Facoltà di Ingegneria Informatica
SAPIENZA – Università di Roma

Tesina per il corso di:

Seminari di ingegneria del software
A.A 2007/2008

docente: **Giuseppe De Giacomo**

autore: **Tassi Carlo**

Indice generale

1 Introduzione

Obiettivo della presente tesina è realizzare in *DL- Lite* le *ontologie* corrispondenti ai diagrammi UML delle classi presi dagli esami del corso di Progettazione del software I dell'anno 2008 e realizzare al posto dei programmi richiesti per gli USE CASE delle analoghe (per quanto possibile) *query congiuntive*.

A tale scopo verranno prima illustrati i concetti di ontologia e la sua applicazione, i linguaggi per le ontologie e la sintassi funzionale, e successivamente verranno illustrati i meccanismi di traduzione dei componenti di un diagramma delle classi affrontando passo passo le diverse problematiche che possono venire a crearsi e successivamente applicando questi meccanismi ai casi di studio.

Le ontologie verranno tradotte quindi in sintassi funzionale per DL-Lite per essere passate al toolkit QuOnto col quale verranno poi valutate le query congiuntive che esprimono i programmi richiesti per gli USE CASE.

2 Ontologie e linguaggi

2.1 Ontologie

*"An **ontology** in computer science and information science is a formal representation of a set of concepts within a domain and the relationships between those concepts. It is used to reason about the properties of that domain, and may be used to define the domain."* (da http://en.wikipedia.org).

Le ontologie mediano l'accesso ai dati rendendo trasparente l'accesso ad essi, infatti nascondono all'utente come e dove sono memorizzati, fornisce una vista concettualizzata di essi ed usa un formalismo semanticamente ricco. É un'impostazione molto simile a quella dell'Integrazione dei dati.

Altra caratteristica fondamentale è che qui le informazioni possono essere incomplete o mancanti e per questo necessitano di strumenti di ragionamento automatico. É questo l'aspetto che le differenzia dagli altri strumenti di rappresentazione quali diagrammi UML delle classi e diagrammi ER per le basi di dati.

2.2 Linguaggi per le ontologie

Un linguaggio per le ontologie è un linguaggio formale usato per codificare le ontologie. Sono basati su First-order Logic oppure su Description Logic.

Ne esistono numerosi sia proprietari che standard based.

Disntinguiamo inoltre tra linguaggi tradizionali e linguaggi di markup.

Tra quelli tradizionali possiamo nominare: *CycL*, *F-Logic*, KIF (*Knowledge Interchange Format*) , *KM programming language*, OCML (*Operational Conceptual Modelling Language*), OKBC (*Open Knowledge Base Connectivity*), ecc...

Tra quelli di markup ci sono invece: DAML+OIL, OIL (*Ontology Inference Layer*), OWL (*Web Ontology Language*), RDF (*Resource Description Framework*), ecc...

In particolare partiamo dal linguaggio OWL per poi arrivare a DL-Lite in quanto trattasi di un linguaggio Description Logic-based.

2.2.1 OWL

Il Web Ontology Language (OWL) è una famiglia di linguaggi della rappresentazione della conoscenza (AI) per la definizione di ontologie, ed è approvato dal World Wide Web Consortium.

La W3C ha specificato tre varianti di linguaggio OWL basate su due tipi di semantiche con differenti livelli di espressività: *OWL DL* e *OWL Lite* sono basate sulle *Description Logics* che hanno proprietà computazionali ben ottimizzate, mentre *OWL Full* usa un nuovo modello semantico che offre compatibilità con gli schemi *RDF*.

In particolare abbiamo:

- **OWL Lite**: il più semplice infatti non molto espressivo, non permette di esprimere vincoli di cardinalità diversi da 0 e 1. In realtà molti costrutti possono essere però espressi tramite complesse combinazioni.
- **OWL DL**: è stato progettato per fornire la massima espressività possibile, pur mantenendo la completezza computazionale e la decidibilità. Esso include tutti i costrutti del linguaggio OWL, ma possono essere usati solo sotto certe restrizioni.
- **OWL Full**: si basa su una semantica diversa da OWL Lite o OWL DL, ed è stato progettato per mantenere la compatibilità con alcuni RDF schema.

Ciascuno di questi sottolinguaggi è un'estensione sintattica del suo predecessore più semplice. Valgono le seguenti relazioni:

> Every legal OWL Lite ontology is a legal OWL DL ontology.
> Every legal OWL DL ontology is a legal OWL Full ontology.
> Every valid OWL Lite conclusion is a valid OWL DL conclusion.
> Every valid OWL DL conclusion is a valid OWL Full conclusion.

OWL inoltre si fonda sul principio di OWA (Open World Assumption), contrariamente a SQL che si fonda su quello di CWA (Closed World Assumption). Sotto questa assunzione (OWA) se un'affermazione non può essere provata come vera allo stato della conoscenza acquisita, non si può concludere che sia falsa. Tuttavia la semantica formale di OWL specifica come derivare le sue conseguenze logiche, ovvero i fatti che non sono presenti nell'ontologia ma che possono essere derivati logicamente dalla semantica. Questo rappresenta la capacità di ragionamento dell'ontologia. Si possono

infatti fare deduzioni logiche per ovviare a problemi di incompletezza e mancanza di dati.

2.2.2 DL- Lite

"***Description logics*** *(DL) are a family of knowledge representation languages which can be used to represent the concept definitions of an application domain (known as terminological knowledge) in a structured and formally well-understood way. The name description logic refers, on the one hand, to concept descriptions used to describe a domain and, on the other hand, to the logic-based semantics which can be given by a translation into first-order predicate logic*" (da http://en.wikipedia.org).

DL-Lite è il linguaggio usato in questo lavoro per modellare le ontologie. Esso è sostanzialmente un sottoisieme di OWL DL ma inoltre esso può modellare anche gli attributi di ruolo che non possono essere modellati con OWL DL.

Il dominio di interesse è costituito da oggetti ed è strutturato in:

- **Concetti**: corrispondono a classi e denotano insiemi di oggetti.
- **Ruoli**: corrispondono a relazioni binarie sugli oggetti, quindi alle associazioni.

La conoscenza si realizza attraverso le cosiddette **asserzioni**, cioè assiomi logici.

Due componenti fondamentali compongono la base di conoscenza di DL:

- **Terminological box** (TBox): contiene enunciati che descrivono le gerarchie di concetto (per esempio relazioni tra concetti) e ruoli. Modella formalmente un frammento di realtà. Descrive il *livello intensionale* della conoscenza.
- **Assertion box** (ABox): contiene asserzioni sulla conoscenza anche parziale del modello descritto nella TBox. Descrive dunque il *livello estensionale* della conoscenza.

Per quanto riguarda la sintassi della Description Logic, consiste in un insieme di simboli d i predicato unari che sono usati per denotare i nomi dei concetti; un inseme di relazioni binarie che sono usate per denotare i ruoli e una definizione ricorsiva per definire termini di concetto a partire da nomi di concetto e nomi di ruoli usando costruttori. Due sono le sintassi più usate per DL-Lite, una denominata **sintassi tedesca**, fa uso di costrutti logici/matematici, tuttavia risulta troppo complicata da parsare in documenti xml da dare in pasto ai software di interpretazione e

interrogazione delle ontologie e così è stata definita una sintassi testuale più intuitiva. Si tratta della **sintassi funzionale**.

In seguito vedremo come avviene il passaggio da *diagramma delle classi UML* → *sintassi tedesca* → *sintassi funzionale*.

2.3 Linguaggio SparSQL

SparSQL è un linguaggio di query per ontologie in DL-Lite la cui sintassi è inspirata dai linguaggi SQL per l'interrogazione delle basi di dati e SPARQL, uno dei più famosi linguaggi proposto dal W3C come linguaggio di query standard per OWL.

Il motivo per cui non viene usato direttamente il linguaggio SQL per interrogare le ontologie viene dal fatto che esso sarebbe indecidibile. Infatti, mentre nelle basi di dati vi è l'assunzione di CWA (Close World Assunption) in cui tutto ciò che non è nella base dati è ritenuto falso, nelle ontologie l'assunzione implicita che viene fatta è quella di OWA (Open World Assunption), cioè tutto quello che non è nella base di dati non è noto e quindi non può essere ritenuto falso.

SparSQL invece è in grado di operare una chiusura dinamica della conoscenza in modo controllato dall'utente mantenendo una capacità espressiva confrontabile a FOL pur rimanendo decidibile.

L'idea è utilizzare un sottoinsieme della FOL, le cosiddette **Conjunctive Queries (CQ)** e le **Union Conjunctive Queries (UCQ)** che sono sempre decidibili, ma hanno un potere espressivo limitato rispetto alla FOL rendendo impossibili le interrogazioni più complesse. Infatti la mancanza degli operatori NOT e FORALL (quantificatore universale) rende impossibile effettuare interrogazioni del tipo “seleziona tutti gli elementi che non rispettano una certa proprietà”.

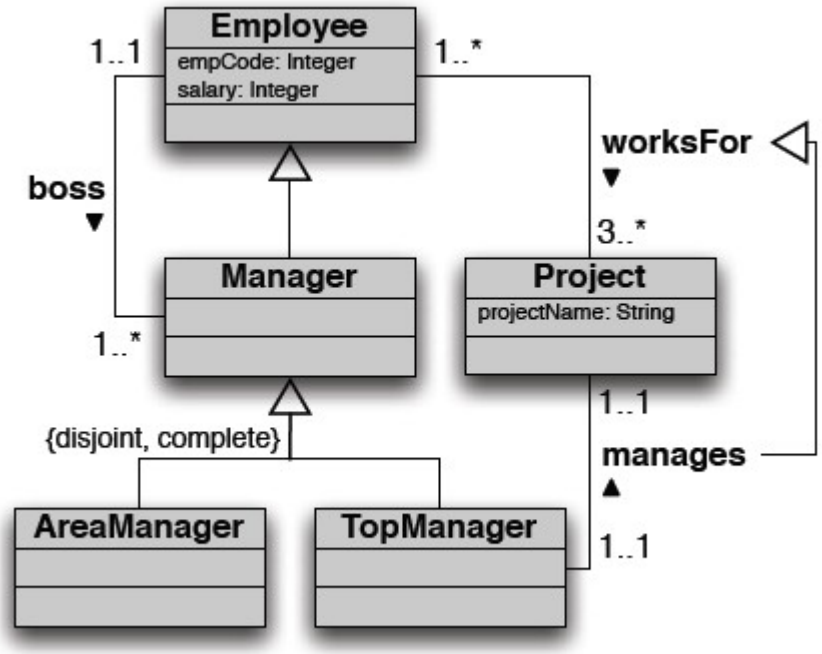
La soluzione a questa problematica viene dai *linguaggi epistemici.* Questo tipo di linguaggi si fondano sul principio che *“su ciò che conosci hai informazione completa”* e non dicono nulla su ciò che non si conosce, e grazie a questo le queries in FOL risultano decidibili.

SparSQL è un'implementazione di EQL-LITE(UCQ) che è appunto un linguaggio epistemico ed ha atomi che sono UCQ's. Questo linguaggio è quindi in grado di effettuare la chiusura della conoscenza dell'ontologia, cosa che SPARQL è in grado di fare, rendendo così decidibili le interrogazioni effettuate in SQL (da questi due linguaggi

infatti prende il nome).

3 Traduzione di diagrammi UML delle classi in DL-Lite

Prendiamo come esempio il seguente diagramma delle classi:



Vediamo ora come vengono tradotti i componenti di un diagramma delle classi UML e i relativi vincoli in ontologie mostrando dapprima la sintassi tedesca per poi passare a quella funzionale da dare in input al toolkit di interrogazione QuOnto/Mastro.

La sintassi tedesca che descive la TBox infatti risulta essere intuitiva in quanto usa costrutti logici/matematici e il passaggio a sintassi funzionale sarà praticamente automatico.

C'è tuttavia da considerare il fatto che non tutti i vincoli potranno essere espressi in sintassi funzionale, ma potranno essere verificati comunque con delle query booleane che saranno espresse in linguaggio SparSQL.

3.1 Classe (concetto)

In sintassi funzionale si modella con il costrutto `Class()`, mentre in sintassi tedesca viene dedotta la presenza del concetto.

Avremo quindi in riferimento al diagramma di esempio:

```
Class(Employee)
```

3.2 Attributi di classe

In riferimento all'attributo “salary” della classe “Employee” abbiamo in sintassi tedesca:

$$range(salary) \subseteq xsd : Integer$$
$$domain(salary) \subseteq Employee$$

mentre in sintassi funzionale:

```
DataPropertyRange(salary rdf:integer)
DataPropertyDomain(salary Employee)
```

Vediamo che per ogni attributo abbiamo l'uso di due statement, uno per la definizione del range (valori che può assumere) e uno per il dominio, cioè il concetto di interesse.

