mentioned earlier the two important things that one have to take into account when creating a JGraph node cell renderer. The first is to create the renderer component and then to implement the getPerimeterPoint method. I didn't wanted to create a renderer for each node which is present on the models specification. Not that their too much but I wasn't sure about how the multiple renderers should be installed on the JGraph component. So I decided create one custom renderer in which I will build the

logic of how each node is rendered. Apart from the shape of the nodes there was also a challenge with the contents of the node. In the end the whole API for the renderer had the following components, a `NodeBorder` and a few renderer classes that were combined to form the desired result. I decided to implement a custom border class `NodeBorder` so that I can plug the border easily to a renderer class. The way that the border class works is that the border class implements some methods that compute intersections of two lines or a line and a circle. Using this I created parameterized functions that draw a border of the four shapes from the specification based on the parameter. Then used this class and delegated these methods the actual renderers methods for drawing a border and finding the intersection point of the border and the connecting edge. So it was already quite easy to give the desired border to any cell renderer.

The next step was to create the renderer insights of the node. I accomplished this by implementing three simple renderers which had a common abstract class `SubRenderer<T>`. The abstract method to be implemented in subclasses is:

```
public abstract void includeInContainer(Container cont, T element);
```

The idea was to have `ContainerRenderer` in which I will put the `SubRenderer` implementations. The `ContainerRenderer` will call the `SubRenderer`'s `includeInContainer` method with a `JPanel`. Here the classes that implement the `SubRenderer<T>`

```
class ListRenderer extends SubRenderer<ICompoundAssertionElement>;  
class IDRenderer extends SubRenderer<INamedElement>;
```

Having this classes implemented, the code for creating the property node renderer looks like this:

```
public class PropertyRenderer extends ContainerRenderer{  
    protected SubRenderer.ListRenderer rendrer = new SubRenderer.ListRenderer();  
  
    public PropertyRenderer() {  
        border = new NodeBorder.BorderRect();  
    }  
  
    @Override  
    protected void buildComponent(IGraphElement elemetn) {  
        rendrer.includeInContainer(container, (ICompoundAssertionElement)elemetn);  
    }  
}
```

5 The Query Designer in Practice

Once the new model and API is finished the query designer will be very practical tool. Along with the development of the query tool I was involved in a project where we were

trying to integrate the designer in to a web application. There the designer gain a also new dimension because that was the first time I changed the model so the QueryGrpah an all of its elements including terms is serializable. It was a success to establish the communication with the glassfish application server and to communicate the query model as well as all the inference that was carried on the server and then the required data was send in the form of sets of IRIs.

Another possible application is to make the designer as a plug-in for Protege, where it will be quite useful in ontology development. For now Protege has a very simple ABox DL query tool which will be outpreformed by this one.

6 Conclusion

The project is undergoing major code changes. One of the reasons for this is the upgrade of the graph model from one that can support conjunctive ABax to a more general model that supports current versions of SPARQL-DL language. From the beginning I was trying to make the code modular and reusable but eventually the program was ended in a state where upgrade was needed to change major bad design approaches. These are concerning the graph model implementation, editor panel design and tasks and graph visualization using JGraph. On the other hand the old implementation contains a lot of useful and reusable code in the form of GUI elements and classes which will be used to rebuild the new GUI.

Currently the program is capable of loading queries from a SPARQL query file and visualizing the query graph. Graph interaction is implemented so in the moment one can insert nodes and edges using click and drag mouse gestures, however the connection with the underlying query model is not correct and there are some bugs to be fixed. Panel editors are not yet designed except for the type node panel editor which is already designed but still not integrated in the GUI correctly. Integration of the new panel editor was possible rebuilding the framework for integration was necessary because of the major changes made in the code. This editor is actually the most complex one. The design editors of other panel editors will be simpler and faster also because the framework for integration will be more stable and the approach will be closely defined.

From the point view of functionality the program is capable only of visualizing the query graph of SPARQL-DL queries stored as SPARQL files. However the underling changes are creating a more stable and reusable basis for integrating it in other applications such as web applications or developer tools, also making it more upgradable and manageable. Next I will list the program modules and their state. The data model for SPARQL-DL queries is implemented and functioning. The undo support was simplified and an easy to use API was designed. Panel editor tasks were reduced and the code in GUI elements that follow the new approach much more readable and manageable. JGraph rendering was introduced in the new version that works almost without problems. Graph interaction were encapsulated in a separate module that can be extended.

A functional version of the application is going to be available soon. However there are already suggestions for the upgrade of the model. These suggestions will be easy to implement in the new version of the query tool because the code was designed so that such type of upgrades will be possible with minor changes.

References

- [1] F. Baader and U. Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.
- [2] Jeremy J. Carroll and Graham Klyne. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, February 2004. <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>.
- [3] Richard Cygnaniak. A relational algebra for SPARQL. Technical report, Hewlett-Packard Development Company, L.P.
- [4] Marie Demlová and Bedřich Pondělíček. *Matematická logika*. CVUT, 1999.
- [5] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible sroiq. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2006)*, pages 57–67. 10th International Conference on Principles of Knowledge Representation and Reasoning, AAAI Press, June 2006.
- [6] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705, pages 161–180. Springer-Verlag, 1999.
- [7] P Kremen and E Sirin. Sparql-dl implementation experience. In *OWL: Experiences and Directions (OWLED-2008)*. Washington: OWL Working Group, 2008, 2008.
- [8] Boris Motik, Bijan Parsia, and Peter F. Patel-Schneider. OWL 2 web ontology language structural specification and functional-style syntax. W3C recommendation, W3C, October 2009. <http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/>.
- [9] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. <http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>.
- [10] E Sirin and B Parsia. Sparql-dl: Sparql query for owl-dl. *3rd OWL Experiences and Directions Workshop (OWLED-2007)*, 2007.
- [11] Evren Sirin and Bijan Parsia. Optimizations for answering conjunctive ABox queries. In *Description Logics*, 2006.