3.3 Generalizzazioni

Vediamo la traduzione dell'ISA tra le classi “Manager”, “AreaManager” e “TopManager”. Qui abbiamo anche i vincoli di *completezza* e *disgiunzione*, tuttavia il vincolo di completezza non è esprimibile ma potrà essere verificato comunque tramite una query booleana.

Sintassi tedesca:

$$AreaManager \subseteq Manager$$
$$TopManager \subseteq Manager$$
$$AreaManager \subseteq \neg TopManager$$

Sintassi funzionale:

```
SubClassOf(AreaManager Manager)
SubClassOf(TopManager Manager)
DisjointClasses(AreaManager TopManager)
```

In questo caso la query booleana che esprime il vincolo di completezza avrà come obiettivo di verificare che tutti gli elementi di “AreaManager” e “TopManager” formino tutti gli elementi di Manager e viceversa.

3.4 Associazione (ruolo)

Qui analizziamo l'associazione “Boss”. In ontologia avremo un ruolo a cui sono

associati un dominio, cioè il concetto sorgente dell'associazione, e il range del ruolo, cioè il concetto destinazione dell'associazione. Abbiamo quindo anche l'orientamento di tale associazione.

Sintassi tedesca:

$\exists Boss \subseteq Employee$

$\exists Bos\bar{s} \subseteq Manager$

[Nota: il segno "-" sopra l'ultima lettera del nome dell'associazione indica il verso opposto dell'associazione stessa]

Sintassi funzionale:

```
ObjectPropertyDomain(Boss Employee)
ObjectPropertyRange(Boss Manager)
```

Senza ulteriori specificazioni, si assume molteplicità 0..* per entrambi i versi dell'associazione. Nelle ontologie però sono esprimibili solo le cardinalità che hanno come estremi soltanto 0, 1 e *, cioè le seguenti: 0..* , 1..* , 0..1 , 1..1. Per verificare le altre cardinalità è necessario l'uso di query booleane.

Per ora tralasciamo questo aspetto che verrà esaminato in seguito in modo più completo.

3.5 Attributi di associazioni

Del tutto equivalente al caso di attributo di classe è la sintassi tedesca:

$range(nomeAttributo) \subseteq xsd:tipo$

$domain(nomeAttributo) \subseteq nomeAssociazione$

mentre in sintassi funzionale abbiamo un costrutto diverso, ma del tutto analogo:

```
ObjectPropertyDataRange(nomeAttributo rdf:tipo)
ObjectPropertyDataDomain(nomeAttributo nomeAssociazione)
```

infatti anche qui per ogni attributo uso due statement, uno per la definizione del range (valori che può assumere) e uno per il dominio, cioè il ruolo di interesse.

3.6 Specializzazioni

É del tutto analogo alla generalizzazione tra classi.

Prendiamo il caso delle associazioni “workFor” e “Manages” in cui la seconda estende la prima.

Sintassi tedesca:

$$manages \subseteq workFor$$

Sintassi funzionale:

```
SubObjectPropertyOf(manages workFor)
```

3.7 Cardinalità

Come accennato precedentemente nelle ontologie non si possono rappresentare tutti i vincoli di cardinalità, ma solamente quelli di tipo 0..*, 1..*, 0..1, 1..1. Per poter esprimere tutti gli altri tipi si deve ricorrere all'uso di query booleane che ne verificano la consistenza.

Iniziamo col vedere come si traducono i vincoli esprimibili e poi vedremo come costruire la query booleana per la verifica degli altri tipi di cardinalità.

A questo scopo faremo riferimento ad esempi del tutto generici.

3.7.1 Cardinalità di associazioni

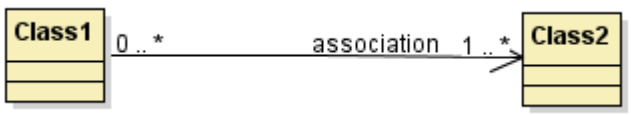
Cardinalità 0..*

Questo tipo di cardinalità è definita implicitamente per cui non occorre alcuna specificazione.

Cardinalità 1..* (cardinalità minima 1)

É necessario distinguere due casi a seconda se è riferita alla classe sorgente dell'associazione oppure alla classe destinazione.

Nel caso in cui la cardinalità 1..* è riferita alla classe destinazione, come mostra il seguente stralcio di diagramma:



il vincolo va tradotto in questo modo:

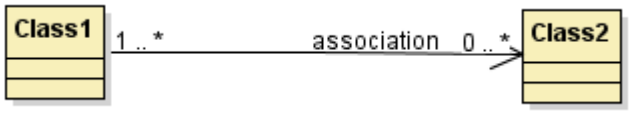
sintassi tedesca:

$$Class1 \subseteq \exists association$$

sintassi funzionale:

```
SubClassOf(Class1 ObjectMinCardinality(1 association))
```

Nel caso invece in cui la cardinalità 1..* è riferita alla classe sorgente, la traduzione avviene nel seguente modo:



sintassi tedesca:

$$\exists association \subseteq Class1$$

[Nota: il segno "-" sopra l'ultima lettera del nome dell'associazione indica il verso opposto dell'associazione stessa]

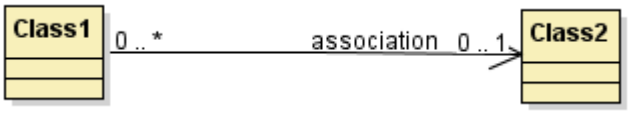
sintassi funzionale:

```
SubClassOf(ObjectMinCardinality(1
    inverseObjectPropertyOf(association)) Class1)
```

Cardinalità 0..1 (cardinalità massima 1)

Anche qui è necessario distinguere i precedenti due casi.

Caso cardinalità massima relativa alla classe destinazione:



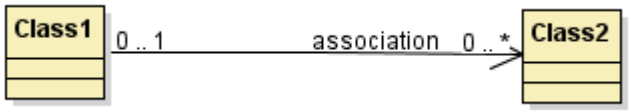
sintassi tedesca:

$$(funct\ association)$$

sintassi funzionale:

```
FunctionalObjectProperty(association)
```

Caso cardinalità massima relativa alla classe sorgente:



sintassi tedesca:

$$(funct\ associatio\bar{n})$$

[Nota: il segno "-" sopra l'ultima lettera del nome dell'associazione indica il verso opposto dell'associazione stessa]

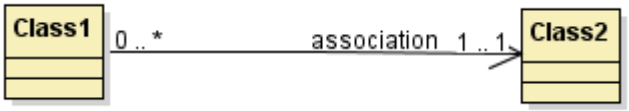
sintassi funzionale:

```
FunctionalObjectProperty(InverseObjectPropertyOf(association))
```

Cardinalità 1..1

In questo tipo di cardinalità vanno semplicemente "fusi" i vincoli precedenti di cardinalità massima 1 e cardinalità minima 1.

Esempio con cardinalità relativa alla classe destinazione



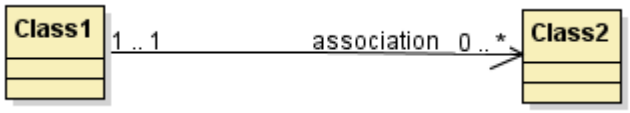
sintassi tedesca:

$$Class1 \subseteq \exists association$$
$$(funct\ association)$$

sintassi funzionale:

```
SubClassOf(Class1 ObjectMinCardinality(1 association))
FunctionalObjectProperty(association)
```

Esempio con cardinalità relativa alla classe sorgente



sintassi tedesca:

$$\exists association \subseteq Class1$$
$$(funct\ association)$$

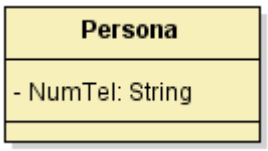
[Nota: il segno "-" sopra l'ultima lettera del nome dell'associazione indica il verso opposto dell'associazione stessa]

sintassi funzionale:

```
SubClassOf(ObjectMinCardinality(1
          InverseObjectPropertyOf(association)) Class1)
FunctionalObjectProperty(InverseObjectPropertyOf(association))
```

3.7.2 Cardinalità di attributi di classe

Prendiamo come riferimento la seguente classe:



Cardinalità 0..*

Come nel caso delle associazioni, questo tipo di cardinalità è definita implicitamente ma come sappiamo gli attributi di una classe/associazione, salvo esplicita dichiarazione, sono considerati avere cardinalità implicita 1..1, per cui è necessario imporre tale vincolo e non lasciare la cardinalità implicita poiché risulterebbe sbagliata.

Cardinalità 1..* (cardinalità minima 1)

Supponiamo che l'attributo "NumTel" abbia cardinalità 1..*, cioè sia un attributo multivalore.

sintassi tedesca:

$$Persona \subseteq domain(NumTel)$$

sintassi funzionale:

```
SubClassOf(Persona DataMinCardinality(1 NumTel))
```

Cardinalità 0..1 (cardinalità massima 1)

Supponiamo ora che l'attributo “NumTel” abbia cardinalità 0..1, cioè si tratti di un attributo opzionale.

sintassi tedesca:

$$(funct\ NumTel)$$

sintassi funzionale per classi:

```
FunctionalDataProperty(NumTel)
```

Cardinalità 1..1

Si tratta della cardinalità implicita per ogni diagramma delle classi UML di ogni attributo. Basta effettuare la fusione dei due costrutti precedenti.

sintassi tedesca:

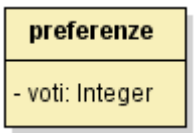
$$Persona \subseteq domain(NumTel)$$
$$(funct\ NumTel)$$

sintassi funzionale:

```
SubClassOf(Persona DataMinCardinality(1 NumTel))
FunctionalDataProperty(NumTel)
```

3.7.3 Cardinalità di attributi di associazione

Prendiamo come riferimento la seguente associaizone con attributo:



Cardinalità 0..*

Come nel caso degli attributi di classe, questo tipo di cardinalità è definita implicitamente ma come sappiamo gli attributi di una classe/associazione, salvo esplicita dichiarazione, sono considerati avere cardinalità implicita 1..1, per cui è necessario imporre tale vincolo e non lasciare la cardinalità implicita poiché risulterebbe sbagliata.

Cardinalità 1..* (cardinalità minima 1)

Supponiamo che l'attributo “voti” abbia cardinalità 1..*, cioè sia un attributo multivalore.

sintassi tedesca:

$$preferenze \subseteq domain(voti)$$

sintassi funzionale:

```
SubObjectPropertyOf(preferenze
    ObjectPropertyDataMinCardinality(1 voti))
```

Cardinalità 0..1 (cardinalità massima 1)

Supponiamo ora che l'attributo “voti” abbia cardinalità 0..1, cioè si tratti di un attributo opzionale.

sintassi tedesca:

$$(funct\ voti)$$

sintassi funzionale per classi:

```
FunctionalObjectPropertyData(voti)
```

Cardinalità 1..1

Si tratta della cardinalità implicita per ogni diagramma delle classi UML di ogni attributo. Basta effettuare la fusione dei due costrutti precedenti.

sintassi tedesca:

$$preferenze \subseteq domain(voti)$$

$$(funct\ voti)$$

sintassi funzionale:

```
SubObjectPropertyOf(Persona ObjectPropertyDataMinCardinality(1
    voti))

FunctionalObjectPropertyData(voti)
```

4 Vincoli non esprimibili in ontologie

Abbiamo visto come alcuni vincoli non si possono esprimere direttamente nell'ontologia in DL-lite, tuttavia è possibile verificarne la consistenza utilizzando altre vie.

Si tratta di verificare che le asserzioni fatte nell'ontologia siano coerenti con questi vincoli e per fare questo si ricorre all'uso di query booleane.

Nello specifico faremo uso del linguaggio SparSQL.

Andiamo ora a vedere come esprimere in query booleane i vincoli di completezza delle generalizzazioni e i vincoli sulle cardinalità diverse da 0..*, 1..*, 0..1, 1..1.

4.1 Completeness

Prendiamo sempre l'esempio con le classi “Manager”, “AreaManager”, “TopManager” in cui “Manager” estende le altre due e abbiamo il vincolo di disgiunzione e completezza.

Ricordo che abbiamo già visto come esprimere la disgiunzione ed è la completezza che non è possibile esprimere nelle ontologie in DL-Lite.

La query dovrà verificare che tutti gli elementi di “AreaManager“ U “TopManager” compongono l'insieme di elementi di “Manager“ e viceversa. Una strategia possibile è quella di verificare che l'insieme S = {Manager \ (AreaManager U TopManager)} sia l'insieme vuoto.

La query booleana risultante in linguaggio SparSQL è la seguente:

```
VERIFY NOT EXISTS (
      SELECT manager.empCode
      FROM sparqltable (
                  SELECT ?empCode
                  WHERE {
                        ?x rdf:type 'Manager'.
                        ?x :empCode ?empCode.
                  }
            ) manager
      WHERE manager.empCode not in (
            SELECT areaManager.empCode
            FROM sparqltable (
                        SELECT ?empCode
                        WHERE {
                              ?x rdf:type 'AreaManager'.
                              ?x :empCode ?empCode.
                        }
```

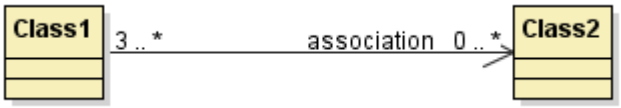
```
            ) areaManager
      UNION
      SELECT topManager.empCode
      FROM sparqltable (
            SELECT ?empCode
            WHERE {
                  ?x rdf:type 'TopManager'.
                  ?x :empCode ?empCode.
            }
      ) topManager
  )
)
```

4.2 Cardinalità diverse da 0..*, 1..*, 0..1 e 1..1

Distinguiamo due casi, il primo in cui la cardinalità minima è diversa da 0 e 1 e il secondo in cui la cardinalità massima è diversa da 1 e *.

Caso 1 (cardinalità minima diversa da 0 e 1)

Prendiamo in considerazione il seguente stralcio di diagramma delle classi:



Dobbiamo realizzare una query booleana che verifichi che per ogni associazione “association” vi siano almeno 3 occorrenze della classe “Class1”.

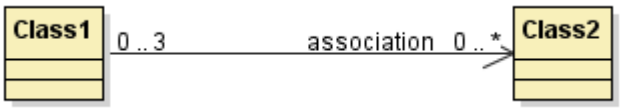
L'idea è verificare che l'insieme S, formato dalle associazioni “association” che hanno un numero di occorrenze della classe “Class1” < 3, sia vuoto.

La query booleana SparSQL risulta la seguente:

```
VERIFY NOT EXISTS (
      SELECT assoc.c1
      FROM sparqltable (
            SELECT ?c1 ?c2
            WHERE {
                  ?c1 rdf:type 'Class1'.
                  ?c1 :association ?c2.
                  ?c2 rdf:type 'Class2'.
            }
      ) assoc
      GROUP BY (asooc.c1)
      HAVING COUNT(*) < 3
)
```

*Caso 2 (cardinalità massima diversa da 1 e *)*

Il diagramma è il seguente:



Analogamente al caso precedente dobbiamo realizzare una query che verifica che per ogni associazione “association” non vi siano più di 3 occorrenze della classe “Class1”.

L'idea è verificare che l'insieme S, formato dalle associazioni “association” che hanno un numero di occorrenze della classe “Class1” > 3, sia vuoto.

La query booleana SparSQL risulta la seguente:

```
VERIFY NOT EXISTS (
      SELECT assoc.c1
      FROM sparqltable (
                  SELECT ?c1 ?c2
                  WHERE {
                        ?c1 rdf:type 'Class1'.
                        ?c1 :association ?c2.
                        ?c2 rdf:type 'Class2'.
                  }
      ) assoc
      GROUP BY (asooc.c1)
      HAVING COUNT(*) > 3
)
```

Ora siamo in grado di rappresentare in ontologia qualunque diagramma delle classi UML per poterlo così interrogare.

Passiamo ad applicare questi meccanismi a casi di studio e a questo scopo prendiamo come riferimento i diagrammi delle classi dei testi d'esame del corso di Progettazione del Software I dell'anno 2008.

5 Casi di studio

5.1 Esame del 16 aprile 2008

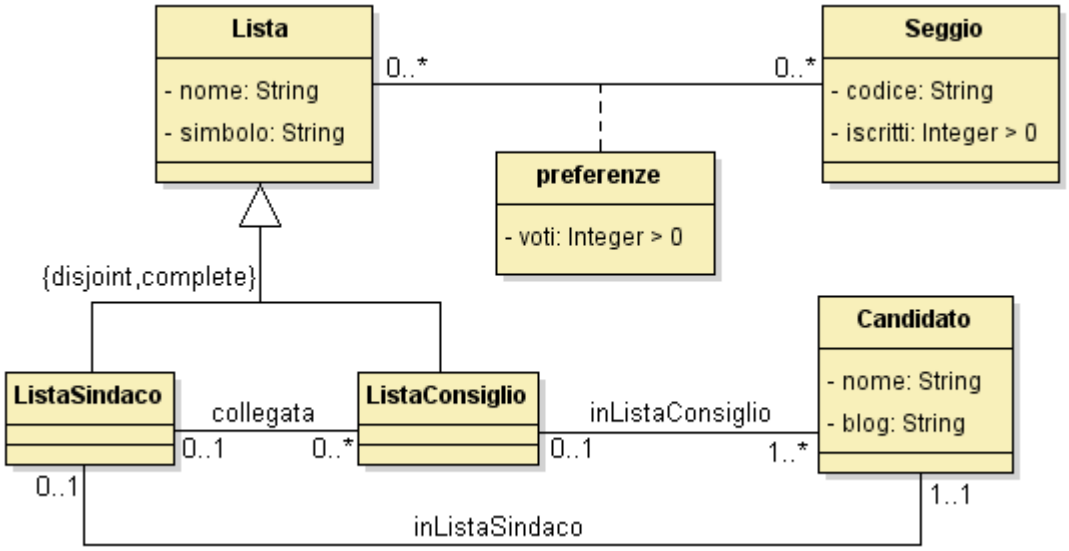
Requisiti

L'applicazione da progettare riguarda una parte del sistema di gestione di elezioni in un collegio elettorale del comune di Noncè. Alle elezioni vengono presentate delle liste, ciascuna con un nome ed un simbolo (una stringa che identifica il file con l'immagine). Le liste sono suddivise in liste per l'elezione del sindaco e liste per l'elezione del consiglio comunale. Ogni lista per il sindaco contiene un solo candidato. Mentre ogni lista per il consiglio contiene un insieme non vuoto e ordinato di candidati. Una lista per il consiglio può essere collegata ad una (e non più di una) lista per il sindaco. Ogni candidato è caratterizzato dal nome e dall'indirizzo web del suo blog (una stringa). Un candidato può presentarsi in al più una lista per il consiglio ed al più una lista per il sindaco. Nel comune di Noncè ci sono diversi seggi elettorali ciascuno contraddistinto da un codice e dal numero di cittadini aventi diritto al voto iscritti al seggio. Data una lista ed un seggio è di interesse memorizzare quanti voti ha preso la lista in quel seggio, ma solo se il numero di voti è maggiore di 0.

L'ufficio elezioni è interessato ad effettuare diversi controlli sui seggi e le liste, in particolare:

- data una lista ℓ, restituire l'insieme dei candidati a sindaco contenuti in ℓ, cioè se ℓ è una lista per il sindaco restituire l'insieme costituito dal solo candidato presente in essa, se ℓ è una lista per il consiglio restituire l'insieme formato dai candidati presenti in essa che sono anche candidati in una lista per il sindaco;
- dato un seggio s, restituire la percentuale di votanti, cioè il rapporto tra voti ottenuti dalle varie liste nel seggio s e numero di iscritti ad s.

Diagramma UML delle classi



TBox in sintassi funzionale DL-Lite

Iniziamo dichiarando le classi:

```
Class(Lista)
Class(Seggio)
Class(ListaSindaco)
Class(ListaConsiglio)
Class(Candidato)
```

e specificando gli attributi di classe con la relativa cardinalità. In realtà non sono specificate cardinalità diverse da 1..1 per cui tutti gli attributi avranno il costrutto per indicare cardinalità 1..1.

```
DataPropertyRange(nomeLista rdf:string)
DataPropertyDomain(nomeLista Lista)
SubClassOf(Lista dataMinCardinality(1 nomeLista))
FunctionalDataProperty(nomeLista)

DataPropertyRange(simbolo rdf:string)
DataPropertyDomain(simbolo Lista)
SubClassOf(Lista dataMinCardinality(1 simbolo))
FunctionalDataProperty(simbolo)

DataPropertyRange(codice rdf:string)
DataPropertyDomain(codice Seggio)
SubClassOf(Seggio dataMinCardinality(1 codice))
FunctionalDataProperty(codice)
```

```
DataPropertyRange(iscritti rdf:integer)
DataPropertyDomain(iscritti Seggio)
SubClassOf(Seggio dataMinCardinality(1 iscritti))
FunctionalDataProperty(iscritti)

DataPropertyRange(nome rdf:string)
DataPropertyDomain(nome Candidato)
SubClassOf(Candidato dataMinCardinality(1 nome))
FunctionalDataProperty(nome)

DataPropertyRange(blog rdf:string)
DataPropertyDomain(blog Candidato)
SubClassOf(Candidato dataMinCardinality(1 blog))
FunctionalDataProperty(blog)
```

Per quel che riguarda vincoli tra classi abbiamo una generalizzazione “disjoint, complete”. Ricordiamo che il vincolo di completezza sarà verificato tramite una query booleana mentre qui indicheremo solo il vincolo di disgiunzione.

```
SubClassOf(ListaSindaco Lista)
SubClassOf(ListaConsiglio Lista)
DisjointClasses(ListaSindaco ListaConsiglio)
```

A questo punto indichiamo le associazioni con i relativi vincoli di cardinalità:

```
ObjectPropertyDomain(preferenze Lista)
ObjectPropertyRange(preferenze Seggio)

ObjectPropertyDomain(inListaConsiglio Candidato)
ObjectPropertyRange(inListaConsiglio ListaConsiglio)
SubClassOf(ObjectMinCardinality(1
     inverseObjectPropertyOf(inListaConsiglio)) Candidato)
FunctionalObjectProperty(inListaConsiglio)

ObjectPropertyDomain(inListaSindaco Candidato)
ObjectPropertyRange(inListaSindaco ListaSindaco)
SubClassOf(ObjectMinCardinality(1
InverseObjectPropertyOf(inListaSindaco)) Candidato)
FunctionalObjectProperty(InverseObjectPropertyOf(inListaSindaco)
)
FunctionalObjectProperty(inListaSindaco)

ObjectPropertyDomain(collegata ListaSindaco)
ObjectPropertyRange(collegata ListaConsiglio)
FunctionalObjectProperty(InverseObjectPropertyOf(collegata))
```

e infine indichiamo l'unico attributo di associazione presente:

```
ObjectPropertyDataRange(voti rdf:integer)
ObjectPropertyDataDomain(voti preferenze)
SubObjectPropertyOf(preferenze
ObjectPropertyDataMinCardinality(1 voti))
FunctionalObjectPropertyData(voti)
```

Query per vincoli non esprimibili in DL-Lite

In questo caso abbiamo solo il vincolo di completezza da verificare, per cui risulta la query booleana:

```
VERIFY NOT EXISTS (
      SELECT      lista.nome
      FROM        sparqltable (
                        SELECT ?nome ?simbolo
                        WHERE {
                              ?x rdf:type 'Lista'.
                              ?x :nome ?nome.
                              ?x :simbolo ?simbolo.
                        }
                  ) lista
      WHERE       lista.nome not in (
                  SELECT listaSindaco.nome
                  FROM sparqltable (
                        SELECT ?nome ?simbolo
                        WHERE {
                              ?x rdf:type 'ListaSindaco'.
                              ?x :nome ?nome.
                              ?x :simbolo ?simbolo.
                              }
                        ) listaSindaco
                  WHERE lista.simbolo = listaSindaco.simbolo AND
                        lista.nome = listaSindaco.nome
                  UNION
                  SELECT listaConsiglio.nome
                  FROM sparqltable (
                        SELECT ?nome ?simbolo
                        WHERE {
                              ?x rdf:type 'ListaConsiglio'.
                              ?x :nome ?nome.
                              ?x :simbolo ?simbolo.
                        }
                  ) listaConsiglio
            WHERE lista.simbolo = listaConsiglio.simbolo AND
                  lista.nome = listaConsiglio.nome
      )
)
```

Query congiuntive per gli use case

Passiamo alla traduzione del programma descritto dagli use case in una query congiuntiva, sempre che questo sia possibile.

Prendiamo in considerazione il programma *CandidatiSindaco*. Si tratta di una union conjunctive query e può essere formulata come segue:

```
SELECT      cs.nome
FROM        sparqltable (
                  SELECT ?nome ?lista
                  WHERE {
                        ?x rdf:type 'Candidato'.
                        ?x :inListaSindaco ?y.
                        ?y rdf:type 'ListaSindaco'.
                        ?x :nome ?nome.
                        ?y :nomeLista ?lista.
                  }
            ) cs
WHERE       cs.lista = 'listaConsiglio'
UNION
SELECT      cc.nome
FROM        sparqltable (
                  SELECT     ?nome ?lista
                  WHERE {
                        ?x rdf:type 'Candidato'.
                        ?x :inListaConsiglio ?y.
                        ?y rdf:type 'ListaConsiglio'.
                        ?x :nome ?nome.
                        ?y :nomeLista ?lista.
                  }
            ) cc,
            sparqltable (
                  SELECT     ?nome
                  WHERE {
                        ?x rdf:type 'Candidato'.
                        ?x :inListaSindaco ?y.
                        ?y rdf:type 'ListaSindaco'.
                        ?x :nome ?nome.
                  }
            ) cs
WHERE       cc.nome = cs.nome AND
            cc.lista = 'listaConsiglio'
```

Prendiamo ora in considerazione il programma *PercentualeVotanti*. Questo programma non può essere espresso tramite una query congiuntiva poiché non è traducibile in una espressione FOL per via della necessità di utilizzo di operatori di somma e raggruppamento. Comunque la query corrispondente in sparSQL è la seguente:

```
SELECT     SUM(t.preferenze)*100/t.iscritti AS
percentualeVotanti
FROM       sparqltable (
                SELECT ?seggio ?lista ?iscritti ?preferenze
                WHERE {
                      ?x rdf:type 'Lista'.
                      (?x :preferenze ?y) :voti ?preferenze.
                           ?y rdf:type 'Seggio'.
                      ?x :nomeLista ?lista.
                      ?y :codice ?seggio.
                      ?y :iscritti ?iscritti.
                }
           ) t
WHERE      t.seggio = 'seggio1'
GROUP BY   t.seggio, t.iscritti
```

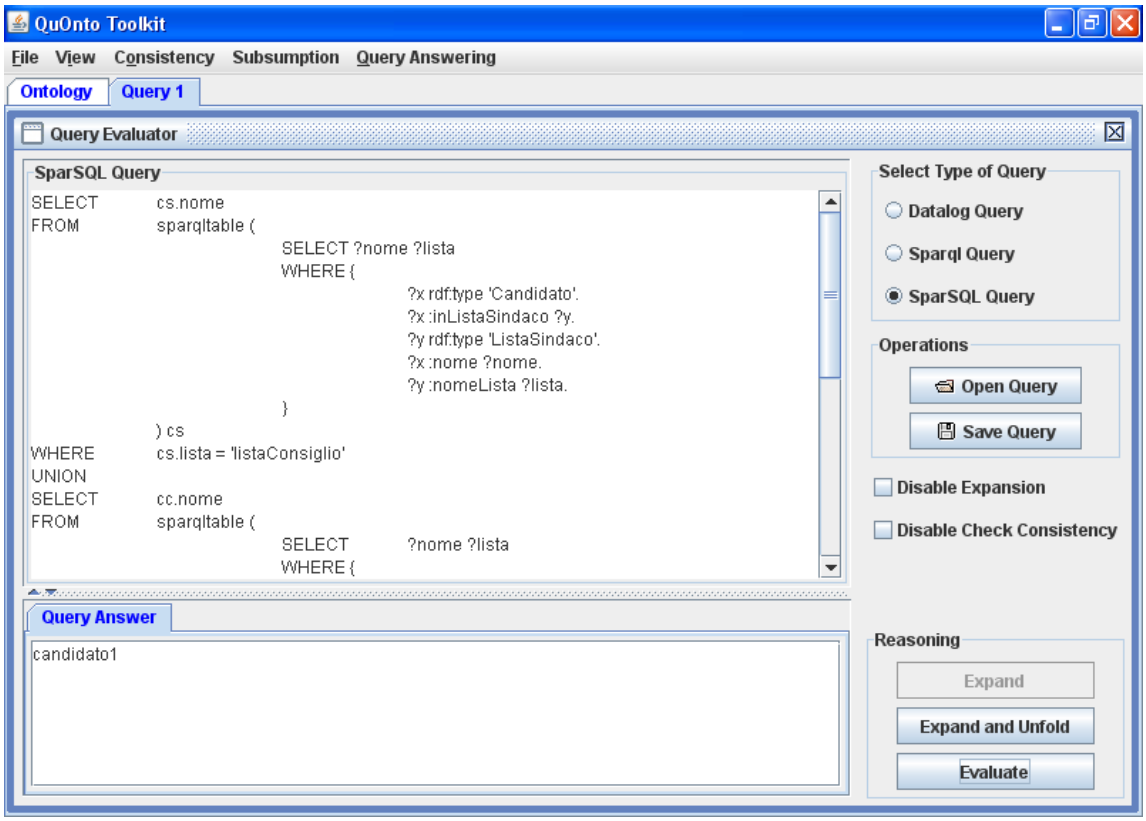
Test delle queries

Al fine di testare le queries è stata creata una **ABox** che riporto di seguito:

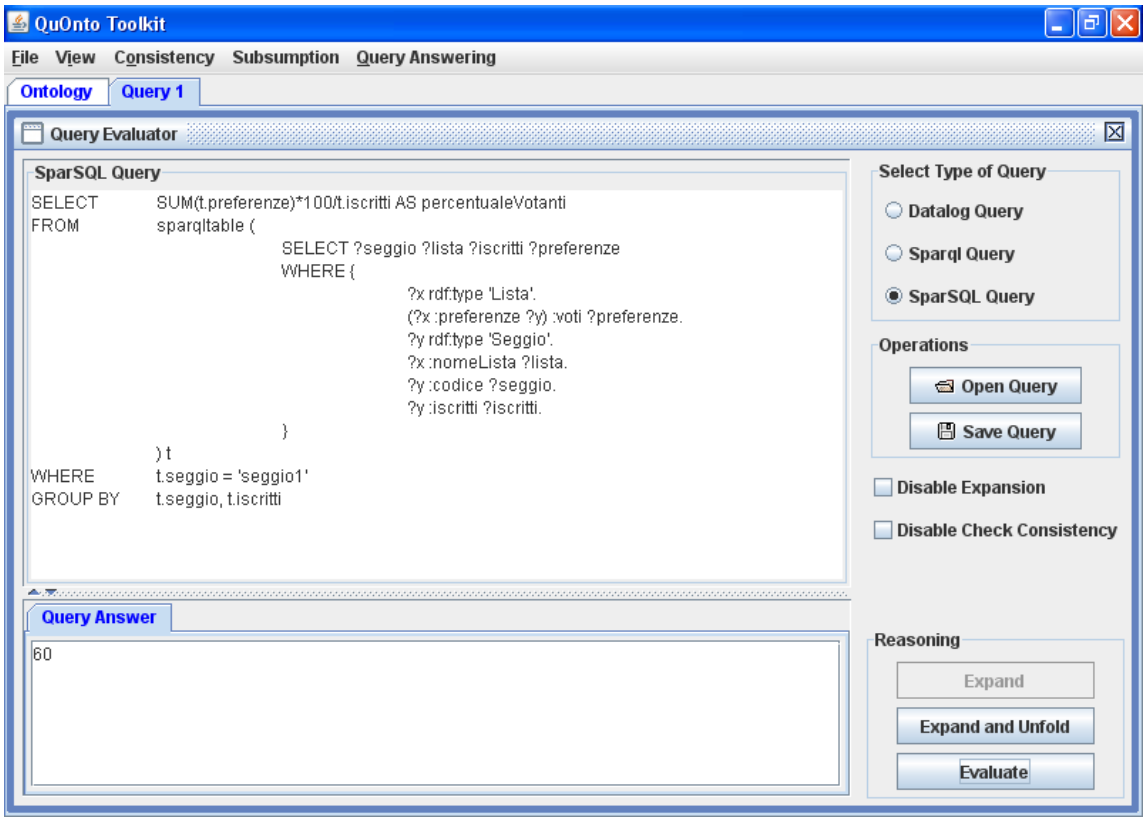
```
classassertion(L1 Lista)
dataPropertyAssertion(nomeLista L1 listaSindaco)
dataPropertyAssertion(simbolo L1 symLista1)

classassertion(L2 Lista)
dataPropertyAssertion(nomeLista L2 listaConsiglio)
dataPropertyAssertion(simbolo L2 symLista2)

classassertion(LS ListaSindaco)
dataPropertyAssertion(nomeLista LS listaSindaco)
dataPropertyAssertion(simbolo LS symLista1)

classassertion(LC ListaConsiglio)
dataPropertyAssertion(nomeLista LC listaConsiglio)
dataPropertyAssertion(simbolo LC symLista2)

classassertion(C1 Candidato)
dataPropertyAssertion(nome C1 candidato1)
dataPropertyAssertion(blog C1 blogCandidato1)

classassertion(C2 Candidato)
dataPropertyAssertion(nome C2 candidato2)
dataPropertyAssertion(blog C2 blogCandidato2)

classassertion(C3 Candidato)
dataPropertyAssertion(nome C3 candidato3)
dataPropertyAssertion(blog C3 blogCandidato3)

classassertion(C4 Candidato)
dataPropertyAssertion(nome C4 candidato4)
dataPropertyAssertion(blog C4 blogCandidato4)
```

```
classassertion(S1 Seggio)
dataPropertyAssertion(codice S1 seggio1)
dataPropertyAssertion(iscritti S1 100)

objectPropertyAssertion(inListaSindaco C1 LS)

objectPropertyAssertion(inListaConsiglio C1 LC)
objectPropertyAssertion(inListaConsiglio C2 LC)
objectPropertyAssertion(inListaConsiglio C3 LC)
objectPropertyAssertion(inListaConsiglio C4 LC)

objectPropertyAssertion(preferenze LS S1)
objectPropertyDataAssertion(voti LS S1 10)

objectPropertyAssertion(preferenze LC S1)
objectPropertyDataAssertion(voti LC S1 50)
```

Vediamo il risultato della query relativa al programma *CandidatiSindaco*:



e vediamo il risulta della query *PercentualeVotanti:*



5.2 Esame del 3 aprile 2008

Requisiti

L'applicazione da progettare riguarda una parte del sistema di gestione di un asilo per il corrente anno di iscrizione. Ogni classe è caratterizzata da un nome (una stringa), dai bambini ad essa assegnati e dalle maestre che vi insegnano. In una classe insegnano 1 o 2 maestre. Ogni bambino ha un nome e un'età (compresa tra 0 e 5 anni) ed è assegnato ad esattamente una classe. Ogni maestra ha un nome ed una anzianità di servizio (un intero). Alcune classi sono classi di scolarizzazione e ad esse vengono assegnati almeno 5 bambini non-scolarizzati. Dei bambini non-scolarizzati interessa sapere se portano ancora il pannolino (un booleano).

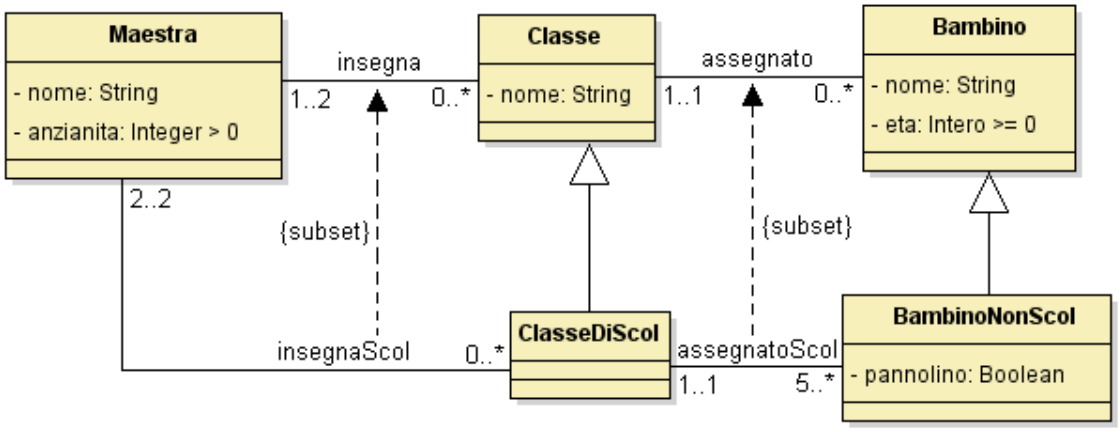
Nelle classi di scolarizzazione insegnano esattamente 2 maestre.

Un bambino è inizialmente accolto dalla struttura, dopo qualche giorno passa alla fase di inserimento. Completata questa fase il bambino viene considerato inserito. Se però si assenta per un periodo lungo allora torna alla fase di inserimento.

Il coordinatore didattico è interessato ad effettuare diversi controlli sulle classi, in particolare:

- dato un insieme di classi s, restituire il sottoinsieme formato dalle classi problematiche di s: dove una classe è problematica se è una classe di scolarizzazione tale che tutti i bambini assegnati ad essa sono non-scolarizzati;
- data una classe c, restituire l'età media dei bambini ad essa assegnati.

Diagramma UML delle classi



TBox in sintassi funzionale DL-Lite

Iniziamo con le classi:

```
Class(Maestra)
Class(Classe)
Class(Bambino)
Class(ClasseDiScol)
Class(BambinoNonScol)
```

Gli attributi di classe:

```
DataPropertyRange(nomeMaestra rdf:string)
DataPropertyDomain(nomeMaestra Maestra)
SubClassOf(Maestra dataMinCardinality(1 nomeMaestra))
FunctionalDataProperty(nomeMaestra)

DataPropertyRange(anzianita rdf:integer)
DataPropertyDomain(anzianita Maestra)
SubClassOf(Maestra dataMinCardinality(1 anzianita))
FunctionalDataProperty(anzianita)

DataPropertyRange(nomeClasse rdf:string)
DataPropertyDomain(nomeClasse Classe)
SubClassOf(Classe dataMinCardinality(1 nomeClasse))
FunctionalDataProperty(nomeClasse)

DataPropertyRange(nomeBambino rdf:string)
DataPropertyDomain(nomeBambino Bambino)
SubClassOf(Bambino dataMinCardinality(1 nomeBambino))
FunctionalDataProperty(nomeBambino)

DataPropertyRange(eta rdf:integer)
```

```
DataPropertyDomain(eta Bambino)
SubClassOf(Bambino dataMinCardinality(1 eta))
FunctionalDataProperty(eta)

DataPropertyRange(pannolino rdf:integer)
DataPropertyDomain(pannolino BambinoNonScol)
SubClassOf(BambinoNonScol dataMinCardinality(1 pannolino))
FunctionalDataProperty(pannolino)
```

Le generalizzazioni:

```
SubClassOf(ClasseDiScol Classe)
SubClassOf(BambinoNonScol Bambino)
```

Le associazioni:

In questo caso è da sottolineare che il vincolo di cardinalità massima 1 sull'associazione "assegnato" in relazione alla classe destinazione deve essere omesso in quanto si tratta di una restrizione di DL-LiteA. Questo vincolo dunque verrà verificato tramite una query booleana come anche le altre cardinalità non esprimibili.

```
ObjectPropertyDomain(insegna Maestra)
ObjectPropertyRange(insegna Classe)
SubClassOf(ObjectMinCardinality(1
inverseObjectPropertyOf(insegna)) Maestra)

ObjectPropertyDomain(assegnato Bambino)
ObjectPropertyRange(assegnato Classe)
SubClassOf(Bambino ObjectMinCardinality(1 assegnato))

ObjectPropertyDomain(insegnaScol Maestra)
ObjectPropertyRange(insegnaScol ClasseDiScol)

ObjectPropertyDomain(assegnatoScol BambinoNonScol)
ObjectPropertyRange(assegnatoScol ClasseDiScol)
SubClassOf(Bambino ObjectMinCardinality(1 assegnatoScol))
FunctionalObjectProperty(assegnatoScol)
```

Infine abbiamo le due specializzazioni:

```
SubObjectPropertyOf(insegnaScol insegna)
SubObjectPropertyOf(assegnatoScol assegnato)
```

Query per vincoli non esprimibili in DL-Lite

In questo diagramma abbiamo molti vincoli di cardinalità sulle associazioni diversi da 0 e 1, inoltre dobbiamo verificare il vincolo di cardinalità massima 1 sull'associazione "assegnato" poiché è stato omesso per via della specializzazione nell'associazione "assegnatoScol" che lo rendeva inesprimibile in DL-LiteA.

Vincolo insegna_max2:

```
VERIFY not exists (
      SELECT insegna.c2
      FROM sparqltable (
            SELECT ?c1 ?c2
            WHERE {
                  ?c1 rdf:type 'Maestra'.
                  ?c1 :insegna ?c2.
                  ?c2 rdf:type 'Classe'.
            }
      ) insegna
      GROUP BY (insegna.c2)
      HAVING COUNT(*) > 2
)
```

Vincolo assegnato_max1:

```
VERIFY not exists (
      SELECT assegnato.c1
      FROM sparqltable (
            SELECT ?c1 ?c2
            WHERE {
                  ?c1 rdf:type 'Bambino'.
                  ?c1 :assegnato ?c2.
                  ?c2 rdf:type 'Classe'.
            }
      ) assegnato
      GROUP BY (assegnato.c1)
      HAVING COUNT(*) > 1
)
```

Vincolo assegnatoScol_min5:

```
VERIFY not exists (
      SELECT assegnatoScol.c2
      FROM sparqltable (
            SELECT ?c1 ?c2
            WHERE {
                  ?c1 rdf:type 'BambinoNonScol'.
                  ?c1 :assegnatoScol ?c2.
```

```
                ?c2 rdf:type 'ClasseDiScol'.
          }
     ) assegnatoScol
     GROUP BY (assegnatoScol.c2)
     HAVING COUNT(*) < 5
)
```

Vincolo insegnaScol_min2:

```
VERIFY not exists (
     SELECT insegnaScol.c2
     FROM sparqltable (
          SELECT ?c1 ?c2
          WHERE {
                ?c1 rdf:type 'Maestra'.
                ?c1 :insegnaScol ?c2.
                ?c2 rdf:type 'ClasseDiScol'.
          }
     ) insegnaScol
     GROUP BY (insegnaScol.c2)
     HAVING COUNT(*) < 2
)
```

Vincolo insegnaScol_max2:

```
VERIFY not exists (
     SELECT insegnaScol.c2
     FROM sparqltable (
          SELECT ?c1 ?c2
          WHERE {
                ?c1 rdf:type 'Maestra'.
                ?c1 :insegnaScol ?c2.
                ?c2 rdf:type 'ClasseDiScol'.
          }
     ) insegnaScol
     GROUP BY (insegnaScol.c2)
     HAVING COUNT(*) > 2
)
```

Query congiuntive per gli use case

Passiamo alla traduzione del programma descritto dagli use case in una query congiuntiva, sempre che questo sia possibile.

Prendiamo in considerazione il programma *ClassiProblematiche*. Questo programma non può essere formulata come una conjunctive query poichè non è possibile

dichiarare gli operatori **NOT** e **FORALL**. Infatti la quer in SparSQL risulta essere:

```
SELECT     t.classe
FROM       sparqltable (
                SELECT ?bambino ?classe
                WHERE {
                      ?x rdf:type 'Bambino'.
                      ?x :assegnato ?y.
                      ?y rdf:type 'ClasseDiScol'.
                      ?x :nomeBambino ?bambino.
                      ?y :nomeClasse ?classe.
                }
           ) t
WHERE      t.classe = 'classeScol1'
GROUP BY   t.classe
HAVING COUNT(*) = (
     SELECT     COUNT(*)
     FROM       sparqltable (
                      SELECT ?bambino ?classe
                      WHERE {
                            ?x rdf:type 'BambinoNonScol'.
                            ?x :assegnatoScol ?y.
                            ?y rdf:type 'ClasseDiScol'.
                            ?x :nomeBambino ?bambino.
                            ?y :nomeClasse ?classe.
                      }
                ) t
     WHERE      t.classe = 'classeScol1'
)
```

Prendiamo in considerazione il programma *EtaMedia*. Neanche questo programma può essere espresso tramite una query congiuntiva poiché non è traducibile in una espressione FOL per via della necessità di utilizzo di operatori di **media** e **raggruppamento**. La query comunque è:

```
SELECT     AVG(t.eta)
FROM       sparqltable (
                SELECT ?bambino ?eta ?classe
                WHERE {
                      ?x rdf:type 'Bambino'.
                      ?x :assegnato ?y.
                      ?y rdf:type 'Classe'.
                      ?x :nomeBambino ?bambino.
                      ?x :eta ?eta.
                      ?y :nomeClasse ?classe.
                }
           ) t
WHERE      t.classe = 'classeScol1'
```

Test delle queries

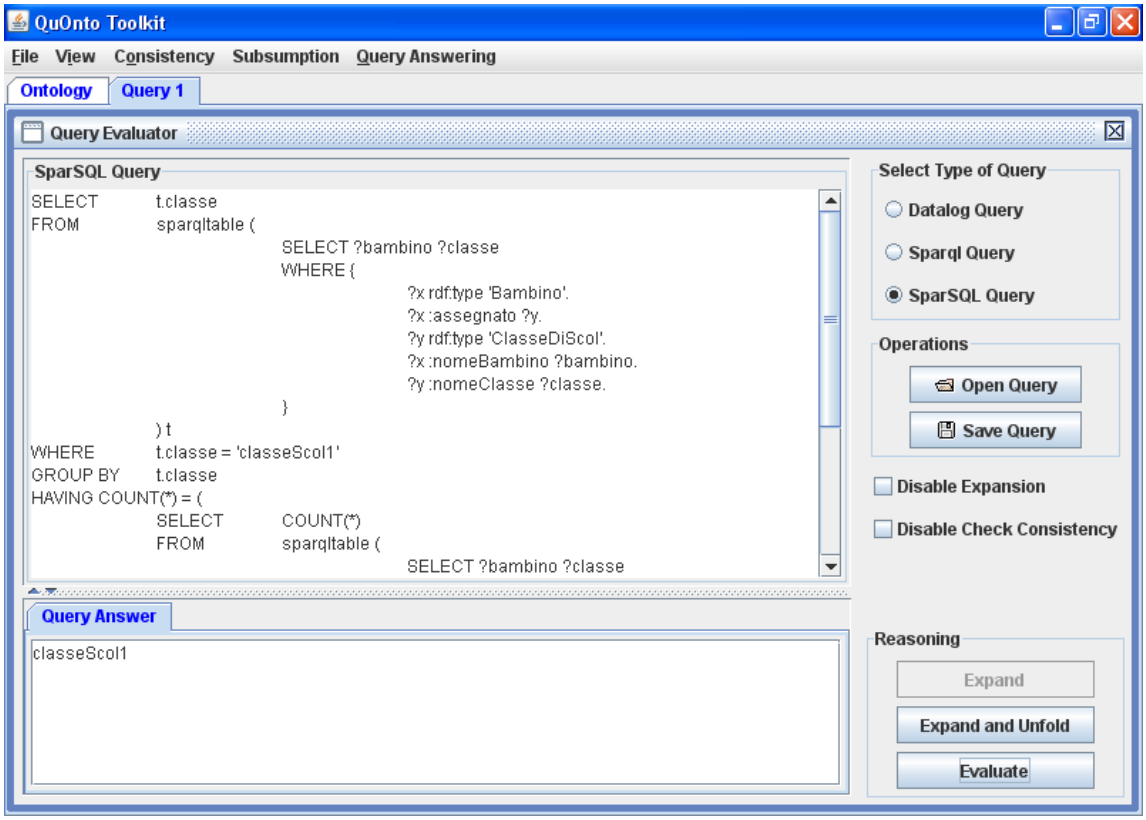
Al fine di testare le queries è stata creata una ABox che riporto di seguito:

```
classassertion(M1 Maestra)
dataPropertyAssertion(nomeMaestra M1 maestra1)
dataPropertyAssertion(anzianita M1 20)

classassertion(M2 Maestra)
dataPropertyAssertion(nomeMaestra M2 maestra2)
dataPropertyAssertion(anzianita M2 30)

classassertion(C1 Classe)
dataPropertyAssertion(nomeClasse C1 classe1)

classassertion(C2 Classe)
dataPropertyAssertion(nomeClasse C2 classe2)

classassertion(B1 Bambino)
dataPropertyAssertion(nomeBambino B1 bamb1)
dataPropertyAssertion(eta B1 8)

classassertion(BN1 BambinoNonScol)
dataPropertyAssertion(nomeBambino BN1 bambNon1)
dataPropertyAssertion(eta BN1 10)
dataPropertyAssertion(pannolino BN1 1)

classassertion(BN2 BambinoNonScol)
dataPropertyAssertion(nomeBambino BN2 bambNon2)
dataPropertyAssertion(eta BN2 12)
dataPropertyAssertion(pannolino BN2 1)

classassertion(BN3 BambinoNonScol)
dataPropertyAssertion(nomeBambino BN3 bambNon3)
dataPropertyAssertion(eta BN3 8)
dataPropertyAssertion(pannolino BN3 1)

classassertion(BN4 BambinoNonScol)
dataPropertyAssertion(nomeBambino BN4 bambNon4)
dataPropertyAssertion(eta BN4 10)
dataPropertyAssertion(pannolino BN4 1)

classassertion(BN5 BambinoNonScol)
dataPropertyAssertion(nomeBambino BN5 bambNon5)
dataPropertyAssertion(eta BN5 9)
dataPropertyAssertion(pannolino BN5 1)

classassertion(CS1 ClasseDiScol)
dataPropertyAssertion(nomeClasse CS1 classeScol1)

classassertion(CS2 ClasseDiScol)
```

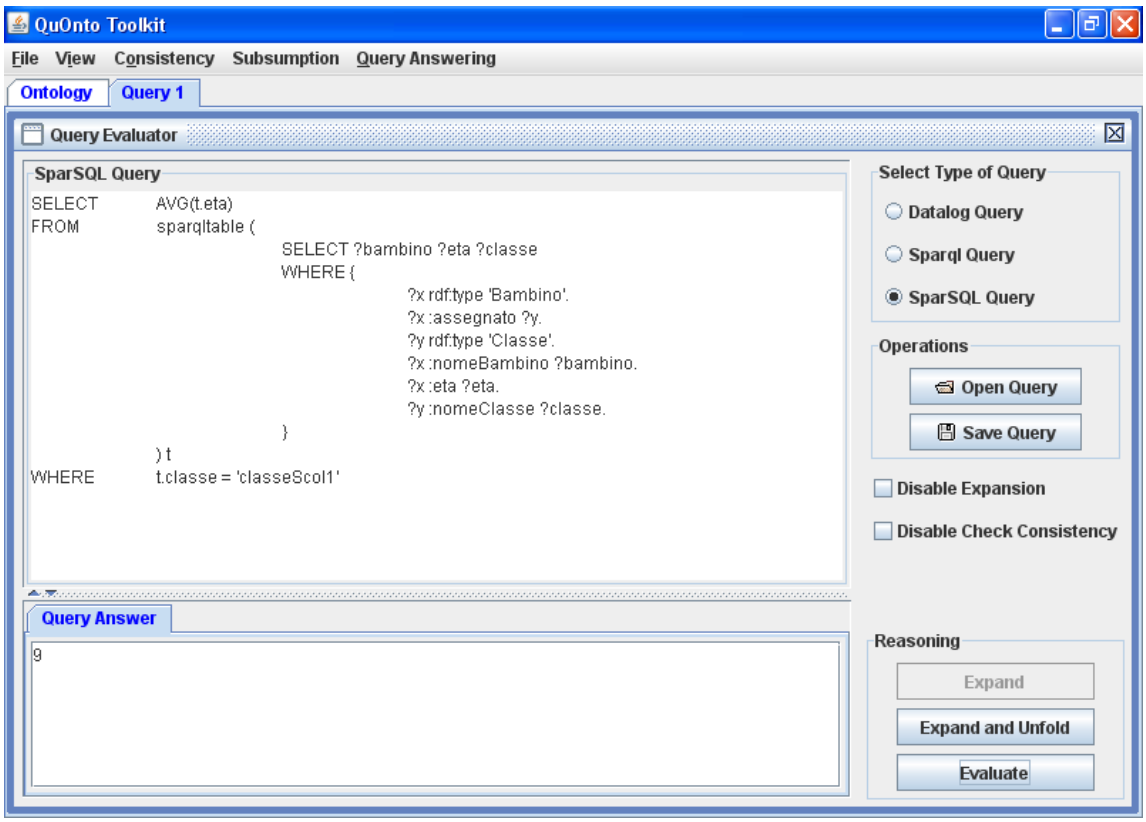
```
dataPropertyAssertion(nomeClasse CS2 classeScol2)

objectPropertyAssertion(assegnatoScol BN1 CS1)
objectPropertyAssertion(assegnatoScol BN2 CS1)
objectPropertyAssertion(assegnatoScol BN3 CS1)
objectPropertyAssertion(assegnatoScol BN4 CS1)
objectPropertyAssertion(assegnatoScol BN5 CS1)

objectPropertyAssertion(assegnato B1 C1)

objectPropertyAssertion(insegna M1 C1)
objectPropertyAssertion(insegna M2 C1)

objectPropertyAssertion(insegnaScol M1 CS1)
objectPropertyAssertion(insegnaScol M2 CS1)
```

Vediamo il risultato che offre QuOnto su queste queries.

query *ClassiProblematiche:*



query *EtaMedia:*



5.3 Esame del 16 luglio 2008

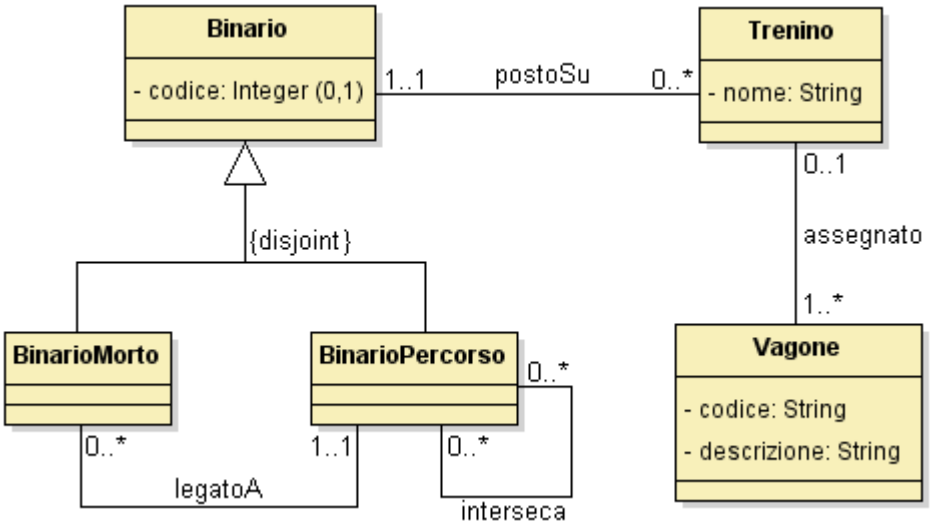
Requisiti

L'applicazione da progettare riguarda la gestione di trenini elettrici che girano su un sistema di rotaie condiviso. Ogni trenino è caratterizzato da un nome (una stringa) e da una sequenza non vuota ordinata di vagoni ad esso assegnati. Un vagone può ovviamente essere assegnato ad al più un trenino in ogni momento.

Ogni vagone è caratterizzato da un codice (una stringa) e da una descrizione (una stringa). Ogni trenino è posto su un binario. Ciascun binario può essere caratterizzato da un codice (un intero) e può essere un binario morto oppure un percorso. Ogni binario morto è legato ad un singolo percorso. Ogni percorso può intersecare altri percorsi (si noti che se il percorso i interseca il percorso j allora ovviamente il percorso j interseca il percorso i). Il trenino, che è inizialmente fermo, può essere fatto partire in avanti o indietro, una volta partito può essere fermato. E' di interesse conoscere lo stato del trenino. Inoltre il trenino può essere modificato, aggiungendo o eliminando vagoni solo quando è fermo. L'utente del sistema è interessato ad effettuare diversi controlli in particolare:

- dato un insieme I di trenini, verificare se almeno due di essi sono posti su percorsi che si intersecano;
- dato un insieme J di vagoni restituire il sottoinsieme di J costituito da vagoni non assegnati ad alcun trenino.

Diagramma UML delle classi



TBox in sintassi funzionale DL-Lite

Le classi sono:

```
Class(Trenino)
Class(Binario)
Class(BinarioMorto)
Class(BinarioPercorso)
Class(Vagone)
```

Gli attributi di classe:

```
DataPropertyRange(codiceBinario rdf:integer)
DataPropertyDomain(codiceBinario Binario)
FunctionalDataProperty(codiceBinario)

DataPropertyRange(nome rdf:string)
DataPropertyDomain(nome Trenino)
SubClassOf(Trenino dataMinCardinality(1 nome))
FunctionalDataProperty(nome)

DataPropertyRange(codiceVagone rdf:string)
DataPropertyDomain(codiceVagone Vagone)
SubClassOf(Vagone dataMinCardinality(1 codiceVagone))
FunctionalDataProperty(codiceVagone)

DataPropertyRange(descrizione rdf:string)
DataPropertyDomain(descrizione Vagone)
SubClassOf(Vagone dataMinCardinality(1 descrizione))
FunctionalDataProperty(descrizione)
```

Notiamo che l'attributo “codice” della classe “Binario” è opzionale (come si evince dai requisiti).

Le generalizzazioni:

```
SubClassOf(BinarioMorto Binario)
SubClassOf(BinarioPercorso Binario)
DisjointClasses(BinarioMorto BinarioPercorso)
```

Le associazioni e i relativi vincoli:

```
ObjectPropertyDomain(assegnato Vagone)
ObjectPropertyRange(assegnato Trenino)
SubClassOf(ObjectMinCardinality(1
inverseObjectPropertyOf(assegnato)) Vagone)
FunctionalObjectProperty(assegnato)

ObjectPropertyDomain(postoSu Trenino)
ObjectPropertyRange(postoSu Binario)
SubClassOf(Trenino ObjectMinCardinality(1 postoSu))
FunctionalObjectProperty(postoSu)

ObjectPropertyDomain(legatoA BinarioMorto)
ObjectPropertyRange(legatoA BinarioPercorso)
SubClassOf(BinarioMorto ObjectMinCardinality(1 legatoA))
FunctionalObjectProperty(legatoA)

ObjectPropertyDomain(interseca BinarioPercorso)
ObjectPropertyRange(interseca BinarioPercorso)
```

Query per vincoli non esprimibili in DL-Lite

Non sono necessarie query per verificare vincoli poiché sono stati tutti espressi in sintassi funzionale per DL-Lite.

Query congiuntive per gli use case

Prendiamo il primo programma dello use case: *SiIntersecano.* Questo può essere espresso con la seguente formula:

Supponiamo di avere l'insieme di trenini I = {'treno1','treno2','treno3'}. Con un numero maggiore di trenini è la stessa cosa, ho assunto un numero pari a tre per semplificare la lunghezza della query che in SparSQL risulta essere:

```
VERIFY EXISTS (
     SELECT     t1.treno, t1.trenoInters
```

```
     FROM       sparqltable (
                      SELECT ?treno ?trenoInters
                      WHERE {
                      ?x rdf:type 'Trenino'.
                      ?x :postoSu ?y.
                      ?y rdf:type 'BinarioPercorso'.
                      ?y :interseca ?z.
                      ?z rdf:type 'BinarioPercorso'.
                      ?t :postoSu ?z.
                      ?t rdf:type 'Trenino'.
                      ?x :nome ?treno.
                      ?t :nome ?trenoInters.
                      }
                ) t1
     WHERE      t1.treno = 'treno1' AND t1.trenoInters =
'treno2'
     UNION
     SELECT     t1.treno, t1.trenoInters
     FROM       sparqltable (
                SELECT ?treno ?trenoInters
                WHERE {
                      ?x rdf:type 'Trenino'.
                      ?x :postoSu ?y.
                      ?y rdf:type 'BinarioPercorso'.
                      ?y :interseca ?z.
                      ?z rdf:type 'BinarioPercorso'.
                      ?t :postoSu ?z.
                      ?t rdf:type 'Trenino'.
                      ?x :nome ?treno.
                      ?t :nome ?trenoInters.
                      }
                ) t1
     WHERE      t1.treno = 'treno2' AND t1.trenoInters =
'treno1'
     UNION
     SELECT     t1.treno, t1.trenoInters
     FROM       sparqltable (
                SELECT ?treno ?trenoInters
                WHERE {
                      ?x rdf:type 'Trenino'.
                      ?x :postoSu ?y.
                      ?y rdf:type 'BinarioPercorso'.
                      ?y :interseca ?z.
                      ?z rdf:type 'BinarioPercorso'.
                      ?t :postoSu ?z.
                      ?t rdf:type 'Trenino'.
                      ?x :nome ?treno.
                      ?t :nome ?trenoInters.
                      }
                ) t1
     WHERE      t1.treno = 'treno1' AND t1.trenoInters =
'treno3'
```

```
      UNION
      SELECT      t1.treno, t1.trenoInters
      FROM        sparqltable (
                  SELECT ?treno ?trenoInters
                  WHERE {
                        ?x rdf:type 'Trenino'.
                        ?x :postoSu ?y.
                        ?y rdf:type 'BinarioPercorso'.
                        ?y :interseca ?z.
                        ?z rdf:type 'BinarioPercorso'.
                        ?t :postoSu ?z.
                        ?t rdf:type 'Trenino'.
                        ?x :nome ?treno.
                        ?t :nome ?trenoInters.
                        }
                  ) t1
      WHERE       t1.treno = 'treno3' AND t1.trenoInters =
'treno1'
      UNION
      SELECT      t1.treno, t1.trenoInters
      FROM        sparqltable (
                  SELECT ?treno ?trenoInters
                  WHERE {
                        ?x rdf:type 'Trenino'.
                        ?x :postoSu ?y.
                        ?y rdf:type 'BinarioPercorso'.
                        ?y :interseca ?z.
                        ?z rdf:type 'BinarioPercorso'.
                        ?t :postoSu ?z.
                        ?t rdf:type 'Trenino'.
                        ?x :nome ?treno.
                        ?t :nome ?trenoInters.
                        }
                  ) t1
      WHERE       t1.treno = 'treno2' AND t1.trenoInters =
'treno3'
      UNION
      SELECT      t1.treno, t1.trenoInters
      FROM        sparqltable (
                  SELECT ?treno ?trenoInters
                  WHERE {
                        ?x rdf:type 'Trenino'.
                        ?x :postoSu ?y.
                        ?y rdf:type 'BinarioPercorso'.
                        ?y :interseca ?z.
                        ?z rdf:type 'BinarioPercorso'.
                        ?t :postoSu ?z.
                        ?t rdf:type 'Trenino'.
                        ?x :nome ?treno.
                        ?t :nome ?trenoInters.
                        }
                  ) t1
```

```
     WHERE      t1.treno = 'treno3' AND t1.trenoInters =
'treno2'
)
```

In realtà potevo usare gli operatori **IN**, **COUNT** e **>**, ma in questo modo ho la forma di query congiuntiva.

Il secondo programma invece non è esprimibile come conjunctive query a causa della necessità dell'operatore **NOT** come vediamo dalla seguente query SparSQL *VagoniVuoti*:

```
SELECT     t.vagone
FROM       sparqltable (
                SELECT ?vagone
                WHERE {
                      ?x rdf:type 'Vagone'.
                      ?x :codiceVagone ?vagone.
                }
           ) t
WHERE      t.vagone NOT IN (
                SELECT     t1.vagone
                FROM       sparqltable (
                                SELECT ?vagone
                                WHERE {
                                      ?x rdf:type 'Vagone'.
                                      ?x :assegnato ?y.
                                      ?y rdf:type 'Trenino'.
                                      ?x :codiceVagone ?vagone.
                                }
                           ) t1
           )
           AND
           t.vagone IN (
                SELECT     t2.vagone
                FROM       sparqltable (
                                SELECT ?vagone
                                WHERE {
                                      ?x rdf:type 'Vagone'.
                                      ?x :codiceVagone ?vagone.
                                }
                           ) t2
                WHERE      t2.vagone = 'vagone3' OR
                           t2.vagone = 'vagone4' OR
                           t2.vagone = 'vagone5'
           )
```

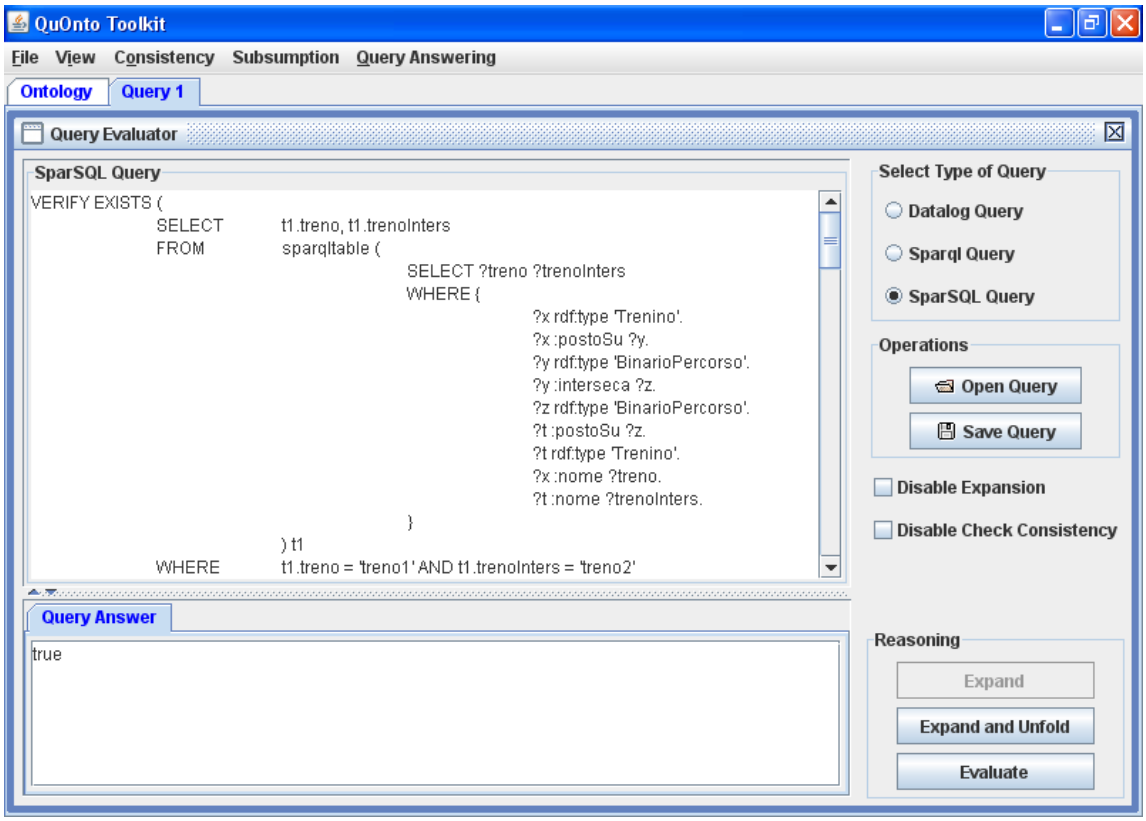
Test delle queries

Al fine di testare le queries è stata creata una ABox che riporto di seguito:

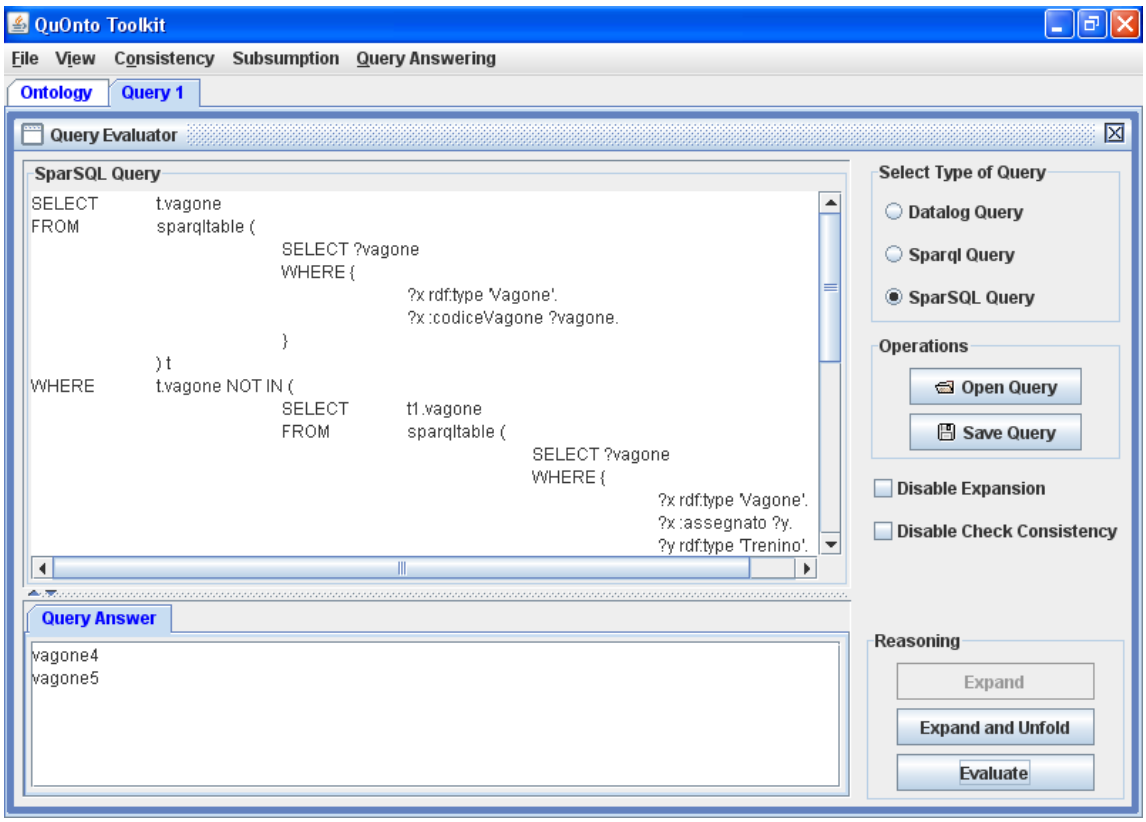
```
classassertion(B1 BinarioPercorso)
dataPropertyAssertion(codiceBinario B1 101)

classassertion(B2 BinarioPercorso)
dataPropertyAssertion(codiceBinario B2 102)

classassertion(B3 BinarioPercorso)
dataPropertyAssertion(codiceBinario B3 103)

classassertion(T1 Trenino)
dataPropertyAssertion(nome T1 treno1)

classassertion(T2 Trenino)
dataPropertyAssertion(nome T2 treno2)

classassertion(T3 Trenino)
dataPropertyAssertion(nome T3 treno3)

classassertion(V1 Vagone)
dataPropertyassertion(codiceVagone V1 vagone1)
dataPropertyassertion(descrizione V1 vagone1)

classassertion(V2 Vagone)
dataPropertyassertion(codiceVagone V2 vagone2)
dataPropertyassertion(descrizione V2 vagone2)

classassertion(V3 Vagone)
dataPropertyassertion(codiceVagone V3 vagone3)
dataPropertyassertion(descrizione V3 vagone3)

classassertion(V4 Vagone)
dataPropertyassertion(codiceVagone V4 vagone4)
dataPropertyassertion(descrizione V4 vagone4)

classassertion(V5 Vagone)
dataPropertyassertion(codiceVagone V5 vagone5)
dataPropertyassertion(descrizione V5 vagone5)

objectPropertyAssertion(assegnato V1 T1)
objectPropertyAssertion(assegnato V2 T2)
objectPropertyAssertion(assegnato V3 T3)

objectPropertyAssertion(postoSu T1 B1)
objectPropertyAssertion(postoSu T2 B2)
objectPropertyAssertion(postoSu T3 B3)

objectPropertyAssertion(interseca B1 B2)
```

```
objectPropertyAssertion(interseca B1 B3)
```

Di seguito riporto l'output del toolkit QuOnto sulla query del programma use case *SiIntersecano*:



E l'output della query del programma *VagoniLiberi*:



5.4 Esame del 12 settembre 2008

Requisiti

L'applicazione da progettare riguarda la gestione di gare che si svolgono in un campionato di ciclismo su pista. Ogni gara è caratterizzata dall'orario, dal giorno e dal nome della città in cui si svolge, e ad essa sono iscritti dei corridori. Ogni corridore è caratterizzato dal nome e dalla squadra di appartenenza (unica). Di una squadra sono di interesse il nome, l'anno di fondazione ed il capitano, che è un corridore appartenente alla squadra. Del capitano, che è unico per ciascuna squadra, si vuole anche sapere l'anno dal quale appartiene alla squadra di cui è capitano. Per le gare disputate interessa l'ordine di arrivo dei corridori che vi hanno preso parte. Per ogni gara disputata vi è almeno un corridore che compare nell'ordine di arrivo.

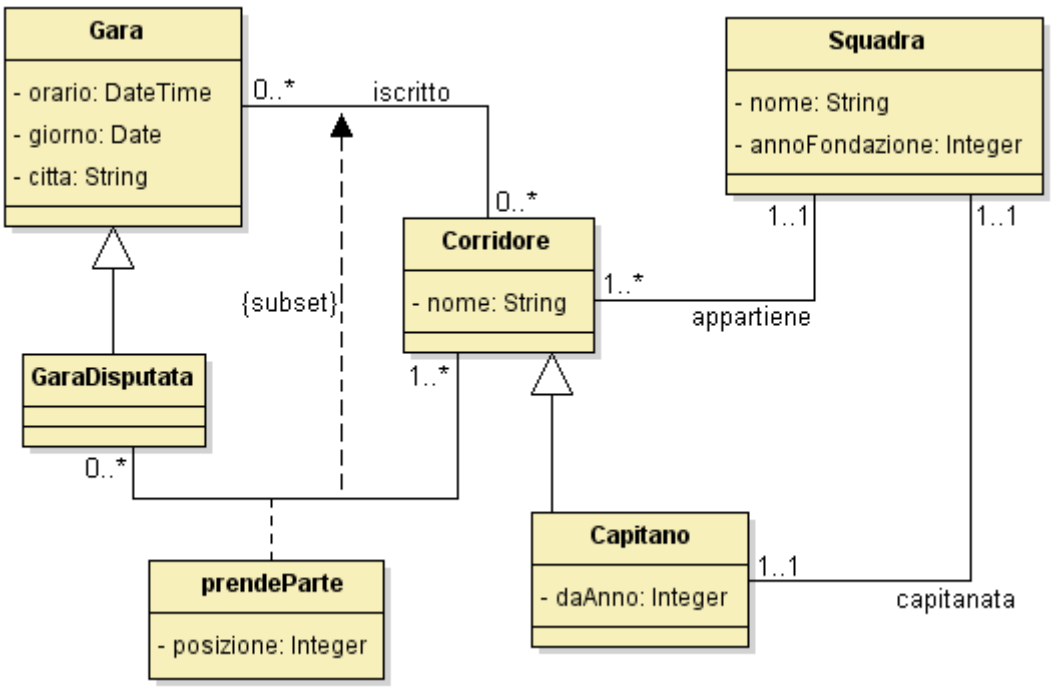
Ovviamente, possono comparire nell'ordine di arrivo di una gara disputata solo corridori iscritti a quella gara.

Una gara si considera disputata dal momento in cui ha inizio ed è quindi in fase di svolgimento. Solo durante questa fase è possibile specificare l'ordine di arrivo dei corridori. Una volta terminata, la gara deve essere validata dai giudici di gara: nel caso in cui venga rilevata qualche irregolarità, la gara viene annullata, altrimenti la gara è considerata valida. E' di interesse monitorare lo stato della gara disputata.

L'utente del sistema è interessato ad effettuare diversi controlli. In particolare:

- data una gara disputata restituire l'insieme dei corridori ritirati, cioè i corridori iscritti alla gara ma che una volta disputata non compaiono nell'ordine di arrivo;
- dato un corridore, restituire l'insieme delle gare che ha vinto.

Diagramma UML delle classi



TBox in sintassi funzionale DL-Lite

Classi:

```
Class(Gara)
Class(GaraDisputata)
Class(Squadra)
Class(Corridore)
Class(Capitano)
```

Attributi di classe:

```
DataPropertyRange(orario rdf:string)
DataPropertyDomain(orario Gara)
SubClassOf(Gara dataMinCardinality(1 orario))
FunctionalDataProperty(orario)

DataPropertyRange(giorno rdf:date)
DataPropertyDomain(giorno Gara)
SubClassOf(Gara dataMinCardinality(1 giorno))
FunctionalDataProperty(giorno)

DataPropertyRange(citta rdf:string)
DataPropertyDomain(citta Gara)
SubClassOf(Gara dataMinCardinality(1 citta))
```

```
FunctionalDataProperty(citta)

DataPropertyRange(nomeCorridore rdf:string)
DataPropertyDomain(nomeCorridore Corridore)
SubClassOf(Corridore dataMinCardinality(1 nomeCorridore))
FunctionalDataProperty(nomeCorridore)

DataPropertyRange(daAnno rdf:integer)
DataPropertyDomain(daAnno Capitano)
SubClassOf(Capitano dataMinCardinality(1 daAnno))
FunctionalDataProperty(daAnno)

DataPropertyRange(nomeSquadra rdf:string)
DataPropertyDomain(nomeSquadra Squadra)
SubClassOf(Squadra dataMinCardinality(1 nomeSquadra))
FunctionalDataProperty(nomeSquadra)

DataPropertyRange(annoFondazione rdf:integer)
DataPropertyDomain(annoFondazione Squadra)
SubClassOf(Squadra dataMinCardinality(1 annoFondazione))
FunctionalDataProperty(annoFondazione)
```

Generalizzazioni:

```
SubClassOf(GaraDisputata Gara)
SubClassOf(Capitano Corridore)
```

Associazioni:

```
ObjectPropertyDomain(iscritto Corridore)
ObjectPropertyRange(iscritto Gara)

ObjectPropertyDomain(appartiene Corridore)
ObjectPropertyRange(appartiene Squadra)
SubClassOf(Corridore ObjectMinCardinality(1 appartiene))
FunctionalObjectProperty(appartiene)
SubClassOf(ObjectMinCardinality(1
inverseObjectPropertyOf(appartiene))  Corridore)

ObjectPropertyDomain(prendeParte Corridore)
ObjectPropertyRange(prendeParte GaraDisputata)
SubClassOf(ObjectMinCardinality(1
inverseObjectPropertyOf(prendeParte)) Corridore)

ObjectPropertyDomain(capitanata Squadra)
ObjectPropertyRange(capitanata Capitano)
SubClassOf(Corridore ObjectMinCardinality(1 capitanata))
FunctionalObjectProperty(capitanata)
SubClassOf(ObjectMinCardinality(1
InverseObjectPropertyOf(capitanata)) Corridore)
```

```
FunctionalObjectProperty(InverseObjectPropertyOf(capitanata))
```

Attributi di associazioni:

```
ObjectPropertyDataRange(posizione rdf:integer)
ObjectPropertyDataDomain(posizione prendeParte)
SubObjectPropertyOf(prendeParte
ObjectPropertyDataMinCardinality(1    posizione))
FunctionalObjectPropertyData(posizione)
```

Specializzazioni:

```
SubObjectPropertyOf(prendeParte iscritto)
```

Query per vincoli non esprimibili in DL-Lite

Non sono necessarie query per la verifica di vincoli non esprimibili in DL-Lite.

Query congiuntive per gli use case

Prendiamo il primo programma dello Use Case, *Ritirati*. Questo non può essere espresso tramite una conjunctive query a causa della presenza dell'operatore **NOT**, infatti la query SparSQL riluta essere:

```
SELECT      t.corridore
FROM        sparqltable (
                  SELECT ?corridore ?citta ?giorno ?orario
                  WHERE {
                        ?x rdf:type 'Corridore'.
                              ?x :iscritto ?y.
                        ?y rdf:type 'Gara'.
                        ?x :nomeCorridore ?corridore.
                        ?y :citta ?citta.
                        ?y :giorno ?giorno.
                        ?y :orario ?orario.
                  }
            ) t
WHERE       t.citta = 'viterbo' AND
            t.giorno = '2008-09-20' AND
            t.orario = '1840' AND
            t.corridore NOT IN (
                  SELECT      t.arrivato
                  FROM  sparqltable (
                        SELECT ?arrivato ?citta ?giorno ?orario
                        WHERE {
                              ?x rdf:type 'Corridore'.
                              ?x :prendeParte ?y.
```

```
                        ?y rdf:type 'GaraDisputata'.
                        ?x :nomeCorridore ?arrivato.
                        ?y :citta ?citta.
                        ?y :giorno ?giorno.
                        ?y :orario ?orario.
                    }
                ) t
                WHERE t.citta = 'viterbo' AND
                    t.giorno = '2008-09-20' AND
                    t.orario = '1840'
        )
```

Il secondo programma, *GareVinte*, invece è esprimibile tramite una conjunctive query e possiamo vederelo dalla seuente traduzione:

```
SELECT     t.citta, t.giorno, t.orario
FROM sparqltable(
           SELECT ?corridore ?citta ?giorno ?orario ?posizione
           WHERE {
                ?x rdf:type 'Corridore'.
                (?x :prendeParte ?y)  :posizione ?posizione.
                ?y rdf:type 'GaraDisputata'.
                ?x :nomeCorridore ?corridore.
                ?y :citta ?citta.
                ?y :giorno ?giorno.
                ?y :orario ?orario.
           }
     ) t
WHERE t.posizione = 1 AND
     t.corridore = 'corridore3'
```

Test delle queries

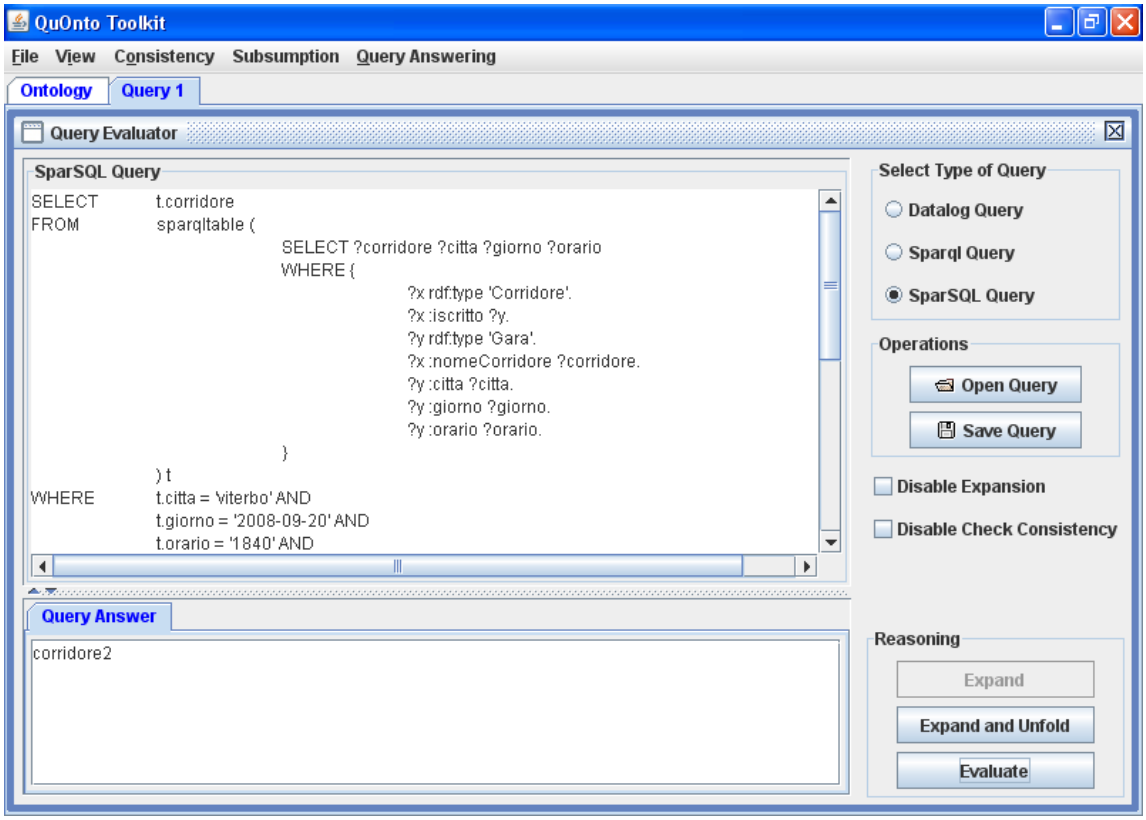
Al fine di testare le queries è stata creata la seguente ABox:

```
classassertion(C1 Corridore)
dataPropertyAssertion(nomeCorridore C1 corridore1)

classassertion(C2 Corridore)
dataPropertyAssertion(nomeCorridore C2 corridore2)

classassertion(C3 Corridore)
dataPropertyAssertion(nomeCorridore C3 corridore3)

classassertion(CA1 Capitano)
dataPropertyAssertion(nomeCorridore CA1 corridore1)
dataPropertyAssertion(daAnno CA1 2001)

classassertion(CA3 Capitano)
```

```
dataPropertyAssertion(nomeCorridore CA3 corridore3)
dataPropertyAssertion(daAnno CA3 2002)

classassertion(S1 Squadra)
dataPropertyAssertion(nomeSquadra S1 squadra1)
dataPropertyAssertion(annoFondazione S1 1998)

classassertion(S2 Squadra)
dataPropertyAssertion(nomeSquadra S2 squadra2)
dataPropertyAssertion(annoFondazione S2 2000)

classassertion(G1 Gara)
dataPropertyAssertion(orario G1 1840)
dataPropertyAssertion(giorno G1 2008-09-20)
dataPropertyAssertion(citta G1 viterbo)

classassertion(GD1 GaraDisputata)
dataPropertyAssertion(orario GD1 1840)
dataPropertyAssertion(giorno GD1 2008-09-20)
dataPropertyAssertion(citta GD1 viterbo)

classassertion(G2 Gara)
dataPropertyAssertion(orario G2 1040)
dataPropertyAssertion(giorno G2 2008-11-29)
dataPropertyAssertion(citta G2 roma)

classassertion(GD2 GaraDisputata)
dataPropertyAssertion(orario GD2 1040)
dataPropertyAssertion(giorno GD2 2008-11-29)
dataPropertyAssertion(citta GD2 roma)

objectPropertyAssertion(appartiene CA1 S1)
objectPropertyAssertion(appartiene C2 S1)
objectPropertyAssertion(appartiene CA3 S2)

objectPropertyAssertion(capitanata S1 CA1)
objectPropertyAssertion(capitanata S2 CA3)

objectPropertyAssertion(iscritto CA1 G1)
objectPropertyAssertion(iscritto C2 G1)
objectPropertyAssertion(iscritto CA3 G1)

objectPropertyAssertion(iscritto CA1 G2)
objectPropertyAssertion(iscritto CA3 G2)

objectPropertyAssertion(prendeParte CA1 GD1)
objectPropertyDataAssertion(posizione CA1 GD1 2)

objectPropertyAssertion(prendeParte CA3 GD1)
objectPropertyDataAssertion(posizione CA3 GD1 1)

objectPropertyAssertion(prendeParte CA1 GD2)
```

```
objectPropertyDataAssertion(posizione CA1 GD2 1)

objectPropertyAssertion(prendeParte CA3 GD2)
objectPropertyDataAssertion(posizione CA3 GD2 2)
```

Di seguito riporto gli output del toolkit QuOnto. Query *Ritirati*:



Query *GareVinte*: