Seminario di Ingegneria del Software

Studio di RacerPro per fare delle interrogazioni in OWL-DL. In particolare dovrei cercare di esprimere delle conjunctive queries interessanti

Alessio De Gaetanis

INTRODUZIONE ... **4**

Capitolo 1 - Sistema Racer ... **5**
1.1 Introduzione a RacerPro ... 5
1.1.1 RacerPro come Sistema di Ragionamento Web Semantico ... 5
1.1.2 RacerPro come Sistema di Logica Descrittiva ... 5
1.1.3 RacerPro come combinazione di Logiche Descrittive e Algebre Relazionali Specifiche ... 6
1.2 RacerPorter, il client di RacerPro ... 7
1.3 Basi di conoscenza in RacerPro ... 9
1.3.1 Definizione dei concetti ... 9
1.3.2 Assiomi di concetto e TBoxes ... 11
1.3.3 Dichiarazioni di ruolo ... 12
1.3.4 Dominio concreto ... 14
1.3.5 Attributi di dominio concreto ... 15
1.3.6 Asserzioni individuali e ABoxes ... 15
1.3.7 Modi di inferenza ... 16

Capitolo 2 - Modellazione di Logiche Descrittive con RacerPro ... **18**
2.1 Rappresentazioni di dati con Logiche Descrittive ... 18
2.2 Nominali o Domini Concreti ... 18
2.3 Assunzione di mondo aperto ... 20
2.4 Assunzione di mondo chiuso ... 20
2.5 Assunzione di nome unico ... 20
2.6 Differenza in espressività tra Linguaggi di Query e Linguaggi di Concetto ... 20
2.7 Interfaccia OWL ... 21

Capitolo 3 - nRQL (new Racer Query Language) ... **22**
3.1 Introduzione ... 22
3.2 Atomi di query, oggetti, individui e variabili ... 24
3.2.1 Atomi di query di concetto (Concept Query Atoms) ... 25
3.2.2 Atomi di query di ruolo (Role Query Atoms) ... 27
3.2.3 Atomi di query di vincolo (Constraint Query Atoms) ... 29
3.2.4 Atomi di query SOME-AS (SOME-AS Query Atoms) ... 32
3.2.5 Atomi di query ausiliare (Auxiliary Query Atoms) ... 32
3.3 Operatori Query Head Projection per recuperare valori noti ... 33
3.3.1 Operatore di proiezione dell'attributo ... 34
3.3.2 Operatore di proiezione dei valori noti ... 35
3.3.3 Osservazioni sulla completezza ... 35
3.3.4 Recuperare i riempitivi di valore noto delle OWL datatype properties ... 35

3.3.4 Recuperare valori noti di OWL Annotation Properties ... 36
3.4 Query Complesse ... 37
3.4.1 Costruttore AND – Query Congiuntive ... 38
3.4.2 Costruttore UNION ... 38
3.4.3 Costruttore NEG – La negazione come costruttore di fallimento ... 40
3.4.4 Costruttore PROJECT-TO: operatore di proiezione per il corpo della query 42
3.5 Query definite ... 43
3.6 Arricchimento della ABox con semplici regole ... 45
3.7 Query su TBox complesse ... 46
3.8 Sintassi formale di una query ... 47
3.9 Motore di processamento della query nRQL ... 49
3.9.1 Modi di processamento della query di nRQL ... 49
3.9.2 Il ciclo di vita di una query ... 51
3.9.3 Configurare il grado di completezza di nRQL ... 52
3.10 LUBM Benchmark ... 54

Capitolo 4 - Casi di studio ... 57
4.1 Prima base di conoscenza ... 57
4.2 Seconda base di conoscenza ... 62
4.3 Esame del 19/12/2002 – Compito A ... 63
4.4 Esame del 19/12/2002 – Compito B ... 65
4.5 Esame del 19/12/2005 – Compito A ... 67
4.6 Esame del 19/12/2005 – Compito B ... 69

Conclusioni ... 71

Riferimenti bibliografici ... 72

INTRODUZIONE

Il seguente lavoro prevede lo studio del linguaggio nRQL, il nuovo linguaggio di query di Racer, per cercare di esprimere della *conjunctive query* interessanti.
Partendo da una prima analisi del sistema RacerPro, analizzerò quali sono le sue caratteristiche e le funzioni supportate dall'applicazione client RacerPorter, che permette di interagire con il server. Vedremo inoltre come si possono creare delle ontologie con questo sistema, in modo tale da poter verificare come il sistema risponde alle query presenti nelle slide dell'articolo *"Epistemic First-Order Queries over Description Logic Knowledge Base"* di Giuseppe De Giacomo (lavoro svolto con Diego Calvanese, Domenico Lembo, Maurizio Lenzerini e Riccardo Rosati). Per esprimere queste query studieremo prima il linguaggio nRQL, analizzando tutti i suoi costrutti, a partire dal semplice atomo (di concetto, di ruolo, di vincolo e altri ausiliari), passando per operatori query head projection per recuperare valori noti, fino ad arrivare alla struttura di query più complesse (costruite tramite gli operatori AND, UNION, NEG, PROJECT-TO).
Infine cercherò di esprimere delle query su delle ontologie espresse in OWL, sviluppate da Emma Di Pasquale e Teresa Raguso utilizzando il sistema Protegè in un lavoro precedente.

CAPITOLO 1
Sistema Racer

1.1 Introduzione a RacerPro

Il termine RacerPro sta per Renamed Abox and Concept Expression Professional.
RacerPro può essere utilizzato per diversi scopi, come:

- Sistema di Ragionamento Web Semantico;
- Sistema di Logica Descrittiva;
- Combinazione di Logiche Descrittive e Algebre Relazionali Specifiche.

1.1.1 RacerPro come Sistema di Ragionamento Web Semantico

RacerPro è in grado di processare documenti OWL (Web Ontology Language)-Lite (è la versione del linguaggio OWL più semplice in grado di definire solo gerarchie di classi e vincoli poco complessi) o documenti OWL-DL (è la versione intermedia, che offre un potere espressivo più elevato di OWL-Lite e garantisce sia la completezza computazionale, ovvero permette di calcolare tutte le conclusioni, sia la decidibilità, ovvero tutte le computazioni si concludono in un tempo finito.
I servizi offerti da RacerPro per ontologie OWL e descrizioni di dato RDF sono:

- Controllare la consistenza di un ontologia OWL e di un insieme di descrizioni di dato.
- Trovare relazione di sottoclasse implicite indotte dalla dichiarazione nell'ontologia.
- Trovare sinonimi per le risorse (sia per classi che per nomi d'istanze).
- Client HTTP per recuperare risorse importate dal web. Risorse multiple possono essere importate in una ontologia.
- Supporta l'uso adattabile della risorsa computazionale (recupera i prossimi n risultati della query): le risposte che richiedono poche risorse computazionali sono consegnate per prime, poi l'applicazione utente decide se vale la pena o meno computare tutte le risposte.

1.1.2 RacerPro come Sistema di Logica Descrittiva

RacerPro è un sistema di rappresentazione della conoscenza che implementa il calcolo basato su tableau altamente ottimizzato per una logica descrittiva molto espressiva. Offre servizi di ragionamento per TBoxes multipli e ABoxes multipli. Il sistema implementa la logica descrittiva *ALCQHIR+* anche nota come *SHIQ*. Questa è la logica di base di *ALC* con restrizioni di numero qualificative, gerarchie di ruolo, ruoli inversi e ruoli transitivi.

Oltre a questi caratteristiche di base, RacerPro offre anche installazioni per il ragionamento algebrico includendo domini concreti per offrire:

- Restrizione di min/max sui interi;
- Equazione di polinomi lineari sui reali o cardinali con relazioni di ordinamento,
- Uguaglianze e disuguaglianze di stringhe.

RacerPro supporta la specifica di assiomi terminologici generali. Una Tbox può contenere inclusioni di concetto generali (GCIs), che indicano la relazione di sussunzione tra due termini di concetti. Supporta inoltre definizioni multiple o definizioni cicliche di concetti.

RacerPro perfeziona anche la maggior parte delle funzioni utilizzate in vecchi sistemi di rappresentazione della conoscenza (come KRSS).

Data una TBox, possono essere richieste vari tipi di queries. Basato sulla semantica logica del linguaggio di rappresentazione, diversi tipi di queries sono definiti come problemi di inferenza. Ad esempio:

1. Consistenza di un concetto: l'insieme di oggetti è descritto da un insieme vuoto?
2. Sussunzione di concetti: c'è una relazione di sottoinsieme tra l'insieme di oggetti descritti dai due concetti?
3. Inconsistenza tra concetti.
4. Determina la relazione padre-figlio di un concetto. Considerando tutti i nomi di concetto in una TBox, la relazione padre (o quella figlio) definisce una struttura a grafo alla quale ci si riferisce come tassonomia (nota che qualche autore usa il nome tassonomia come sinonimo per ontologia).

Nel caso sia definita anche una ABox, è possibile effettuare i seguenti tipi di queries:

1. Controlla la consistenza di una ABox.
2. Testa le istanze.
3. Recupera le istanze.
4. Recupero delle tuple di individui (istanze) che soddisfano certe condizioni.
5. Computazione dei tipi diretti di un individuo, che consiste nel trovare i nomi del concetto più specifico da una TBox dei quali un determinato individuo è un'istanza.
6. Computazione dei riempitivi (fillers) di un ruolo con riferimento ad un individuo.
7. Controlla se i vincoli di un certo dominio concreto sono soddisfatti.

RacerPro offre un linguaggio di query ben definito semanticamente detto **nRQL** (new Racer Query Language), che incorpora anche delle agevolazioni speciali per effettuare delle query in OWL.

1.1.3 RacerPro come combinazione di Logiche Descrittive e Algebre Relazionali Specifiche

RacerPro combina ragionamenti di logiche descrittive con ragionamenti riguardo a relazioni spaziali o temporali dentro nRQL, il linguaggio di query di un ABox. I

collegamenti per le variabili di una query che sono determinati dal ragionamento della ABox possono essere testati più velocemente rispetto a una associata rete di vincoli di relazioni spaziali (o temporali).
Sebbene RacerPro è uno dei primi sistemi che supporta questo tipo di ragionamento in combinazione con le logiche descrittive (o OWL), ci si aspetta che gli sforzi della standardizzazione internazionale riguardino anche questi importanti costrutti di rappresentazione nel prossimo futuro.

1.2 RacerPorter, il client di RacerPro

RacerPro è un server per logiche descrittive o servizi d'inferenza OWL. Con esso si possono implementare progetti di capacità industriale, come la ricerca su basi di conoscenza e lo sviluppo di applicazioni complesse.

Il programma client che fornisce all'utente un'interfaccia grafica per RacerPro è **RacerPorter**. Esso utilizza un'interfaccia di rete TCP/IP per connettersi a uno o a più server RacerPro e aiuta l'utente a gestirli: infatti permette di caricare basi di conoscenza, scegliere tra differenti tassonomie, ispezionare le istanze, visualizzare ABox e TBox, manipolare il server e molto altro.

Una volta avviato il server RacerPro, si deve avviare il programma RacerPorter e premere il pulsante *Connect* per connettere il programma client al server.

Le sei text area in cima all'applicazione RacerPorter costituiscono il cosiddetto *"state display"*, che indica lo stato corrente di RacerPorter. Selezionando per esempio una concetto dalla lista **Concepts**, viene aggiornata la text area corrente relativa nello state display. La stessa cosa avviene automaticamente per tutti gli altri campi. Lo stato corrente è usato per indicare l'oggetto corrente che deve essere ispezionato o interrogato.

E' da notare che per cambiare la ABox (TBox) dal tab **ABoxes** (**TBoxes**) bisogna usare il pulsante *Set Racer ABox (TBox).*

Nel tab **Shell**, è possibile effettuare una query in linguaggio nRQL all'interno della text area *Command* e le risposte vengono visualizzate nella textarea *RacerPro Log.*

Nel tab **Taxonomy**, una volta selezionato un concetto della tassonomia si possono recuperare le istanze del concetto selezionato premendo il pulsante *Concept Query*.

Selezionando un elemento dalla tab **Individuals**, e premendo su *Direct Types* o su *All Type* verrà elencati i tipi dell'elemento selezionato, che verranno evidenziati in rosso nel grafo riportato nel tab **Taxonomy.**

Selezionando il tab **Network** verrà mostrata la struttura della ABox focalizzando l'attenzione sull'elemento selezionato.

Il tab **Roles** è l'unico che permette di effettuare una selezione multipla. L'ultimo ruolo selezionato è quello corrente.

E infine nel tab **Queries** o **Rules** si può selezionare una query nRQL dalla lista, e poi usare i pulsanti per applicare un comando sulla query selezionata.

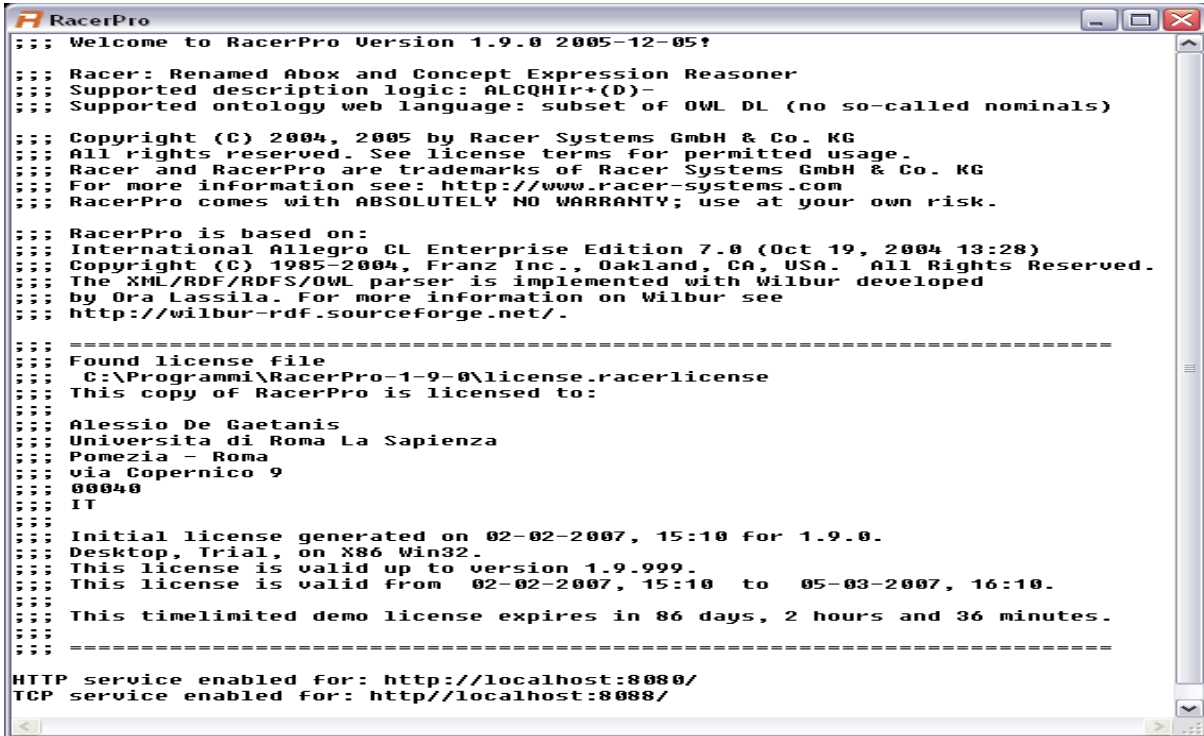


Fig. 1 - Interfaccia del server RacerPro

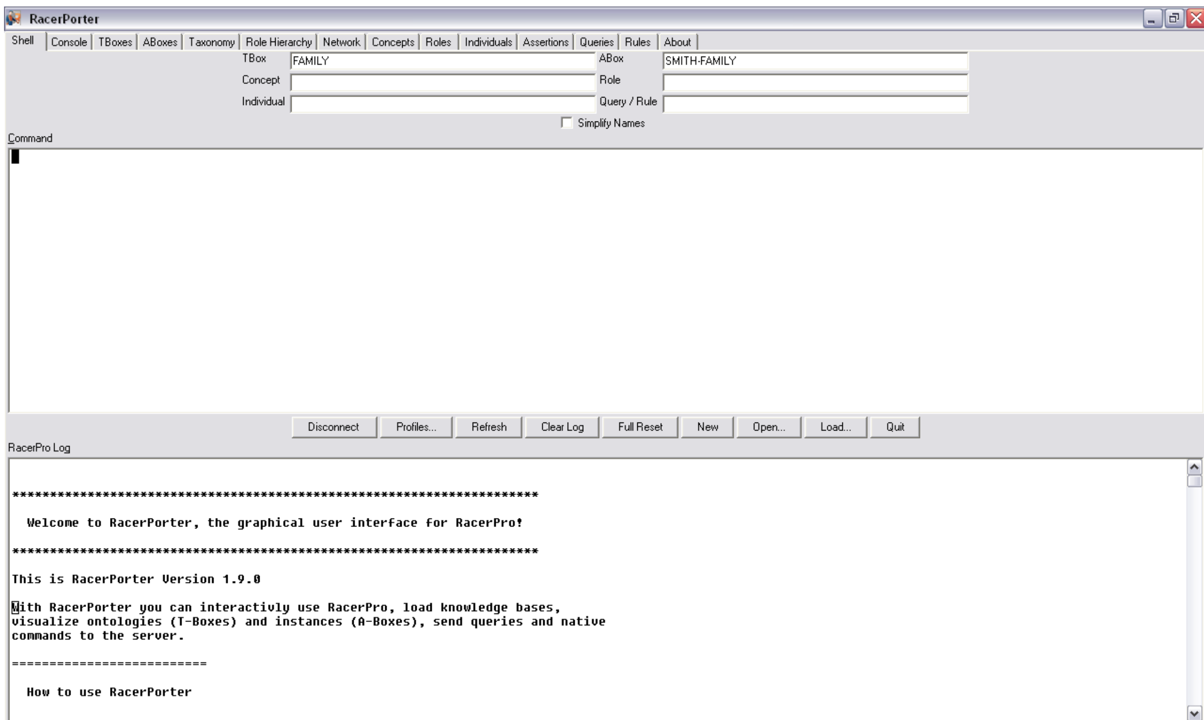


Fig. 2 - Interfaccia del client RacerPorter

1.3 Basi di conoscenza in RacerPro

1.3.1 Definizione dei concetti

Nei sistemi basati sulle logiche descrittive una base di conoscenza è composta da una TBox e da una ABox. La conoscenza concettuale è rappresentata nella TBox e la conoscenza riguardo le istanze del dominio è rappresentata nell'ABox.
Stabiliamo innanzitutto la lista di simboli che tratteremo nel seguente paragrafo:

C	Concept term	*name*	Name of any sort
CN	Concept name	S	List of Assertions
IN	Individual name	GNL	List of group names
ON	Object name	LCN	List of concept names
R	Role term	*abox*	ABox object
RN	Role name	*tbox*	TBox object
AN	Attribute name	n	A natural number
ABN	ABox name	*real*	A real number
TBN	TBox name	*integer*	An integer number
KBN	knowledge base name	*string*	A string

Il contenuto della TBox in RacerPro include la modellazione concettuale dei concetti e dei ruoli, che consiste nello specificare due insiemi distinti: uno riguardo ai nomi dei concetti atomici (insieme *C*) e uno riguardo ai nomi dei ruoli (insieme *R*).
A partire dall'insieme C, possono essere costruiti termini di concetto complessi utilizzando diversi operatori, che sono riportati di seguito:

```
C ⟶  CN                  |
     *top*               |
     *bottom*            |
     (not C)             |
     (and C1 ... Cn)     |
     (or C1 ... Cn)      |
     (some R C)          |
     (all R C)           |
     (at-least n R)      |
     (at-most n R)       |
     (exactly n R)       |
     (at-least n R C)    |
     (at-most n R C)     |
     (exactly n R C)     |
     (a AN)              |
     (an AN)             |
     (no AN)             |
     CDC

R ⟶  RN                  |
     (inv RN)
```

I **termini Booleani** costruiscono concetti usando questi operatori:

	DL notation	RacerPro syntax
Negation	$\neg C$	`(not C)`
Conjunction	$C_1 \sqcap \ldots \sqcap C_n$	`(and` C_1 `...` C_n`)`
Disjunction	$C_1 \sqcup \ldots \sqcup C_n$	`(or` C_1 `...` C_n`)`

Qualified restriction indicano che i riempitivi (fillers) di ruolo devono essere di un certo concetto. **Value restriction** assicura che il tipo di tutti i riempitivi del ruolo sia del concetto specificato, mentre **exists restriction** richiede che c'è qualche riempitivo di quel ruolo che è un'istanza del concetto specificato.

	DL notation	RacerPro syntax
Exists restriction	$\exists R.C$	`(some R C)`
Value restriction	$\forall R.C$	`(all R C)`

Le **Number restriction** invece specificano un lower bound, un upper bound oppure un numero esatto per ogni riempitivo di ruolo. Tali restrizioni possono essere usate solo per ruoli non transitivi o per ruoli che non hanno sotto-ruoli transitivi.

	DL notation	RacerPro syntax
At-most restriction	$\leq n\ R$	`(at-most n R)`
At-least restriction	$\geq n\ R$	`(at-least n R)`
Exactly restriction	$= n\ R$	`(exactly n R)`
Qualified at-most restriction	$\leq n\ R.C$	`(at-most n R C)`
Qualified at-least restriction	$\geq n\ R.C$	`(at-least n R C)`
Qualified exactly restriction	$= n\ R.C$	`(exactly n R C)`

Inoltre ci sono altri due concetti dichiarati implicitamente in ogni TBox: il concetto **top** (T) che indica il concetto più grande nella gerarchia (denotato in RacerPro con *top*) e il concetto **bottom** (F) che indica il concetto di inconsistenza, che è un sotto-concetto di tutti gli altri concetti (denotato in RacerPro con *bottom*).
Invece i **Concrete domain concepts** indicano le restrizioni di predicato concreto per riempitivi d'attributo. RacerPro supporta tre predicati unari per attributi interi (min, max , equal), sei predicati n-ari per gli attributi reali (>,>=,<,<=,<,<>), un predicato esistenziale unario con due varianti sintattiche (a o an), e uno di negazione (no). Le restrizioni per gli attributi di tipo real, devono essere nella forma di disequazioni lineari (con una relazione di ordine), dove il nome dell'attributo gioca il ruolo della variabile.

```
CDC ⟶   (min AN integer)                          |
        (max AN integer)                          |
        (equal AN integer)                        |
        (equal AN AN)                             |
        (divisible AN cardinal)                   |
        (not-divisible AN cardinal)               |
        (> aexpr aexpr)                           |
        (>= aexpr aexpr)                          |
        (< aexpr aexpr)                           |
        (<= aexpr aexpr)                          |
        (<> aexpr aexpr)                          |
        (= aexpr aexpr)                           |
        (string= AN string)                       |
        (string<> AN string)                      |
        (string= AN AN)                           |
        (string<> AN AN)
string ⟶  " letter* "
aexpr ⟶   AN                      | AN must be of type real
          real                                    |
          (+ aexpr1 aexpr1*)                      |
          aexpr1
```

Di seguito sono riportate specifiche espressioni per predicati:

```
aexpr1 ⟶        aexpr2
        |       aexpr3
        |       aexpr5
aexpr2 ⟶        real
        |       AN                    (AN of type real or complex)
        |       (* real AN)           (AN of type real or complex)
aexpr3 ⟶        real
        |       AN                    (AN of type complex)
        |  (* integer aexpr4 aexpr4*)
aexpr4 ⟶        AN                    (AN of type complex)
        |       (expt AN n)           (AN of type complex)
aexpr5 ⟶        integer
        |       AN                    (AN of type cardinal)
        |       (* integer AN)        (AN of type cardinal)
```

1.3.2 Assiomi di concetto e TBoxes

RacerPro supporta diversi tipi di assiomi di concetto, che sono riportati a seguito in notazione DL con la relativa sintassi di RacerPro:

- **General concept inclusion:** Mostra lo stato della relazione di sussunzione tra due concetti.

DL notation: $C_1 \sqsubseteq C_2$
RacerPro syntax: (implies C_1 C_2)

- **Concept equation:** Mostra lo stato di equivalenza tra due concetti.

DL notation: $C_1 \doteq C_2$
RacerPro syntax: (equivalent C_1 C_2)

- **Concept disjointness axioms:** Mostra la disgiunzione tra diversi concetti.

DL notation: $C_1 \sqsubseteq \neg(C_2 \sqcup C_3 \sqcup \cdots \sqcap C_n)$
$C_2 \sqsubseteq \neg(C_3 \sqcup \cdots \sqcap C_n)$
$C_{n-1} \sqsubseteq \neg C_n$
RacerPro syntax: (disjoint C_1 ... C_n)

L'uguaglianza di concetto C1 = C2 può essere espresso da due inclusioni di concetto generali: C1 c C2 e C2 c C1. Anche la disjointness può essere espressa da inclusioni generali di concetto.

I seguenti assiomi di concetto implementano tipi speciali di inclusioni e di uguaglianza di concetto

- **Primitive concept axioms:** Mostra la relazione di sussunzione tra un nome di concetto e un termine di concetto.

DL notation: $(CN \sqsubseteq C)$
RacerPro syntax: (define-primitive-concept CN C)

- **Concept definitions:** Mostra l'equivalenza tra un nome di concetto e un termine di concetto:

DL notation: $(CN \doteq C)$
RacerPro syntax: (define-concept CN C)

Assiomi di concetto potrebbero anche essere ciclici. Possono anche essere riferimenti a concetti che saranno introdotti con define-concept o define-primitive-concept in assiomi successivi. La terminologia di una TBox di RacerPro potrebbe contenere anche diversi assiomi per un singolo concetto. Così se è dato un secondo assioma allo stesso concetto, questo viene aggiunto e non sovra scrive il primo assioma.

1.3.3 Dichiarazioni di ruolo

A differenza degli assiomi di concetto, le dichiarazioni di ruolo sono uniche in RacerPro. Se esiste già una dichiarazione per un nome di ruolo nella base di conoscenza, e viene data una seconda dichiarazione per ruolo, viene segnalato un errore.

L'insieme dei ruoli (*R*) include l'insieme dei features (*F*) e l'insieme dei ruoli transitivi (*R+*). Gli insiemi *F* e *R+* sono disgiunti. Tutti i ruoli in una TBox possono essere organizzati in una gerarchia dei ruoli. L'inverso di un nome di ruolo *RN* può essere dichiarato tramite parola chiave `:inverse` oppure riferito ad esso `(inv RN)`.

Features (detti anche attributi) restringe un ruolo ad essere un ruolo funzionale (ogni individuo può avere solo un riempitivo per questo ruolo).

Ruoli transitivi sono ruoli transitivamente chiusi. Se due coppie di individui IN1 e IN2, e IN2 e IN3 sono legati tramite un ruolo transitivo R, allora anche IN1 e IN3 sono legati tramite R.

Gerarchie di ruolo definiscono una relazione di super e sotto ruolo tra i diversi ruoli. Se R1 è un super-ruolo di R2, allora tutti gli individui che conservano R2, allora conservano anche R1.

Nella corrente implementazione le relazioni di super-ruolo specificato non possono essere cicliche. Se un ruolo ha un super-ruolo, le sue proprietà non sono in ogni caso ereditate dal sotto-ruolo. Le proprietà di un ruolo dichiarato indotte dal suo super-ruolo sono mostrate in figura. La tabella deve essere letta come segue: se per esempio un ruolo RN1 è dichiarato come un semplice ruolo ed ha un feature RN2 come super-ruolo, allora RN1 sarà un feature di se stesso.

		Superrole $RN_2 \in$		
		$\mathcal{R}$	$\mathcal{R}^+$	$\mathcal{F}$
Subrole RN_1	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{F}$
declared as	$\mathcal{R}^+$	$\mathcal{R}^+$	$\mathcal{R}^+$	-
element of:	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$

La combinazione di un feature avente un super-ruolo transitivo non è permesso, e i features non possono essere transitivi. Nota che i ruoli transitivi e i ruoli con sotto-ruoli transitivi non possono essere usati in restrizioni di numero.

RacerPro non supporta a differenza di KRSS la definizione di termini di ruolo, ma permette comunque di definire un ruolo che risulta essere la congiunzione di altri ruoli, usando la gerarchia di ruolo. Infatti la dichiarazione di un ruolo `(define-primitive-role RN (and RN1 RN2))` può essere approssimato in RacerPro come `(define-primitive-role RN :parents (RN1 RN2))`.

RacerPro offre per ogni ruolo la dichiarazione del dominio e la restrizione del range. Queste ultime possono essere espresse sia tramite inclusioni di concetto generali, o dichiarate tramite parole chavi `:domain` e `:range`.

RacerPro Syntax	DL notation
(define-primitive-role RN :domain C)	$(\exists\ RN.\top) \sqsubseteq C$
(define-primitive-role RN :range D)	$\top \sqsubseteq (\forall\ RN.D)$

1.3.4 Dominio concreto

RacerPro supporta ragionamenti su numeri naturali (N), interi (I), reali (R), complessi(C) e stringhe. Per differenti insiemi, diversi tipi di predicato sono supportati.

N → disuguaglianze lineari con vincoli di ordinamento e coefficienti interi
Z → vincoli d'intervallo
R → disuguaglianze lineari con vincoli di ordinamento e coefficienti razionali
string → uguaglianze e disuguaglianze

Per convenienza degli utenti, i coefficienti razionali possono essere specificati in notazione floating point. Essi sono automaticamente trasformati nelle loro equivalenze razionali (per esempio 0.75 viene trasformato in 3/4).
I nomi per i valori del dominio concreto sono detti *oggetti*. L'insieme di tutti gli oggetti è indicato con il simbolo *O*. Gli individui possono essere associati a oggetti tramite i cosiddetti *nomi d'attributo* (o semplicemente *attributi*). Notare che l'insieme di tutti gli attributi *A* deve essere disgiunto dall'insieme dei ruoli (e dei features). L'insieme degli attributi, come anche quello dei concetti atomici e dei ruoli vengono riportati nella segnature della TBox. Di seguito è riportato un esempio di estensione della TBox family.

```
...
(signature
      :atomic-concepts (... teenager)
      :roles (...)
      :attributes ((integer age)
        (real temperature-celsius)
        (real temperature-fahrenheit)))
...
(equivalent teenager (and human (min age 16)))
(equivalent old-teenager (and human (min age 18)))
(equivalent human-with-fever (and human (>= temperature-celsius 38.5))
(equivalent seriously-ill-human(and human (>= temperature-celsius 42.0)))
...
```

In questo caso RacerPro determina che il concetto `seriously-ill-human` è sussunto da `human-with-fever`.
Si potrebbe aggiungere la seguente dichiarazione alla base di conoscenza per rendere effettiva la relazione tra i due attributi `temperature-fahrenheit` e `temperature-celsius`:

```
(implies top (= temperature-fahrenheit
(+ (* 1.8 temperature-celsius) 32))).
```

Se viene definito un concetto `seriously-ill-human-1` come

```
(equivalent seriously-ill-human-1
(and human (>= temperature-fahrenheit 107.6)))
```

RacerPro riconosce la relazione di sussunzione con human-with-fever e la relazione sinonimo con seriously-ill-human.
Nella ABox, è possibile settare vincoli tra individui. Per esempio:

```
...
(signature
     :atomic-concepts (... teenager)
     :roles (...)
     :attributes (...)
     :individuals (eve doris)
     :objects (temp-eve temp-doris))
...
(constrained eve temp-eve temperature-fahrenheit)
(constrained doris temp-doris temperature-celsius)
(constraints
(= temp-eve 102.56)
(= temp-doris 39.5))
```

In questo caso si afferma che `eve` è relazionato tramite l'attributo `temperature-fahrenheit` all'oggetto `temp-eve`. Il primo vincolo `(= temp-eve 102.56)` specifica che l'oggetto `temp-eve` è uguale a 102.56.

1.3.5 Attributi di dominio concreto

Gli attributi sono considerati come "typed" poiché possono assumere valori di tipo `cardinal`, `integer`, `real`, `complex` o `string`. Lo stesso attributo non può essere utilizzato all'interno della stessa TBox in modo tale che due tipi vengano applicati (per esempio non sono permessi `(min has-age 18)` e `(>= has-age 18)`). Se il tipo di un attributo non è dichiarato esplicitamente, viene derivato implicitamente dal suo uso nella TBox/ABox. Un attributo e il suo tipo può essere dichiarato o nella signature o tramite la forma `define-concrete-domain-attribute`.Se un attributo è dichiarato essere di tipo `complex` allora può essere utilizzato in (dis)-uguaglianze lineari. Comunque se un attributo è dichiarato essere di tipo `real` o `integer`, è un errore utilizzarlo in termini per polinomi non lineari. Similmente, un attributo di tipo `integer` non può essere utilizzato neanche in un termine per polinomi lineari. Se i coefficienti sono interi, allora il tipo `cardinal` può essere usato in un polinomio lineare. Inoltre attributi di tipo `string` non possono essere usati su polinomi, e quelli non-string non possono essere utilizzati in vincoli per stringhe.

1.3.6 Asserzioni individuali e ABoxes

Un ABox contiene asserzioni riguardo individui. L'insieme degli individui *I* è la signature della ABox. L'insieme degli individui deve essere disgiunto da quello dei nomi di concetto e da quello dei nomi di ruolo. Ci sono quattro tipi di asserzioni:

Asserzioni di concetto con `instance` indica che un individuo *IN* è un'istanza di uno specifico concetto *C*.

Asserzioni di ruolo con `related` indica che un individuo *IN1* è un riempitivo di ruolo per un ruolo *R* rispetto ad un individuo *IN2*.

Asserzioni d'attributo con `constrained` indica che un oggetto ON è un riempitivo per un ruolo *R* rispetto ad un individuo *IN*.

Vincoli con `constraints` indica che le relazioni tra oggetti del dominio concreto.

RacerPro conserva l'*assunzione di nome unico*, che significa che tutti i nomi di individui usati in una ABox si riferiscono a oggetti di dominio distinti, quindi due nomi non possono riferirsi allo stesso oggetto di dominio. Tale assunzione non è mantenuta per nomi di oggetti.
Nel sistema RacerPro ogni ABox si riferisce ad una TBox. Le asserzioni di concetto presenti nella ABox sono interpretati rispetto agli assiomi di concetto dati nella TBox di riferimento. Le asserzioni di ruolo sono anche esse interpretate in accordo alle dichiarazioni che stanno nella TBox. Quando un'ABox viene costruita, la TBox a cui si riferisce dovrebbe gia esistere. La stessa TBox può essere utilizzata da diverse ABoxes. Se non viene dichiarata nessuna signature nella TBox, le asserzioni nella ABox potrebbero utilizzare nomi nuovi per i ruoli e per i concetti che non sono menzionati nella TBox.

1.3.7 Modi di inferenza

Dopo la dichiarazione della TBox e della ABox, RacerPro può essere istruito a rispondere a delle query. Il processamento della KB per rispondere ad una query potrebbe richiedere un pò di tempo. Il modo d'inferenza standard di RacerPro assicura il seguente comportamento: a seconda del tipo di query, RacerPro cerca di essere tanto intelligente quanto possibile per minimizzare localmente il tempo di computazione (modo d'inferenza *lazy*). Per esempio, per rispondere ad una query di sussunzione data una TBox, non è necessario classificare la TBox. Comunque, una volta che una TBox viene classificata, rispondere a query di sussunzione per concetti atomici è giusto un attimo.
Inoltre per verificare se esiste un concetto atomico nella TBox che è inconsistente, non è richiesto che la TBox sia classificata. Nel modo d'inferenza *lazy* (modo di default), RacerPro evita computazioni che non riguardano la query corrente. In qualche situazione, comunque, per cercare di minimizzare globalmente il tempo di processamento potrebbe essere meglio classificare prima una TBox prima di rispondere ad una query (modo d'inferenza *eager*).

Una strategia simile è applicata se è richiesta la computazione dei tipi diretti di individui. RacerPro richiede come pre-condizione che la corrispondente TBox deve essere classificata. Se è abilitata la modalità di inferenza *lazy*, solo gli individui coinvolti in una query di tipi diretti sono realizzati.

Si raccomanda che la TBox e la ABox dovrebbero essere mantenuti in file separati, in modo tale che quando una ABox viene modificata, non c'è bisogno di ricomputare ogni volta la TBox. Comunque, se la TBox è posizionata nello stesso file, la nuova valutazione del file presumibilmente causa la ri-inizializzazione della TBox e gli assiomi devono essere

dichiarati di nuovo. Quindi per rispondere ad una query dell'ABox, saranno necessarie le recomputazioni riguardanti la TBox. Se bisogna testare diverse ABox, esse saranno probabilmente localizzate separatamente dalla Tbox associata, in modo da salvare il tempo di processamento.

Durante la fase di sviluppo di una TBox potrebbe essere vantaggioso invocare direttamente servizi d'inferenza. Per esempio, durante la fase di sviluppo di una TBox potrebbe essere utile controllare quali concetti atomici nella TBox sono inconsistenti invocando `check-tbox-coherence`.

CAPITOLO 2

Modellazione di Logiche Descrittive con RacerPro

2.1 Rappresentazioni di dati con Logiche Descrittive

Non è richiesto nulla per utilizzare un sistema di inferenza di logiche descrittive per memorizzare dati. In particolare, non c'è bisogno di specificare alcun tipo di informazione di gestione della memoria come nei database o anche nei sistemi di programmazione object-oriented.
Le logiche descrittive sono significative se i nomi dei concetti hanno definizioni nella TBox o se la ABox contiene descrizioni indefinite (come `(instance john (or french italian))`). Anche se è possibile rappresentare un database come una ABox, i sistemi di logiche descrittive non prevedono ne le transazioni ne la persistenza dei dati, e per assicurare la decidibilità nel caso generale, il linguaggio di query è in qualche senso meno espressivo del linguaggio di query del database relazionale come SQL. Quindi le grandi quantità di dati (che rappresentano informazioni definite) sono memorizzati in modo migliore nei database, mentre le tecnologie delle logiche descrittive (e del web semantico) entrano in gioco quando deve essere trattata l'informazione indefinita (informazioni disgiuntive).

2.2 Nominali o Domini Concreti

Per definire il concetto di *nominali* prendiamo in considerazione il seguente assioma che determina il concetto `human`:

```
(define-primitive-role ancestor-of :transitive t :inverse has-descendant)
(define-primitive-role has-descendant)
(equivalent human (some ancestor-of (one-of adam)))
(instance john human)
(related kain john has-descendant)
```

L'operatore `one-of` prende un individuo e costruisce un concetto la cui estensione contiene l'oggetto semantico a cui l'individuo `adam` è mappato. Allora l'estensione di `(one-of adam)` è un insieme singleton. Gli individui in concetti sono detti *nominali*.

RacerPro però non supporta nominali nella piena generalità, infatti sono supportati solo assiomi della forma

```
(equivalent (one-of i) c)
```

```
(equivalent (one-of i) (some r (one-of j)))
```

Questi assiomi sono molto importanti e corrispondono direttamente alle seguenti asserzioni della Abox:

```
(instance i c)
(related i j r)
```

In molti altri casi non sono richiesti i nominali in termini di concetto, perché lo stesso effetto può essere ottenuto utilizzando valori di dominio concreto. Per esempio, invece di utilizzare i nominali per rappresentare i colori di un semplice semaforo (`(one-of red green)`), si può utilizzare un ABox e il dominio concreto string per costruire un modello formale di attraversamento con semafori, come nel seguente caso:

```
(in-knowledge-base traffic-lights)

(define-concrete-domain-attribute color :type string)

(define-concept colorful-object
  (or (string= color "red")
       (string= color "green")))

(define-concept traffic-light
  (and (a color) colorful-object))

(instance traffic-light-1 traffic-light)
(instance traffic-light-2 traffic-light)
(instance traffic-light-3 traffic-light)
(instance traffic-light-4 traffic-light)

(constrained traffic-light-1 ?color-traffic-light-1 color)
(constrained traffic-light-2 ?color-traffic-light-2 color)
(constrained traffic-light-3 ?color-traffic-light-3 color)
(constrained traffic-light-4 ?color-traffic-light-4 color)

(constraints (string= ?color-traffic-light-1 ?color-traffic-light-3))
(constraints (string= ?color-traffic-light-2 ?color-traffic-light-4))
(constraints (string<> ?color-traffic-light-1 ?color-traffic-light-2))

(constraints (string<> ?color-traffic-light-2 "red"))

(constraint-entailed? (string= ?color-traffic-light-2 "green"))
(constraint-entailed? (string= ?color-traffic-light-4 "green"))
(constraint-entailed? (string= ?color-traffic-light-1 "red"))
(constraint-entailed? (string= ?color-traffic-light-3 "red"))
```

Le quattro query alla fine ritornano tutte true anche se l'informazione indefinita è indicata esplicitamente, perché se il colore di `color-traffic-light-2` non è `red`, allora è `green`, e di conseguenza, a causa degli altri vincoli, sono determinati anche i colori degli altri semafori. E' ovvio che invece di usare volori di stringhe, sarebbe stato più opportuno utilizzare i nominali. Comunque algoritmi di ragionamento ottimizzati per nominali, saranno parte della versione futura di RacerPro.

2.3 Assunzione di mondo aperto

Come gli altri sistemi di logica descrittiva, RacerPro utilizza l'assunzione di mondo aperto (OWA: Open World Assumption) per il ragionamento. Questo significa che quello che non può essere provato esser vero, non vuol dire che sia falso. Prendendo come esempio la TBox e la ABox `family` possiamo osservare che alla query

```
(individual-instance? alice (at-most 2 has-child))
```

il sistema risponde NIL, che non significa no, ma semplicemente che non può essere provato a causa dell'informazione data a RacerPro. L'assenza di informazione riguardo un terzo figlio non è interpretato come "non c'è" (come nel CWA: Closed-World Assumption). Infatti ci potrebbe essere un asserzione `(related alice william has-child)` che potrebbe essere aggiunto alla ABox solo più tardi. Quindi la risposta è corretta ma deve essere interpretato nel senso che "non può essere provato".

2.4 Assunzione di mondo chiuso

Nei sistemi di inferenza di logica descrittiva tipicamente non viene supportata l'assunzione di mondo chiuso (Closed-World Assumption) a causa della sua natura non-monotica e alla ambiguità riguardo a cosa chiudere. Comunque con l'interfaccia di RacerPro, gli utenti possono utilizzare anche l'assunzione di mondo chiuso locale (LCW: Local Closed-World). Il linguaggio di query nRQL permette di interrogare l'ABox usando la negazione come semantica di fallimento.

2.5 Assunzione di nome unico

RacerPro può essere istruito anche per utilizzare l'assunzione di nome unico (UNA: Unique Name Assumption), in modo tale che tutti gli individui usati nella ABox devono essere collegati a elementi differenti dell'universo, per esempio due individui non possono riferirsi allo stesso elemento di dominio. Quindi aggiungendo `(instance alice (at-most 1 has-child))`, il sistema non identifica `betty` e `charles`, ma rende l'ABox inconsistente. Per essere conforme con la semantica di OWL RacerPro non applica questa assunzione per default. Per utilizzarla basta aggiungere il seguente comando prima di fare qualsiasi query:

```
(set-unique-name-assumption t)
```

2.6 Differenza in espressività tra Linguaggi di Query e Linguaggi di Concetto

Il linguaggio di query di RacerPro (nRQL) è differente dal linguaggio di concetto. Infatti le catene dei comportamenti e i nominali sono supportati in query ma non nel linguaggio di concetto. Inoltre specifici predicati di dominio concreto possono essere usate nelle espressioni di query, che non sono supportate dal linguaggio di concetto.

2.7 Interfaccia OWL

RacerPro è in grado di leggere file RDF, RDFS e OWL.
L'informazione in un file RDF è rappresentata usando un ABox in modo tale che le triple sono rappresentate come il comando `related`:, per esempio, il soggetto (subject) della tripla è rappresentato come un individuo, la proprietà (property) come un ruolo e anche l'oggetto (object) come un individuo. La proprietà `rdf:type` sono trattate in maniera speciale, infatti le triple con una proprietà del genere sono rappresentate come asserzioni di concetto. Le triple in file RDF sono processati in un modo speciale, infatti sono rappresentate come assiomi della TBox. Se la proprietà è `rdf:type`, l'oggetto dovrebbe essere `rdf:Class` o `rdf:Property`. Questi comandi sono interpretati come dichiarazioni per nomi di concetto e di ruolo, rispettivamente. Tre tipi di assiomi sono supportati con le seguenti proprietà: `rdfs:subClassOf`, `rdfs:range` e `rdfs:domain`. Altre triple sono ignorate.

I file OWL sono processati in maniera simile. A riguardo vengono riportate soltanto alcune restrizioni nell'implementazione di RacerPro. L'assunzione di nome unico non può essere eliminata e le restrizioni di numero per attributi non sono supportate. Sono supportati solo datatype di base dello schema XML.

CAPITOLO 3

nRQL (new Racer Query Language)

3.1 Introduzione

nRQL (si pronuncia *Nercle*) è il linguaggio di query per Abox di RacerPro, e può essere usato per interrogare ABox e Tbox di RacerPro, documenti RDF e documenti OWL. Quindi è un linguaggio di query di ABox per DL espressive $ALCQHI_{R+}(D-)$, un linguaggio di query RDF e OWL.

nRQL permette la formulazione di *query congiuntive*. In una query nRQL, vengono usate le *variabili della query*, che saranno legate a quegli individui della ABox che soddisfano la query. Le query faranno uso di concetti arbitrari e termini di ruolo. Le Tbox (o definizioni di classi OWL) forniranno il vocabolario specifico del dominio che verrà utilizzato nelle query. Comunque tale linguaggio offre molto di più che semplici query congiuntive. Il comportamento di tale linguaggio può essere riassunto come segue:

- Query complesse sono costruite a partire da atomi di query.
  Il linguaggio permette di combinare atomi di query di concetti, di ruoli, di vincolo e SOME-AS in query più complesse utilizzando i costruttori di query and, union, neg e project-to.
- nRQL ha una sintassi ben definita e una pura semantica composizionale.
  Il linguaggio risulta facile da capire a quegli utenti di RacerPro che possiedono già una certa conoscenza della nozione di *conseguenza logica* (*logical entailment*)**.** Una variabile all'interno della query è legata a un individuo della ABox se quest'ultimo soddisfa la query. Il termine 'soddisfa' significa che la query risultante dalla sostituzione di tutte le variabili con i loro collegamenti è conseguenza logica della KB. Le dichiarazioni quantificate esistenzialmente possono essere fatte sfruttando la potenza espressiva delle espressioni di concetto di RacerPro (ad esempio se volessi vedere se esiste un riempitivo del ruolo `has-child` di `?x` tale che …, potrebbe essere usata una dichiarazione come `(?x (some has-child ...))`).
- E' disponibile la *negazione come insuccesso* (*NAF*) come anche la *classica negazione del vero.*
  NAF è molto utili per misurare il grado di completezza di un modellamento del dominio nella KB (per esempio potrebbe essere utile chiedere gli individui che sono note essere donne ma che non sono madri). nRQL è l'unico linguaggio di query OWL che permette l'esecuzione di queste query ("autoepistemic").

- Supporto speciale per interrogare la parte di dominio concreto di una ABox.
  nRQL permette di dichiarare *predicati di dominio concreto*, specificando condizioni di ricerca complesse su valori dell' attributo di un individuo di una ABox (per esempio se si vogliono ricercare tutti gli individui adulti, i valori dell'attributo has-age dovrà soddisfare il predicato di dominio concreto >= 18). Inoltre il linguaggio supporta anche *constraint checking* che permette di fare delle espressioni di predicato complesse (per esempio si posso richiedere tutte le coppie di individui che hanno al più 8 anni di differenza).
- Operatore di proiezione project-to per il corpo della query.
- nRQL è anche un potente linguaggio di query OWL e RDF, che è molto più espressivo dei tipici linguaggi di query RDF, e anche più espressivo in certi aspetti dei semi-ufficiali linguaggi di query OWL.
- Sono anche disponibili query TBox complesse. Ciò permette di cercare certi modelli di relazioni sub/superclassi in una tassonomia (di una TBox di RacerPro o di una KB OWL).
- Supporta anche query ibride.

Il linguaggio nRQL deve essere distinto dal *motore di processamento della query nRQL (nRQL query processing engine)*, che è una parte interna del server RacerPro e ha le seguenti caratteristiche:

- Ottimizzatore di euristiche basate sui costi.
- Agevolazione per le query definite.
- Un semplice *meccanismo di regole.* La maggior parte delle funzionalità delle query sono applicate anche alle regole.
- Query (e regole) sono mantenute come oggetti all'interno del motore, che garantisce la gestione del ciclo di vita delle query.
- Multi-processamento delle query: può rispondere a più query simultaneamente.
- Supporta differenti *modi di interrogazione (quering)*:
  - "Set at a time mode": la risposta alla query è consegnata in un unico insieme. Le API nRQL lavorano in maniera sincrona, ovvero sono bloccate (non disponibili) fino a che la risposta viene computata e ritornata al client.
  - "Tupla at a time mode": la risposta alla query è calcolata e consegnata in maniera incrementale, tupla per tupla. Le API in questo modo lavorano in maniera asincrona. Un client può richiedere una tupla aggiuntiva alla volta di una risposta. Il vero multi-processamento delle query è disponibile in questo modo.
    Questo modo lavora sia in maniera *lazy*, computa la prossima tupla solo su domanda, ma anche in modo *eager*, pre-computa le prossime tuple anche se esse non sono state richieste ancora dal client. Nel primo caso viene massimizzata la

disponibilità di nRQL, mentre nel secondo caso si ha il vantaggio che la richiesta di una tupla successiva di una certa query sarà più veloce se è stata già computata.

- *Grado di completezza* configurabile: nRQL offre un *modo completo* così come *vari modi incompleti* che differiscono dal grado completezza che essi raggiungono. Un modo incompleto restituirà un sotto-insieme della risposta alla query rispetto a quello che sarà restituito dal modo completo.
- *Risorse runtime* configurabili: il numero di tuple di risposte restituite possono essere limitate, può essere settato un timeout, e sono disponibili diversi modi incompleti. Possono essere escluse anche le permutazioni delle tuple di risposta.
- Supporto per il mantenimento e l'interrogazione delle cosiddette *rappresentazioni ibride*. Un 'substrate representation layer' può essere associato ad una ABox per creare una rappresentazione ibrida stratificata. Sono disponibili sia *data substrate* sia *RCC substrate*. Una volta creata la rappresentazione ibrida, questa può essere interrogata con nRQL che un linguaggio di query ibrido.
- *Ragionamento su query*: questa funzionalità è ancora in via sperimentale (quindi è incompleta). nRQL offre il controllo per la soddisfacibilità delle query (incompleta) e per la sussunzione di query. nRQL infatti può essere avvisato per mantenere il cosiddetto *query repository (QBox)*, che è una cache di query DAG-strutturata. nRQL usa il controllo di sussunzione della query per mantenere e computare questo DAG.

Dopo aver introdotto il linguaggio nRQL e descritto il suo motore di processamento, focalizziamo l'attenzione sulla sintassi e la semantica del linguaggio, prendendo come riferimento per eventuali esempi la base di conoscenza `family.racer`, presente nella pagina di download di RacerPro personale. Quindi, una volta connesso RacerPorter al server, bisogna premere sul pulsante **Load** per caricare la TBox da utilizzare nei prossimi esempi. Per ogni query che verrà presa in considerazione, sarà fornita subito nella riga successiva, identificata dal simbolo '>', la risposta che il sistema dà alla query.

3.2 Atomi di query, oggetti, individui e variabili

Le espressioni di base del linguaggio nRQL sono detti *atomi di query*, o anche semplicemente *atomi*. Essi possono essere unari (fanno riferimento ad un unico oggetto) o binari (fanno riferimento a due oggetti).

Un *oggetto* può essere sia un *individuo ABox* (per es. `betty`) oppure una *variabile* (come `?x` oppure `$?x`). Le variabili sono legati a quei individui ABox che soddisfano l'espressione della query.
nRQL offre due diversi tipi di variabile:

- *variabile iniettive* (prefissata con `?x`): può essere legata ad un individuo ABox che non è già legato ad una altra variabile iniettiva (es. se `?x` è legata alla variabile `betty`, allora la variabile `?y` non può essere legata a `betty`).

- *variabile ordinary (non-iniettive)* (prefissate con `$?x`): il mapping non deve essere iniettivo (nell'esempio precedente la variabile `$?y` può essere legata a `betty`).

  Nel caso in cui solo una coppia di variabili deve essere legata a differenti individui si può utilizzare l'espressione `(neg (same-as $?x $?y))` per indicare che le variabili `$?x` e `$?y` dovranno essere associate a individui dell'ABox differenti.

Ci sono anche quattro tipi differenti di atomi:

- Atomi unari
  - Atomi di query di concetto.
- Atomi binari
  - Atomi di query di ruolo;
  - Atomi di query di vincolo;
  - Atomi di query SOME-AS.

C'è inoltre qualche atomo ausiliare, come `bind-individual`, etc.
Analizziamo ora ogni singolo atomo, ricordando che lo si può negare in due modi differenti: la negazione come insuccesso (NAF) e la classica negazione del vero.

3.2.1 Atomi di query di concetto (Concept Query Atoms)

Sono atomi unari e vengono utilizzati per recuperare tutti gli individui (membri) di un concetto o di una classe OWL. Un esempio di questo tipo di atomo è `(?x woman)` che viene utilizzato nella query nRQL necessaria per recuperare tutte le istanze del concetto `woman` della ABox:

```
(retrieve (?x) (?x woman))
 > (((?X EVE)) ((?X DORIS)) ((?X ALICE)) ((?X BETTY))).
```

Come possiamo osservare il sistema restituisce come risultato una *lista di liste di binding* (collegamenti). Ogni lista elenca un numero di coppie variabile-valore. Il formato di queste liste di binding è specificato nella *query head* (in questo caso `?x`), mentre il *query body* (o *query expression*) è costituito dall'atomo di query di concetto `(?x woman)`. L'insieme di oggetti menzionati nella query head deve essere un sottoinsieme dell'insieme di oggetti citati nel query body. E' permessa anche una query head vuota.

Per le variabili è utilizzata la cosiddetta *semantica del dominio attivo*. Infatti esse possono essere associate soltanto a individui dell'ABox modellati esplicitamente nella corrente ABox. Bisogna notare che non ci sono garanzie sull'ordine con cui le possibili associazioni (bindings) sono consegnate. Attualmente infatti, non c'è nessuna funzione di ordinamento disponibile.

Un qualsiasi atomo di query di concetto `(?x C)` per un arbitrario concetto C, ha la stessa semantica di `(concept-instances C)`. Inoltre invece di un semplice concetto come woman, possiamo utilizzare un termine di concetto più complesso come `(and human (some has-gender female))`.

Query con individui. Supponiamo che vogliamo sapere se c'è qualche `woman` nella ABox corrente. Ciò si può fare semplicemente utilizzando la seguente query.

```
(retrieve () (?x woman))
> T
```

Si può notare che la query head è vuota, in modo tale che non venga restituito nessun binding variabile-valore, ma semplicemente T se può essere trovato qualche binding che soddisfa la query, altrimenti NIL.
E' possibile inoltre utilizzare un individuo dell'ABox al posto di una variabile, come nel seguente caso, utilizziamo l'oggetto `betty` per vedere se è una `woman`:

```
(retrieve () (betty woman))
> T .
```

Interrogare KB OWL (RDF Data) con atomi di query di concetto. Considerando il seguente documento OWL, che definisce un'istanza `michael` della classe `person`, e un'istanza `book123` della classe `book`.

```
<?xml version="1.0"?>
     <rdf:RDF
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
          xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
          xmlns:owl="http://www.w3.org/2002/07/owl#"
          xmlns="http://www.owl-ontologies.com/unnamed.owl#"
     xml:base="http://www.owl-ontologies.com/unnamed.owl">
     <owl:Ontology rdf:about=""/>

     <owl:Class rdf:ID="person"/>
     <owl:Class rdf:ID="book"/>

     <person rdf:ID="michael"/>
     <book rdf:ID="book123"/>

</rdf:RDF>
```

Con nRQL possiamo recuperare le istanze della classe person nel seguente modo:

```
(retrieve (?x) (?x |http://www.owl-ontologies.com/unnamed.owl#person|))
 > (((?X |http://www.owl-ontologies.com/unnamed.owl#michael|)))
```

E' importante l'uso delle 'pipes'(|…|), perché altrimenti la query non funziona.

nRQL è più potente di qualsiasi altro linguaggio di interrogazione di documenti RDF perchè permette di utilizzare espressioni di concetti complesse come

```
(retrieve (?x) (?x (or |http://www.owl-ontologies.com/unnamed.owl#person|
                       |http://www.owl-ontologies.com/unnamed.owl#book|)))
```

che non possono essere espresse con un qualsiasi altro linguaggio di query RDF.

3.2.2 Atomi di query di ruolo (Role Query Atoms)

Differentemente da quelli visti in precedenza, questi sono atomi binari e vengono utilizzati per recuperare coppie dei riempitivi di ruolo contenuti in una ABox o coppie di individui OWL (o RDF) che sono in relazione a una certa proprietà di oggetti OWL l'uno con l'altro.

Se per esempio siamo interessati a conoscere le coppie madre-figlio nella nostra ABox `smith-family`, utilizziamo il ruolo `has-child` per formulare la seguente query:

```
(retrieve (?mother ?child) (?mother ?child has-child))
> (((?MOTHER BETTY) (?CHILD DORIS))
   ((?MOTHER BETTY) (?CHILD EVE))
   ((?MOTHER ALICE) (?CHILD BETTY))
   ((?MOTHER ALICE) (?CHILD CHARLES))).
```

In questo caso l'espressione `(?mother ?child has-child)` costituisce l'atomo di query di ruolo.
Se per esempio, però fossimo interessato soltanto ai figli di Betty, la query diventerebbe:

```
(retrieve (?child-of-betty) (betty ?child-of-betty has-child))
> (((?CHILD-OF-BETTY DORIS)) ((?CHILD-OF-BETTY EVE)))
```

Termini di ruolo in atomi di query di ruolo. L'insieme dei termini di ruolo di RacerPro è piuttosto limitato in quanto l'unico costruttore dei termini disponibile è **`inv`**, che permette di costruire ***ruoli invertiti***. La query seguente che utilizza tale costrutto

```
(retrieve (?mother ?child) (inv (?mother ?child has-child)))
```

risulta essere equivalente alla query

```
(retrieve (?mother ?child) (?child ?mother (inv has-child)))
```

che dà la stessa risposta della query

```
(retrieve (?mother ?child) (?mother ?child has-child)),
> (((?MOTHER BETTY) (?CHILD DORIS))
   ((?MOTHER BETTY) (?CHILD EVE))
   ((?MOTHER ALICE) (?CHILD BETTY))
   ((?MOTHER ALICE) (?CHILD CHARLES))).
```

nRQL supporta un ulteriore costruttore dei termini che può essere utilizzato soltanto con atomi di query di ruolo che è il cosiddetto ***ruolo negato classico*** (**`not`**), a differenza di altre logiche descrittive *ALCQHI*$_{R+}$*(D-)* (DL implementate da RacerPro) che non supportano questo costrutto per motivi di decidibilità. Attenzione però a non confondere un atomo di query positivo che usa un ruolo negato (classica negazione del vero) con atomi di query di ruolo negati (la negazione viene interpretata come semantica di fallimento).

Un esempio di query di concetto che utilizza questo costrutto è

```
(retrieve (?x) (?x (not mother)))
> (((?X CHARLES)))
```

perché `CHARLES` è un `man`, e `man` e `woman` sono insiemi disgiunti. Quindi il sistema è in grado di dimostrare che `CHARLES` è un'istanza del concetto `(not mother)`.

Inoltre il costrutto not può essere utilizzato per provare che un certa coppia di individui ABox non può essere coinvolta in una certa relazione di ruolo. Per esempio introducendo un ***feature*** (ruolo funzionale) `has-father`, si potrebbe provare che certi individui non possono essere in relazione di ruolo `has-father` con un altro. Infatti la query

```
(retrieve (?x ?y) (?x ?y (NOT has-father)))
```

restituisce tutte le coppie in cui la variabile `?y` viene associata solo a donne, che non possono essere coinvolte nella relazione `has-father`.

Bisogna tenere in considerazione che all'interno di questo tipo di atomi, possiamo utilizzare soltanto i ruoli (o termini di ruoli) e i features (che non sono altro che ruoli funzionali). Invece non è permesso utilizzare ***attributi del dominio concreto***. Infatti nel caso fosse definito l'attributo `age` come segue

```
(define-concrete-domain-attribute age :type cardinal)
```

Non è possibile eseguire una query per trovare l'età dei membri della famiglia nel seguente modo

```
(retrieve (?person ?age) (?person ?age age)).
```

Infatti il sistema risponde alla query con un messaggio di errore, per il semplice motivo che in nRQL le variabili possono essere associate solo a individui della ABox. Tuttavia l'età degli individui può essere recuperata tramite gli operatori ***head projection*** di cui tratteremo in seguito.

Interrogazione per individui uguali semanticamente (NRQL-EQUAL-ROLE**).** nRQL supporta un ruolo di uguaglianza che può essere usato per trovare coppie di individui che denotano lo stesso individuo di dominio in tutti i modelli della KB (quindi coppie di individui che sono semanticamente uguali). Questo speciale ruolo può essere utilizzato soltanto all'interno di atomi di query di ruolo.
Prendendo in considerazione l'ABox costruita come risultato delle seguenti dichiarazioni:

```
> (full-reset)
:OKAY-FULL-RESET

> (define-primitive-role f :feature t)
F

> (related i j f)
> (related i k f)
```

Possiamo affermare che gli individui `j` e `k` denotano lo stesso individuo nel dominio, e quindi sono semanticamente uguali, anche se hanno nomi differenti. Infatti utilizzando il ruolo speciale `NRQL-EQUAL-ROLE` si ottiene:

```
(retrieve (?x ?y) (?x ?y nrql-equal-role))
> (((?X K) (?Y J)) ((?X J) (?Y K)))

(retrieve ($?x $?y) ($?x $?y nrql-equal-role))
>((($?X K) ($?Y J)) (($?X J) ($?Y K))
  (($?X I) ($?Y I)) (($?X J) ($?Y J)) (($?X K) ($?Y K)))
```

Interrogare KB OWL (RDF Data) con atomi di query di ruolo. Nel caso di documenti OWL, le ***object properties*** sono equivalenti ai ruoli. Considerando il seguente documento OWl:

```
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns="http://www.owl-ontologies.com/unnamed.owl#"
   xml:base="http://www.owl-ontologies.com/unnamed.owl">
   <owl:Ontology rdf:about=""/>
   <owl:Class rdf:ID="person"/>
   <owl:ObjectProperty rdf:ID="hasChild">
     <rdfs:domain rdf:resource="#person"/>
     <rdfs:range rdf:resource="#person"/>
   </owl:ObjectProperty>
   <person rdf:ID="margrit">
     <hasChild>
       <person rdf:ID="michael"/>
     </hasChild>
   </person>
</rdf:RDF>
```

In questa KB, l'individuo `http://www.owl-ontologies.com/unnamed.com#michael` è il riempitivo della *object property* `http://www.owl-ontologies.com/unnamed.com#hasChild` dell'individuo `http://www.owl-ontologies.com/unnamed.com#margrit`. Siccome le object properties sono equivalenti ai ruoli, allora possiamo costruire semplici atomi di query di ruolo come il seguente

```
(retrieve (?x ?y)
     (?x ?y |http://www.owl-ontologies.com/unnamed.owl#haschild|))
> (((?X |http://www.owl-ontologies.com/unnamed.owl#margrit|)
    (?Y |http://www.owl-ontologies.com/unnamed.owl#michael|))).
```

3.2.3 Atomi di query di vincolo (Constraint Query Atoms)

Sono atomi binari e vengono utilizzati per rivolgersi ad una parte del dominio concreto di una KB. Un atomo del genere può essere utilizzato per recuperare quelle coppie di individui di un ABox (o OWL) il cui valore di certi attributi di dominio concreto soddisfa un specificato predicato di dominio concreto, anche detto ***vincolo***.

Riprendendo l'esempio, aggiungiamo con un qualsiasi text editor le seguenti righe alla fine del file:

```
    (instance alice (= age 80))
    (instance betty (= age 50))
    (instance charles (= age 55))
```

```
(instance eve (= age 18))
(instance doris (= age 24))
```

che associano un età ad ogni individuo della ABox. A questo punto possiamo effettuare una query che restituisce tutte le persone che hanno un età maggiore di 75 anni:

```
(retrieve (?x) (?x (>= age 75)))
> (((?X ALICE))).
```

Possiamo costruire anche query più complesse. Per esempio potremmo essere interessati a trovare chi è più grande di chi (oppure chi è più grande di un determinato individuo). La query che ci permette di farlo è

```
(retrieve (?x ?y) (?x ?y (constraint age age >)))
> (((?X CHARLES) (?Y EVE))
   ((?X CHARLES) (?Y DORIS))
   ((?X CHARLES) (?Y BETTY))
   ((?X ALICE) (?Y CHARLES))
   ((?X ALICE) (?Y EVE))
   ((?X ALICE) (?Y DORIS))
   ((?X ALICE) (?Y BETTY))
   ((?X DORIS) (?Y EVE))
   ((?X BETTY) (?Y EVE))
   ((?X BETTY) (?Y DORIS)))
```

Da ciò deduciamo che `CHARLES` è più grande di `EVE`, di `DORIS`, di `BETTY`, e così via.

Invece di utilizzare due singoli nomi di attributi (come `age age` nel caso precedente), potrebbe essere utile costruire ***catene di ruoli di lunghezza arbitraria*** tale che l'ultimo argomento in ogni catena è un attributo di dominio concreto. Riprendendo il solito esempio, dopo aver modificato sia la TBox che la ABox, introducendo le features has-mother e has-father potrebbe essere interessante vedere chi ha un padre più grande della madre, come segue:

```
(retrieve (?x) (?x ?x (constraint (has-father age)
                                   (has-mother age) >))).
```

E' da notare che le catene di ruoli, `(has-father age)` e `(has-mother age)`, terminano tutte con attributi di domini concreto e possono avere lunghezza arbitraria, come per esempio (has-father has-mother age).

Espressioni di predicato complesse. nRQL permette di utilizzare anche espressioni di predicato più complesse. Supponendo di essere interessati a chi è almeno 40 anni più grande di qualcun' altro, la query che permette di farlo avrà la seguente struttura:

```
(retrieve (?x ?y) (?x ?y (constraint (age) (age)
                         (> age-1 (+ age-2 40))))).
```

All'interno dell'espressione vengono utilizzati i termini age-1 e age-2, che permettono di differenziare l'attributo age dell'individuo associato alla variabile ?x da quello associato alla variabile ?y. Nel caso in cui i nomi degli attributi che compaiono nella query sono differenti, allora il suffisso non dovrà essere aggiunto, come nel caso che segue

```
(retrieve (?x ?y) (?x ?y (constraint (age) (works-for-years-in-company)
                          (< age (+ works-for-years-in-company 10))))).
```

Interrogare KB OWL (RDF Data) con atomi di query di vincolo. Nel caso di OWL, il concetto equivalente di attributi del dominio concreto è **OWL *datatype properties***. In questo modo, gli atomi di query di vincolo possono essere usati per interrogare documenti OWL con atomi di query di vincolo relazionati a datatype properties come se fossero attributi di dominio concreto.

Prendiamo in considerazione la seguente KB OWL, nella quale sono state definite 2 datatype properties age e name, e tre istanze del concetto person (a,b,c).

```
<?xml version="1.0"?>
<rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns="http://www.owl-ontologies.com/unnamed.owl#"
   xml:base="http://www.owl-ontologies.com/unnamed.owl">

   <owl:Ontology rdf:about=""/>

   <owl:Class rdf:ID="person"/>

   <owl:DatatypeProperty rdf:ID="age">
     <rdfs:domain rdf:resource="#person"/>
     <rdf:type
     rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
     <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
   </owl:DatatypeProperty>

   <owl:FunctionalProperty rdf:ID="name">
     <rdfs:range
     rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
     <rdfs:domain rdf:resource="#person"/>
     <rdf:type
     rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
   </owl:FunctionalProperty>

   <person rdf:ID="b">
     <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">45</age>
     <name
     rdf:datatype="http://www.w3.org/2001/XMLSchema#string">betty</name>
   </person>

   <person rdf:ID="a">
     <name
     rdf:datatype="http://www.w3.org/2001/XMLSchema#string">betty</name>
     <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">35</age>
   </person>

   <person rdf:ID="c">
     <name
          rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Michael
     </name>
     <age rdf:datatype="http://www.w3.org/2001/XMLSchema#int">35</age>
   </person>
```

```
</rdf:RDF>
```

Per recuperare gli individui che hanno la stessa età, possiamo utilizzare atomi di query di vincolo come segue

```
(retrieve (?x ?y)(?x ?y
 (constraint |http://www.owl-ontologies.com/unnamed.owl#age|
          |http://www.owl-ontologies.com/unnamed.owl#age| =)))
> (((?X |http://www.owl-ontologies.com/unnamed.owl#c|)
    (?Y |http://www.owl-ontologies.com/unnamed.owl#a|))
   ((?X |http://www.owl-ontologies.com/unnamed.owl#a|)
    (?Y |http://www.owl-ontologies.com/unnamed.owl#c|)))
```

3.2.4 Atomi di query SOME-AS (SOME-AS Query Atoms)

Sono atomi binary e vengono utilizzati per obbligare un collegamento a un individuo dell'ABox o per obbligare che due variabili non-iniettive siano limitate allo stesso (o a un differente) individuo della ABox. Il seguente esempio non ha bisogno di spiegazioni:

```
(full-reset)
> :OKAY-FULL-RESET

(instance i top)
(instance j top)

(retrieve (?x) (same-as ?x i))
> (((?X I)))

(retrieve (?x ?y) (same-as ?x ?y))
> NIL

(retrieve ($?x $?y) (same-as $?x $?y))
> ((($?X J) ($?Y J)) (($?X I) ($?Y I)))

(retrieve ($?x $?y) (and (same-as $?x $?y) (same-as j $?y)))
>((($?X J) ($?Y J)))

(retrieve () (same-as i j))
>NIL

(retrieve () (same-as i i))
>T
```

3.2.5 Atomi di query ausiliare (Auxiliary Query Atoms)

nRQL offre anche atomi di query ausiliari che sono meno importanti dei precedenti, ma che comunque vale la pena analizzare. Questi sono gli atomi di query **HAS-KNOWN-SUCCESSOR** che permettono di recuperare gli individui che hanno un certo successore di ruolo modellato esplicitamente nell'ABox, ma non siamo interessati attualmente al recupero di questo successore. Per esempio supponiamo di voler conoscere gli individui che hanno un figlio modellato esplicitamente nell'ABox. Possiamo ottenere questi individui con la seguente query:

```
(retrieve (?x) (?x (has-known-successor has-child)))
> (((?X CHARLES)) ((?X BETTY)) ((?X ALICE))).
```

Alternativamente potevamo usare

```
(retrieve (?x) (?x ?y has-child)).
```

Comunque c'è una sottile differenza tra le due query si evidenzierà quando considereremo in seguito *atomi di query negati*.
Notiamo anche che la query

```
(retrieve (?x) (?x (has-known-successor has-child)))
```

non è equivalente a

```
(retrieve (?x) (?x (some has-child top))).
```

Infatti supponendo di inserire l'assioma

```
(individual-instance doris mother)
```

nell'ABox, il sistema risponde a quest'ultima query nel seguente modo:

```
(((?X DORIS)) ((?X CHARLES)) ((?X BETTY)) ((?X ALICE))),
```

Mentre per la prima query non viene aggiunta `DORIS` perché non c'è nella ABox nessun suo figlio modellato esplicitamente.

3.3 Operatori Query Head Projection per recuperare valori noti

Gli operatori di proiezione nella testa della query (query head) sono necessari per recuperare:

- i riempitivi degli attributi del dominio concreto di individui dell'ABox (anche detti oggetti del dominio concreto),
- i valori noti di tali attributi del dominio concreto (anche detti valori del dominio concreto),
- i valori datatype dello schema XML che sono valori noti del datatype OWL e proprietà di annotazione OWL dell'individuo OWL.

Con questi operatori non solo è possibile recuperare questi valori noti, ma anche specificare condizioni di recupero complesse (usando predicati di dominio concreto) sui valori noti che devono essere recuperati.

Tali operatori denominati *head projection operators* vengono definiti nella testa della query, che non è altro che una lista di oggetti. Sono denotati in uno stile funzionale, come ad esempio `(<op1> ?x)` che sta a significare che l'operatore di proiezione `<op1>` è applicato al collegamento corrente di `?x`.

Ma quale è il vantaggio di utilizzare questi operatori? Le variabili possono essere associate soltanto a individui della ABox, e mai a oggetti e a valori del dominio concreto. Ed è proprio in questo caso che sfruttiamo il potere computazionale degli operatori head projection, che permettono di recuperare oggetti del dominio concreto e quei valori noti (per esempio potremmo essere interessati al valore noto dell'attributo di dominio concreto `age` dell'individuo `alice`).

Bisogna fare attenzione a come definire un attributo del dominio concreto all'interno dell'ABox. Infatti, possiamo definire l'età di alice nel nostro esempio in tre modi differenti:

1. Tramite un assioma di appartenenza al concetto come

   ```
   (instance alice (= age 80))
   ```

   dove `(= age 80)` è una semplice espressione di concetto.

2. Tramite il comando `(constrained alice alice-age age)`, nel quale l'oggetto `alice-age` è anche detto oggetto di dominio concreto, che è il riempitivo dell'attributo di dominio concreto `age` di `alice`.

3. Tramite il comando `(attribute-filler alice 80 age)`, che crea un oggetto di dominio concreto anonimo, che svolge lo stesso ruolo di `alice-age`.

Se però viene definito seguendo il caso 1, il valore non può essere recuperato dagli operatori *head projection*; cosa che invece è possibile dichiarando il valore dell'attributo seguendo i casi 2 e 3.

Prima di esaminare i due operatori di proiezione si deve inserire all'interno del file `family.racer` le seguenti righe:

```
(constrained alice alice-age age)
(constraints (= alice-age 80))
```

3.3.1 Operatore di proiezione dell'attributo

Un operatore di questo tipo (es `(age alice)`) recupera oggetti del domino concreto della ABox che sono noti essere un riempitivo del dominio concreto (`age`) di un individuo (`alice`). Questo operatore è disponibile per ogni attributo del dominio concreto definito nella TBox di riferimento.
Per esempio la query

```
(retrieve (alice (age alice))(bind-individual alice))
> ((($?ALICE ALICE) ((AGE $?ALICE) (ALICE-AGE)))).
```

Nel caso ci fosse più di un riempitivo dell'oggetto di dominio concreto per l'attributo `age` di `alice`, come nel caso in cui venisse aggiunta l'asserzione `(constrained alice huhu age)` nella KB, otterremo la risposta `((($?ALICE ALICE) ((AGE $?ALICE) (HUHU ALICE-AGE))))` alla domanda precedente. In questo caso `HUHU e ALICE-AGE` rappresentano lo stesso oggetto di dominio concreto rispetto al valore.

Se non ci fosse alcun oggetto di dominio concreto che è noto essere un riempitivo di un attributo, viene introdotto il termine `NO-KNOWN-CD-OBJECT` che indica il fatto che non c'è alcun oggetto di dominio concreto che è un valore di questo attributo.

```
(retrieve (betty (age betty))(bind-individual betty))
> ((($?BETTY BETTY) ((AGE $?BETTY) :NO-KNOWN-CD-OBJECTS))).
```

3.3.2 Operatore di proiezione dei valori noti

L'operatore `(told-value (age alice))` può essere utilizzato su oggetti di dominio concreto per recuperare il loro valore attuale. Nel nostro esempio per recuperare l'età di Alice utilizziamo la query:

```
    (retrieve (alice (age alice) (told-value (age alice)))
                                  (bind-individual alice))
     > ((($?ALICE ALICE)
          ((AGE $?ALICE) (HUHU ALICE-AGE))
          ((:TOLD-VALUE (AGE $?ALICE)) (:NO-TOLD-VALUE 80))))
```

E' da notare che i termini `huhu` e `alice-age` denotano semanticamente lo stesso oggetto di dominio concreto, con la differenza che il primo non ha alcun valor noto `(NO-TOLD-VALUE)`, mentre il corretto valor noto del secondo è 80.

3.3.3 Osservazioni sulla completezza

L'operatore `told-value` è a volte incompleto. Considerando per esempio il concetto `(and (> age 18) (< age 20))`. Se `age` è un attributo di tipo cardinale o intero, allora `age` potrebbe essere anche 19. Quindi, 19 potrebbe a volte essere considerato anche come valore noto. Siccome però non c'è modo per RacerPro o nRQL di ritornare questo valore come un `told-value`, allora RacerPro può solo controllare la soddisfacibilità di un sistema di vincoli di dominio concreto, ma non calcola le sue soluzioni (anche se a volte in molti casi potrebbe essere possibile ritornare queste soluzioni come valori singoli).

3.3.4 Recuperare i riempitivi di valore noto delle OWL datatype properties

All'interno della KB riportata di seguito, è definita la datatype properties http://a.com/ontology#p1 di tipo int e per l'individuo *i* sono definiti i valori interi 1, 2 e 3 come riempitivi della datatype property http://a.com/ontology#p1.

```
<?xml version="1.0"?>
  <rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">

  <owl:Ontology rdf:about=""/>

  <owl:Class rdf:ID="test"/>

  <owl:DatatypeProperty rdf:ID="p1">
     <rdfs:domain rdf:resource="#test"/>
     <rdfs:range
     rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
  </owl:DatatypeProperty>

  <test rdf:ID="i">
```

```
      <p1 rdf:datatype="http://www.w3.org/2001/XMLSchema#int">3</p1>
      <p1 rdf:datatype="http://www.w3.org/2001/XMLSchema#int">2</p1>
      <p1 rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</p1>
  </test>
</rdf:RDF>
```

Per specificare le condizioni di recupero su valori datatype, utilizziamo l'operatore di proiezione dell'attributo :

```
(retrieve
      (?x
       (told-value (|http://www.owl-ontologies.com/unnamed.owl#p1| ?x)))
       (?x |http://www.owl-ontologies.com/unnamed.owl#test|))
> (((?X |http://www.owl-ontologies.com/unnamed.owl#i|)
      ((:TOLD-VALUE
       (|http://www.owl-ontologies.com/unnamed.owl#p1| ?X))
       (1 2 3))))
```

come se `http://www.owl-ontologies.com/unnamed.owl#p1` fosse un attributo di dominio concreto. Al posto di `told-value`, può essere utilizzato anche `fillers`, `datatype fillers`, `told-values` o `told-fillers`.
Si estende la sintassi di concetto di RacerPro per esprime condizioni di recupero complesse sui fillers di una certa datatype properties, come per esempio:

```
   (retrieve (?x (datatype-fillers
               (|http://www.owl-ontologies.com/unnamed.owl#p1| ?x)))
      (?x (at-least 3 |http://www.owl-ontologies.com/unnamed.owl#p1|
                    (and (min 0) (max 5) (not (equal 4))))))
```

a cui il sistema risponde come nel caso precedente.

3.3.4 Recuperare valori noti di OWL Annotation Properties

Le *annotation properties* sono utilizzate per annotare risorse con meta informazione (come per esempio un commento sull'autore dell'ontologia). Tale annotazioni non vengono utilizzate per il ragionamento. Di solito queste annotazioni sono semplici stringhe, valori noti che possono essere recuperati con nRQL.

nRQL può essere usato per recuperare i riempitivi di queste proprietà. Infatti le annotation properties OWL sono gestiti in maniera simile ai valori noti degli oggetti di dominio concreto.

Nella seguente KB sono dichiarate due annotation properties `annot1` e `annot2` così come qualche istanza:

```
<owl:DatatypeProperty rdf:ID="annot1">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdf:type
       rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="annot2">
  <rdf:type
```

```
    rdf:resource="http://www.w3.org/2002/07/owl#AnnotationProperty"/>
</owl:ObjectProperty>
<c rdf:ID="i">
  <annot2 rdf:resource="#j"/>
  <r rdf:resource="#j"/>
  <r rdf:resource="#k"/>
  <annot1 rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
    Annotation
  </annot1>
</c>
```

Possiamo trovare i valori noti di queste proprietà d'annotazione come segue:

```
(retrieve ((annotations
     (|http://www.owl-ontologies.com/unnamed.owl#annot1| ?x)))
     (?x (an |http://www.owl-ontologies.com/unnamed.owl#annot1|)))
>((((:TOLD-VALUE
         (|http://www.owlontologies.com/unnamed.owl#annot1| ?X))
         ("Annotation"))))
```

e

```
(retrieve (?x ?y)
     (?x ?y |http://www.owl-ontologies.com/unnamed.owl#annot2|))
>(((?X |http://www.owl-ontologies.com/unnamed.owl#i|)
   (?Y |http://www.owl-ontologies.com/unnamed.owl#j|))).
```

3.4 Query Complesse

Dopo aver discusso degli atomi di query disponibili e della struttura della testa (head) di una query nRQL, posiamo l'attenzione sulla struttura del corpo (body) della query. Infatti un corpo nRQL è definito sia come un singolo atomo di query, sia come un body più complesso costruito fornendo i diversi body di query che lo costituiscono come argomenti ad uno dei seguenti costruttori di query:

AND: è un costruttore n-ario che viene usato per la formulazione di query congiuntive. Gli argomenti dell'AND sono detti congiunti.

UNION: può essere utilizzato per calcolare l'unione dei singoli insiemi di risposta dei corpi di query argomenti. Gli argomenti dell'UNION sono detti disgiunti.

NEG: implementa una negazione.

INV: può essere usato per invertire tutti gli atomi di query di ruolo eventualmente presenti nel corpo di query argomento.

PROJECT-TO: è l'operatore di proiezione per body di query. Il primo argomento di questo costrutto è una *projection list*, che dovrebbe essere una lista di oggetti. Qui non sono permessi operatori head projection. Il secondo argomento è un corpo di query ordinario. Comunque, come per una query nRQL, gli oggetti menzionati nella lista di proiezione dovrebbero formare un sottoinsieme degli oggetti menzionati nel corpo della query. Questo costrutto non deve essere confuso con operatori head projection che abbiamo discusso prima.

3.4.1 Costruttore AND – Query Congiuntive

Un esempio di query che utilizza questo costrutto è quella che restituisce tutte le mamme di persone maschili nella KB family.racer:

```
(retrieve (?x ?y) (and (?x mother) (?y man) (?x ?y has-child)))
> (((?X ALICE) (?Y CHARLES))).
```

Nelle query congiuntive bisogna fare attenzione al tipo di variabile che si sta utilizzando. Infatti risposte a query che utilizzano variabili iniettive come

```
(retrieve (?x ?y) (and (?x man) (?y man)))
> NIL
```

sarà sicuramente differente da quella che risponde a query in cui sono presenti variabile non-iniettive come

```
(retrieve ($?x $?y) (and ($?x man) ($?y man)))
> ((($?X CHARLES) ($?Y CHARLES))).
```

Di seguito è riportato un esempio di query più complessa, nella quale vengono menzionati anche i ruoli, ed effettua la ricerca di tutte le persone che hanno una mamma in comune.

```
(retrieve (?mother ?child1 ?child2)
       (and (?child1 human)
            (?child2 human)
            (?mother ?child1 has-child)
            (?mother ?child2 has-child))).
>(((?MOTHER BETTY) (?CHILD1 DORIS) (?CHILD2 EVE))
 ((?MOTHER BETTY) (?CHILD1 EVE) (?CHILD2 DORIS))
 ((?MOTHER ALICE) (?CHILD1 BETTY) (?CHILD2 CHARLES))
 ((?MOTHER ALICE) (?CHILD1 CHARLES) (?CHILD2 BETTY)))
```

3.4.2 Costruttore UNION

Un esempio di query che utilizza questo costrutto è

```
(retrieve (?x) (union (?x woman) (?x man)))
>(((?X CHARLES))((?X EVE))((?X DORIS))((?X BETTY))((?X ALICE))).
```

Questo costrutto permette di unire gli insiemi di risposta delle query argomento (disgiunti). Comunque l'unione è ben-definita e significativa solo se gli insiemi argomento hanno la stessa arità. Quindi i corpi delle query argomento dovranno essere trasformati in modo tale che gli insiemi di risposta che producono abbiano tutti la stessa arità. Per esempio la query

```
(retrieve (?x ?y) (union (?x ?y has-child) (?x man)))
```

può essere trasformata in

```
(retrieve (?x ?y) (union (?x ?y has-child) (and (?x man) (?y top)))).
```

In questo modo le query componenti restituiscono entrambe una lista di coppie `(?x ?y)`, e quindi l'unione è significativa.

Inoltre se i corpi delle query argomento di una UNION fanno riferimento a differenti variabili, allora nRQL assicurerà che ogni argomento si riferisca alle stesse variabili, anche se hanno la stessa arità. Per esempio, la query

```
(retrieve (?x ?y) (union (?x woman) (?y man)))
```

sarà riscritta in

```
(retrieve (?x ?y) (union (and (?x woman) (?y top))
                          (and (?x top) (?y man))))
```

Ogni variabile nominate differentemente crea un nuovo asse in uno spazio della tupla n-dimensionale. Il risultato della precedente query è

```
(((?X EVE) (?Y DORIS))
((?X EVE) (?Y CHARLES))
((?X EVE) (?Y BETTY))
((?X EVE) (?Y ALICE))
((?X DORIS) (?Y EVE))
((?X DORIS) (?Y CHARLES))
((?X DORIS) (?Y BETTY))
((?X DORIS) (?Y ALICE))
((?X BETTY) (?Y DORIS))
((?X BETTY) (?Y EVE))
((?X BETTY) (?Y CHARLES))
((?X BETTY) (?Y ALICE))
((?X ALICE) (?Y DORIS))
((?X ALICE) (?Y EVE))
((?X ALICE) (?Y CHARLES))
((?X ALICE) (?Y BETTY))).
```

Notare che il secondo disgiunto non produce tuple aggiuntive.

Prendendo in considerazione le seguente query

```
(retrieve (?y) (union (?x woman) (?y man)))
```

sarebbe un errore pensare che questa è equivalente alla seguente query:

```
(retrieve (?y) (?y man))
> (((?Y CHARLES))).
```

Infatti la prima query può essere riscritta come

```
(retrieve (?y) (union (and (?x woman) (?y top))
                      (and (?x top) (?y man))))
> (((?Y ALICE)) ((?Y DORIS)) ((?Y EVE)) ((?Y CHARLES))
   ((?Y BETTY)))
```

Bisogna tenere in considerazione che il costrutto UNION è diverso da quello OR, perché a causa dell'*Open World Semantics*, ci possono essere individui che non possono né essere provati essere istanze di un concetto `C`, né essere provati essere istanze del concetto `(not C)`. Per esempio considerando l'ABox `(instance i top)`, la query `(retrieve (?x) (union (?x c) (?x (not c))))` ritorna `NIL`, ma `(retrieve (?x) (?x (or c (not c))))`, che è equivalente a `(retrieve (?x) (?x top))`, ritorna `(((?x i)))`.

3.4.3 Costruttore NEG – La negazione come costruttore di fallimento

Questo costruttore è previsto per implementare la *negazione come Semantica di Fallimento (NAF)*, che viene utilizzata specialmente per misurare la completezza della modellazione in una ABox, che è importante in molte applicazioni. Anche il linguaggio di programmazione logica PROLOG supporta *NAF*.

NAF è completamente differente dalla classica negazione del vero.

Negazione come fallimento per Concept Query Atoms. Considerando la query

```
(retrieve (?x) (?x grandmother))
> ((?X ALICE)).
```

Quindi RacerPro prova che `alice` è una `grandmother`. Se utilizziamo un concetto negato nel concept query atom che segue

```
(retrieve (?x) (?x (NOT grandmother)))
> ((?X CHARLES)),
```

visto che `charles` è un `man`, e quindi per definizione non può essere un `grandmother`. Grazie alla open world semantics, `charles` è l'unico individuo che può essere provato non essere un `grandmother`, in quanto tutti gli altri, essendo `woman`, potrebbero diventare o potrebbero già esserlo ma non abbiamo una conoscenza completa riguardo a loro. E quindi, correntemente, non è noto se qualche altro individuo è un `grandmother`.

Per conoscere gli individui che correntemente non sono noti essere istanze di grandmother, possiamo utilizzare un atomo di query di concetto negato come segue:

```
(retrieve (?x) (NEG (?x grandmother))).
> (((?X DORIS)) ((?X EVE)) ((?X CHARLES)) ((?X BETTY))).
```

E' da notare che l'operatore **NEG** è posizionato intorno all'intero atomo, e la risposta che si ottiene è complementare all'insieme di risposta ritornato dalla query `(retrieve (?x) (?x grandmother))`. In generale, per ogni concetto C, la query `(retrieve (?x) (union (?x C) (neg (?x C))))`, ritornerà sempre l'insieme di tutti gli individui della ABox. Inoltre la query `(retrieve (?x) (?x (not C)))` ritornerà sempre un sottoinsieme di `(retrieve (?x) (union (?x C) (neg (?x C))))`.

I due operatori NEG e NOT possono essere anche utilizzati in combinazione per costruire query più complesse, come per esempio la query che permette di recuperare tutti gli individui per cui RacerPro non può provare che siano istanze di `(not grandmother)`:

```
(retrieve (?x) (neg (?x (not grandmother)))),
 > (((?X DORIS)) ((?X EVE)) ((?X BETTY)) ((?X ALICE))).
```

Negazione come fallimento per Role Query Atoms. L'operatore **NEG** può essere utilizzato anche per atomi di query di ruolo. Un role query atom con la sua variante NAF negata sono complementari l'uno con l'altro, e quindi la loro unione restituisce sempre l'insieme di tutte le coppie degli individui dell'ABox, per tutti i termini di ruolo R.
Ora supponiamo di voler conoscere tutte le persone senza figli noti. La query

```
(retrieve (?x) (neg (?x ?y has-child)))
> (((?X EVE)) ((?X BETTY)) ((?X DORIS)) ((?X CHARLES)) ((?X ALICE))).
```

Ma come mai è coinvolta anche `BETTY`, che già sappiamo avere due figli (`doris` e `eve`). Questo è dovuto al fatto che il sistema prepara, per prima cosa, la risposta a `(?x ?y has-child));` poi ottiene l'insieme delle coppie corrette (che soddisfano la negazione) facendo la differenza tra l'insieme ottenuto in precedenza e l'insieme di tutte le coppie di individui; e alla fine effettua la proiezione al primo componente della coppia. Siccome la risposta a `(neg (?x ?y has-child))` contiene anche la coppia `((?X BETTY) (?Y ALICE))`, la proiezione su `?x` contiene anche `betty`.
Neanche la query

```
(retrieve (?x) (?x (not (some has-child top))))
> NIL
```

va bene poiché con l'operatore **`NOT`**, a causa dell'open world semantics, se un individuo nella ABox non ha correntemente una relazione `has-child` con altri individui, non significa che non possa averla. Provando allora ad utilizzare l'operatore **`NEG`**

```
(retrieve (?x) (?x (neg (some has-child top))))
> (((?X DORIS)) ((?X EVE))).
```

In questo caso la risposta è corretta, ma purtroppo non è sempre così. Infatti se si aggiungesse all'interno dell'ABox `(instance doris mother)`, la risposta alla query precedente diventerebbe

```
> (((?X EVE)))
```

dalla quale viene escluso `doris`, anche se non ha un figlio noto, perchè RacerPro può provare che `doris` ha un figlio, e quindi è un istanza del concetto `mother`.

Se vogliamo ottenere una risposta positiva al fatto che doris non ha esplicitamente un figlio, dobbiamo utilizzare la query

```
(retrieve (?x) (neg (?x (has-known-successor has-child)))).
> ((?X DORIS) (?X EVE))
```

anche se è stato aggiunto `(instance doris mother).`

Come per concept query atoms, anche nel caso di `(?x ?y (not R))` otteniamo una risposta che è un sotto-insieme di `(neg (?x ?y R))`, per un qualsiasi ruolo R.

Negazione come fallimento per Costraint Query Atoms. Un costraint query atom negato ritorna il complemento della sua variante non *NAF* negata. Come per esempio

```
(retrieve (?x ?y) (?x ?y (constraint (has-father age) age =)))
 > (((?X EVE) (?Y CHARLES)))
```

(poiche `charles` è il padre di `eve`), e otteniamo le 19 tuple rimanenti dalla sua variante negata

```
(retrieve (?x ?y) (neg (?x ?y (constraint (has-father age) age =)))).
```

SOME-AS Query Atoms negati. Un atomo di questo tipo enumera semplicemente il complemento della sua variante positiva. Quindi mentre

```
(retrieve (?x) (same-as ?x eve))
> (((?X EVE))),
```

possiamo ottenere tutti gli altri individui tranne `eve` con

```
(retrieve (?x) (neg (same-as ?x eve)))
> (((?X CHARLES)) ((?X BETTY)) ((?X ALICE)) ((?X DORIS))).
```

3.4.4 Costruttore PROJECT-TO: operatore di proiezione per il corpo della query

Prendendo come riferimento la seguente ABox :

```
(instance a c)
(instance b d)
(instance c top)
(related a b r)
```

Consideriamo la seguente query:

```
(retrieve (?x) (and (?x c) (?x ?y r) (?y d)))
>(((?X A))).
```

In questa query, come in tutti i casi visti fino ad ora, la proiezione delle variabili è menzionata nella testa della query, e quindi viene eseguita dal sistema come ultima operazione nella catena di processamento della query. Comunque è possibile effettuare un'operazione di proiezione all'interno della catena.

Supponendo di voler recuperare quelle istanze C che non hanno successori noti R che sono istanze di D. In questo caso la risposta dovrebbe essere `(((?X B)) ((?X C)))`.
Ma la query

```
(retrieve (?x) (neg (and (?x c) (?x ?y r) (?y d)))),
```

che è equivalente a

```
(retrieve (?x) (union (and (neg (?x c)) (top ?y))
                      (neg (?x ?y r))
                      (and (neg (?y d)) (top ?x))))
```

effettua una proiezione su `?x` al complemento dell'insieme `((A B))`, che corrisponde all'insieme `((B A) (B C) (C A) (C B) (A C))` ottenendo così l'insieme `(((?X B)) ((?X C)) ((?X A)))`, che non è quello che vogliamo. Si ha questo problema perché l'operatore `neg` viene applicato ad un insieme bidimensionale e restituisce un altro insieme bidimensionale, al quale viene applicata la proiezione.

Per risolvere questo problema, bisogna fare in modo di applicare l'operatore di complemento `neg` dopo la proiezione su `?x`. Quindi utilizziamo l'operatore **`project-to`** all'interno del corpo della query:

```
(retrieve (?x) (neg (project-to (?x) (and (?x c) (?x ?y r) (?y d)))))
```

che ritorna il risultato desiderato.

Al posto di `project-to`, si può utilizzare anche il termine **`project`** e **`pi`**.

Ricordiamo inoltre che l'atomo `(has-known-successor R)` è sintatticamente equivalente a `(project-to (?x) (?x ?y R))`, mentre `(neg (has-known-successor R))` è equivalente a `(neg (project-to (?x) (?x ?y R)))`.

3.5 Query definite

nRQL offre anche un meccanismo per specificare *query definite*. Per esempio si può associare il simbolo `mother-of` con la testa `(?x ?y)` di arità 2 e il corpo `(and (?x woman) (?x ?y has-child) usando` l'agevolazione **`defquery`**:

```
(defquery mother-of (?x ?y) (and (?x woman) (?x ?y has-child))).
```

Il sistema risponde

```
MOTHER-OF
```

per indicare che la definizione è stata memorizzata.
La lista `(?x ?y)` nella definizione di `mother-of` è detta lista dei *parametri formali* della definizione. Tale lista di liste di parametri formali differisce da una query head, poiché sono permessi solo oggetti (variabili e individui), e non sono permessi operatori head projection.
Una query definita può essere riutilizzata all'interno di una query

```
(retrieve (?a ?b) (substitute (mother-of ?a ?b))),
```

oppure utilizzando una sintassi alternativa

```
(retrieve (?a ?b) (?a ?b mother-of)).
```

Data la definizione della query mother-of , entrambe le query sono equivalenti a

```
(retrieve (?a ?b) (and (?a woman) (?a ?b has-child))).
```

Nell'espressione `(substitute (mother-of ?a ?b))`, i parametri `?a ?b` sono detti *parametri attuali*. Il numero di parametri attuali dovrà sempre corrispondere con il numero di parametri formali utilizzati nella definizione della query. Notare infatti che i parametri `?x ?y` utilizzanti nella definizione sono stati rimpiazzati dai parametri attuali `?a ?b`.

Quindi il risultato alle query precedenti è:

```
>(((?A ALICE) (?B CHARLES)) ((?A ALICE) (?B BETTY))
  ((?A BETTY) (?B EVE)) ((?A BETTY) (?B DORIS))).
```

Nel caso volessimo ignorare la corrispondenza con un certo parametro formale, allora possiamo semplicemente utilizzare NIL come parametro attuale, come nel seguente caso:

```
(retrieve (?mother) (substitute (mother-of ?mother NIL))),
```

che è equivalente a

```
(retrieve (?mother) (?mother NIL mother-of)).
```

Ad entrambe le query il sistema risponde:

```
> (((?MOTHER ALICE)) ((?MOTHER BETTY))).
```

A volte potrebbe essere necessario inserire un'operatore project-to intorno alla query definita di riferimento come

```
(retrieve (?a) (neg (project-to (?a) (?a ?b mother-of))))
```

Notare che la query

```
(retrieve (?a ?b) (?a ?b mother-of))
```

è *sintatticamente ambigua*, data la definizione di `mother-of`, visto che l'espressione è sintatticamente indistinguibile da una query di atomo di ruolo, poichè `mother-of` potrebbe essere anche un nome di ruolo. Lo stesso problema può verificarsi per atomi di query di concetto, dove il secondo argomento dell'atomo può riferirsi sia a un nome di concetto sia ad una query definita unaria. In questo caso il sistema risponde con un warning come

```
*** NRQL WARNING: MOTHER-OF EXISTS IN TBOX DEFAULT.
ASSUMING YOU ARE REFERRING TO THE ROLE MOTHER-OF!
```

che ci informa che la query è ambigua, ma poi assume che ci stiamo riferendo al ruolo (o al concetto) con quel nome, e non alla query definita. Altrimenti potremmo usare l'operatore `substitute` per distinguere i due casi.

Uso di query definite nella definizione di query. E' possibile utilizzare delle query definite nella definizione della query. Tuttavia sono proibite definizioni cicliche.
Un esempio è il seguente

```
(defquery mother-of-male-child (?m)
               (and (substitute (mother-of ?m ?c)) (?c man)))

(retrieve (?x) (substitute (mother-of-male-child ?x)))
> (((?X ALICE)))
```

che può essere espresso anche con la sintassi alternativa:

```
(defquery mother-of-male-child (?m)
               (and (?m ?c mother-of) (?c man)))

(retrieve (?x) (?x mother-of-male-child)).
```

Problemi con query definite NAF-negate. Supponiamo di voler conoscere chi *non è la madre di un ragazzo*. Siccome abbiamo definito la query `(mother-of-male-child ?x)` proviamo a negare

```
(retrieve (?x) (neg (substitute (mother-of-male-child ?x)))).
> (((?X CHARLES)) ((?X JAMES)) ((?X DORIS)) ((?X EVE))
   ((?X BETTY)) ((?X ALICE))).
```

Possiamo notare che nella risposta è presente anche `((?X ALICE))` che abbiamo gia dimostrato appartenere alla lista di risposta alla query complementare. Per risolvere questo problema dobbiamo applicare un operatore di proiezione prima di costruire la risposta complementare con **`NEG`**. Infatti non vogliamo costruire il complementare di

```
(and (substitute (mother-of ?m ?c)) (?c man)))
```

ma il complementare di

```
(project-to (?m) (and (substitute (mother-of ?m ?c)) (?c man)))
```

che equivale alla seguente espressione

```
(neg (project-to (?x)
     (union (and (neg (?x woman)) (top ?c))
            (neg (?x ?c has-child))
            (and (neg (?c man)) (top ?x)))))).
```

Quindi per ottenere il risultato desiderato utilizzeremo la seguente query:

```
(retrieve (?x) (neg (project-to (?x)
                        (substitute (mother-of-male-child ?x))))).
> (((?X CHARLES)) ((?X DORIS)) ((?X JAMES)) ((?X EVE))
   ((?X BETTY))).
```

3.6 Arricchimento della ABox con semplici regole

Le regole di nRQL hanno un *antecedente* (è un ordinario corpo di query nRQL) e una *conseguenza* (è un'insieme di asserzioni ABox generalizzate). Un'asserzione ABox generalizzata può anche riferirsi a variabili che sono limitate nell'antecedente della regola.

Di seguito è riportata una semplice regola nRQL che promuove una `woman` che non sono ancora note essere `mother` a essere una `mother`:

```
(firerule (and (?x woman) (neg (?x mother))) ((instance ?x mother)))
>(((INSTANCE EVE MOTHER)) ((INSTANCE DORIS MOTHER)))
```

L'antecedente è il semplice corpo di query congiuntive `(and (?x woman) (neg (?x mother)))` e la conseguenza è una singola asserzione d'istanza generalizzata `((instance ?x mother))`. Se la regola è applicata, allora i collegamenti per `?x` dall'insieme di risposta all'antecedente sono usati per istanziare l'asserzione di concetto generalizzato, producendo così un insieme (lista) di asserzioni d'istanza ordinaria sostituendo `?x` con i correnti collegamenti. Il sistema alla fine risponde in modo tale da indicare che le asserzioni ABox sono state aggiunte dalla regola.
Da ora in poi, se si richiede chi è una mother, il sistema risponde in questo modo:

```
(retrieve (?x) (?x mother))
> (((?X ALICE)) ((?X BETTY)) ((?X DORIS)) ((?X EVE))).
```

Queste regole si comportano in maniera non monotona, ovvero non possono essere applicate di nuovo. Quindi ora tutte le `woman` sono considerate essere `mother`, perciò la query `(neg (?x mother))` potrebbe fallire.

3.7 Query su TBox complesse

nRQL può anche essere usate per ricercare modelli di relazioni specifiche tra sotto-classi, classi, super-classi nella tassonomia di una TBox. Per questo scopo viene utilizzato `tbox-retrieve`.

La tassonomia di una Tbox è un cosiddetto *grafo aciclico diretto* (DAG: directed acyclic graph). Un nodo di questi grafi rappresenta una *classe di equivalenza* di *nomi di concetto equivalenti*, e un arco rappresenta una *relazione inclusione-diretta-di*.
Assumiamo inoltre che vengano mantenute le seguenti leggi:

- Ogni nodo x ha un nome, che corrisponde al nome della classe d'equivalenza [x].
- Per ogni nodo x (rappresentante una classe d'equivalenza [x]) e ogni membro x_i appartenente a [x] in questa classe di equivalenza, assumiamo che il predicato $x_i(x)$ è vero.
- Gli archi nella tassonomia che rappresentano relazioni di inclusione-diretta-di sono etichettate per esempio con *has-child* (*has-child(x,y)* → x è inclusione diretta di y).
- Definiamo *has-parent* come la relazione inversa di *has-child* e *has-descendant* come la chiusura transitiva di *has-child*. Quindi *has-ancestor* potrebbe essere la relazione inversa di *has-descendant*.

La struttura relazionale può essere vista come una ABox che è costruita in accordo alle regole sopra esposte ("taxonomy ABox").
Per esempio, se vogliamo ottenere tutti i sotto-concetti diretti della classe `woman` :

```
tbox-retrieve (?y) (and (?x woman) (?x ?y has-child)))
> (((?Y MOTHER)) ((?Y SISTER))).
```

In alternativa può essere utilizzata anche la query

```
 (tbox-retrieve (?y) (and (?x woman) (?y ?x has-parent))).
```

Se invece vogliamo conoscere tutti i sotto-concetti di `woman`:

```
(tbox-retrieve (?y) (and (?x woman) (?x ?y has-descendant)))
> (((?Y SISTER)) ((?Y AUNT)) ((?Y *BOTTOM*)) ((?Y MOTHER))
   ((?Y GRANDMOTHER)))
```

Per recuperare tutti i concetti tranne che `woman`:

```
(tbox-retrieve (?x) (neg (?x woman))).
>(((?X *TOP*)) ((?X *BOTTOM*)) ((?X GRANDMOTHER))
  ((?X FATHER)) ((?X MOTHER)) ((?X UNCLE))
  ((?X BROTHER)) ((?X AUNT)) ((?X SISTER))
  ((?X PARENT)) ((?X MAN))
  ((?X PERSON)) ((?X HUMAN))
  ((?X MALE)) ((?X FEMALE)))
```

Per recuperare tutti i nomi di concetto, non possiamo utilizzare la query

```
(tbox-retrieve (?x) (?x top))
> (((?X *TOP*)))
```

che produce solo il concetto `*TOP*` a causa delle quattro regole usate per la costruzione della "taxonomy ABox" . Quindi per recuperare tutti i nomi di concetto della tassonomia, abbiamo bisogno di una sintassi speciale:

```
(tbox-retrieve (?x) (top ?x))
> (((?X *TOP*)) ((?X *BOTTOM*)) ((?X GRANDMOTHER))
   ((?X FATHER)) ((?X MOTHER)) ((?X UNCLE))
   ((?X BROTHER)) ((?X AUNT)) ((?X SISTER))
   ((?X PARENT)) ((?X MAN)) ((?X WOMAN))
   ((?X PERSON)) ((?X HUMAN))
   ((?X MALE)) ((?X FEMALE)))
```

3.8 Sintassi formale di una query

Per costruire la sintassi di una query prendiamo in considerazione i seguenti simboli:

*	significa zero o più occorrenze;
'X'	denota un letterale;
{'X','Y','Z'}	significa scegliere esattamente uno tra i letterali dati 'X','Y','Z'.

L'insieme di oggetti utilizzati all'interno della `<query-head>` dovrebbe essere un sotto-insieme degli oggetti utilizzati all'interno `<query-body>`. Altrimenti verrà segnalato un errore.

```
(retrieve <query-head>
          <query-body>)
-----------
<query-head> -> "(" <head-entry>* ")"

<head-entry> -> <abox-query-object> |
                <data-substrate-query-object> |
                <head-projection-operator>

<abox-query-object> -> <abox-query-variable> |
                       <abox-query-individual>

<abox-query-variable> -> "?"<symbol> | "$?"<symbol>

<abox-query-individual> -> <symbol> (naming a RacerPro ABox
                            individual)
-----------
<data-substrate-query-object> -> <data-substrate-query-variable> |
                                 <data-substrate-query-individual>

<data-substrate-query-variable> -> "?*"<symbol> | "$?*"<symbol>

<data-substrate-query-individual> -> "*"<symbol>
```

```
<head-projection-operator -> "("<attribute-name>
                                <abox-query-object> ")" |

                        "(TOLD-VALUE"
                        "(" <attribute-name> <abox-query-object> "))" |

                        "( { "TOLD-VALUE"   | "TOLD-VALUES"          |
                             "DATATYPE-FILLER" | "DATATYPE-FILLERS" |
                             "FILLER"       | "FILLERS"         |
                             "TOLD-FILLER"  | "TOLD-FILLERS"         |
                             "ANNOTATION"   | "ANNOTATIONS"
                           }
                        "(" { <OWL-datatype-property> |
                              <OWL-annotation-property> }
                            <abox-query-object> "))"
-----------
<query-body> ->
     <empty-query-body>       |
     <abox-query-atom>        |
     <substrate-query-atom>   |

     "(" { "PROJECT-TO" | "PROJECT" | "PI" } <def-query-head>
        <query-body> ")"      |

     "(" { "AND" | "CAP" | "INTERSECTION" } <query-body>* ")"   |
     "(" { "OR" | "CUP" | "UNION" }          <query-body>* ")"  |
     "(" { "NOT" | "NEG" }              <query-body> ")"   |
     "(" "INV"                      <query-body> ")"
-----------
<empty-query-body> -> ""
-----------
<abox-query-atom> ->
"(" { "NOT" | "NEG" } <abox-query-atom> ")"                 |
"(" "INV" <abox-query-atom> ")"                             |
"(" <abox-query-object> <concept-expression> ")"            |
"(" <abox-query-object> <abox-query-object> <role-expression> ")"|
"(" <abox-query-object> "NIL" <role-expression> ")"             |
"(" "NIL" <abox-query-object> <role-expression> ")"             |
"(" TOP <abox-query-object> ")"                             |
"(" BOTTOM <abox-query-object> ")"                              |

"(" <abox-query-object> <abox-query-object>
     "(" "CONSTRAINT"
          <role-chain-followed-by-attribute>
          <role-chain-followed-by-attribute>
          <predicate-expression>
     ")"
")"                                                         |
```

```
"(" "BIND-INDIVIDUAL" <abox-query-individual> ")"           |

"(" { "SUBSTITUTE" | "INSERT" }
     "(" <query-name>
          { <abox-query-object> | <data-query-object> | "NIL" }*
     ")"
")"                                                         |

"(" {<abox-query-object> | <data-query-object> }*<query-name>")" |

"(" { "SAME-AS" | "=" | "EQUAL" }
     <abox-query-object> <abox-query-object>
")"                                                    |

"(" <abox-query-object>
     "(" "HAS-KNOWN-SUCCESSOR" <role-expression> ")"
")"
```

3.9 Motore di processamento della query nRQL

Il motore (engine) di processamento della query nRQL implementa il linguaggio nRQL. E' una parte interna di RacerPro che offre vari modi di interrogare una query.

3.9.1 Modi di processamento della query di nRQL

nRQL offre diversi modi per processare una query. I due modi principali sono:

- ***set at a time mode***: viene utilizzato il comando retrieve per recuperare l'insieme di risposta ad una query in un unico insieme.
- ***tuple at a time mode***: a volte potrebbe essere necessario caricare e calcolare le tuple da un insieme di risposta di una query in maniera incrementale. L'utente o l'applicazione può osservare le tuple già recuperate e decidere se è necessario recuperare ancora altre tuple dall'insieme di risposta oppure no.

Per mostrare le differenze tra questi due modi di processare riprendiamo la TBox `family.racer`. Il metodo che viene utilizzato per default *set at a time mode* che, come abbiamo già visto in tutti gli esempi considerati fino ad ora, risponde nel seguente modo:

```
(retrieve (?x) (?x woman))
> (((?X BETTY)) ((?X EVE)) ((?X DORIS)) ((?X ALICE)))
(describe-query-processing-mode)
> (... :SET-AT-A-TIME-MODE ...).
```

Per processare una query secondo il modo *tuple at a time mode*, bisogna eseguire il seguente comando:

```
(process-tuple-at-a-time)
> :OKAY-PROCESSING-TUPLE-AT-A-TIME
(describe-query-processing-mode)
> (... :TUPLE-AT-A-TIME-MODE :EAGER ...)
```

E il risultato della query viene processato così:

```
(retrieve (?x) (?x woman))
> (:QUERY-466 :RUNNING)
(get-next-tuple :last)
> ((?X BETTY))
(get-next-tuple :query-466)
> ((?X EVE))
(get-next-tuple :query-466)
> ((?X DORIS))
(get-next-tuple :query-466)
> ((?X ALICE))
(get-next-tuple :query-466)
> :EXHAUSTED
(get-answer :query-456)
> :NOT-FOUND
(get-answer :query-466)
> (((?X BETTY)) ((?X EVE)) ((?X DORIS)) ((?X ALICE)))
```

In questo caso invece di ritornare l'insieme di risposta alla query, viene restituito l'identificativo della query , che deve essere usato come argomento per le funzioni API come `get-next-tuple`, per identificare la query. Invece l'identificativo `last` si riferisce all'ultima query richiesta.

Il motore nRQL, quando lavora in maniera incrementale, permette di eseguire diverse query concorrentemente, ovvero in parallelo (***multi-processamento delle query***). Infatti è possibile invocare un numero di chiamate per retrieve al motore, e poi richiedere le tuple di queste query in maniera random (utilizzando l'identificativo della query). Per esempio:

```
(retrieve (?x) (?x man))
> (:QUERY-467 :RUNNING)
(retrieve (?x) (?x uncle))
> (:QUERY-468 :RUNNING)
(get-next-tuple :query-467)
> ((?X CHARLES))
(get-next-tuple :query-468)
> ((?X CHARLES))
(get-next-tuple :query-467)
> :EXHAUSTED
(get-next-tuple :query-468)
> :EXHAUSTED
(retrieve (?x) (?x woman))
> (:QUERY-469 :RUNNING)
(retrieve (?x) (?x man))
> (:QUERY-470 :RUNNING)
```

```
(get-next-tuple :query-469)
> ((?X BETTY))

(get-next-tuple :query-469)
> ((?X EVE))
```

Il modo *tuple at a time* può essere processato in due modi differenti:

- *modalità incrementale lazy*, secondo la quale le prossime tuple di una query non sono computate prima che avvenga l'attuale richiesta da parte dell'utente o dell'applicazione. Il thread che computa le tuple (thread che risponde alla query), è messo a dormire (fase di sleep), e viene risvegliato da `get-next-tuple`.
- *Modalità incrementale eager*, secondo il quale il thread non è messo a dormire. Infatti, a differenza del caso precedente, esso continua a calcolare tuple dopo tuple anche se non sono state ancora richieste le tuple future. Tali tuple sono messe in coda per le richieste future.

Le API si comportano nello stesso modo per entrambi i modi. In entrambi i casi il thread muore o quando tutte le tuple sono state calcolate, o quando la query è abortita manualmente dall'utente, o se si è raggiunto il timeout, o se si è raggiunto il massimo numero di tuple richiesto.

3.9.2 Il ciclo di vita di una query

Una query ha un ciclo di vita: viene creata, viene resa attiva (quando computa le tuple), viene messa a dormire, sarà riattivata (pre computare qualche altra tupla), etc., e alla fine muore. RacerPorter provvede il tab "Query" che può essere usato per ispezionare e gestire le query come pure il loro stato corrente.

Internamente il motore mantiene un certo numero di liste per mantenere il ciclo di vita delle query

1. *Una lista di tutte le query*: include le query che non sono state ancora inizializzate, quelle che sono in esecuzione o in attesa, e quelle che sono state già processate e quindi terminate. La funzione API che ritorna questa lista è `all-queries`. Per gestire la lista utilizziamo `delete-query`, `delete-all-query`, etc.
2. *Query che sono pronte per essere eseguite,* ma non sono state ancora inizializzate (query pronte o preparate). Non è stato creato ancora nessun thread per loro. Se usiamo `retrieve`, la query viene inserita automaticamente nella lista delle query attive. Invece per inserire una query all'interno di questa lista dobbiamo utilizzare la funzione API `prepare-abox-query`, e per eseguirla `execute-query`.
3. *Una lista di query attive*: query che sono state già inizializzate. C'è un thread associato con ogni query che è sia correntemente in esecuzione, quindi consuma CPU, sia correntemente a riposo. Per ottenere questa lista possiamo utilizzare la funzione `active-query`. Questa lista è ulteriormente suddivisa in:
    a. *Query che sono correntemente in esecuzione*: il thread ad esse associate sta consumando il tempo della CPU per computare le prossime tuple. Per recuperare tale query utilizziamo la funzione `running-queries`.

b. *Query che sono correntemente in attesa (a riposo)*: una query aspetterà fino a che non venga richiesta un'altra tupla dall'utente o dall'applicazione. Per recuperare questa lista utilizziamo `waiting-queries`.

4. *Lista di query già processate (terminate, inattive)*: query il cui insieme di risposta è stato computato esaustivamente, oppure se è stato raggiunto un timeout, o se la query è stata abortita, oppure se si è raggiunto il massimo numero di tuple richieste. Possiamo utilizzare indifferentemente le funzioni `processed-queries`, `inactive-queries` o `terminated-queries` per ottenere questa lista.

E' possibile inserire nuovamente una query processata nella lista delle query pronte utilizzando la funzione `reprepare-query`.

Le stesse liste vengono mantenute per le regole che possono essere ispezionate e gestite tramite il tab "Rules".

3.9.3 Configurare il grado di completezza di nRQL

In tutti gli esempi visti finora noi abbiamo utilizzato il modo completo di nRQL, secondo il quale nRQL utilizza le funzioni base di interrogazione della ABox in modo da raggiungere la completezza. Comunque per alcune ABox questo grado potrebbe essere pesante. Infatti l'ABox potrebbe diventare troppo grande per essere controllata per la consistenza con RacerPro. Se il controllo della consistenza diventa impossibile, anche le funzioni base di recupero della ABox che sono invacata da nRQL durante un recupero, come concept-instances, potrebbero fallire. In questo caso, quando il ragionamento e i servizi di recupero sulla ABox diventano praticamente indisponibili a causa dell'alta complessità, potremmo considerare la possibilità di utilizzare un *modo incompleto* di nRQL per recuperare un individuo sulla ABox. Comunque tali modi incompleti potranno risultare completi per quei tipi di ABox che sono semplicemente strutturate (per esempio ABox che contengono solo dati, e non contengono disgiunzioni o restrizioni di ruolo).

nRQL offre i seguenti gradi di completezza che sono discussi in ordine crescente di completezza:

- ***Mode 0****: interrogazione di informazioni note*. Viene restituita solo l'informazione sintattica, nota che è data dalle asserzioni della ABox. Per esempio, se la ABox contiene `(instance betty mother)`, allora `(retrieve (?x) (?x mother))` ritornerà correttamente `(((?X BETTY)))`, ma la query `(retrieve (?x) (?x woman))` fallirà, perché l'informazione non è data esplicitamente. Il modo quindi è severamente incompleto.
  Riguardo alla struttura relazionale della ABox che è generata dalle asserzioni di ruolo, nRQL è solo completo in questo modalità se non ci sono restrizioni di numero at-most e se non vengono usati i features. Questo vuol dire che comunque è in grado di catturare gli effetti delle gerarchie di ruolo, ruoli chiusi transitivamente e ruoli inversi (logica descrittiva *ALCHI*$_{R+}$). Per abilitare questo modo basta eseguire `(set-nrql-mode 0)`.

- ***Mode 1**: Interrogazione di informazioni note più informazioni della TBox utilizzate per nomi di concetto.* E' come lo scenario precedente, ma adesso l'informazione della TBox è presa in considerazione per asserzioni di appartanenza ad un concetto della forma `(istance i C)` dove C è un nome di concetto. In questo caso il sistema classifica la TBox per computare l'insieme `atomic-concept-synonyms` e l'insieme `atomic-concept-ancestors`, e li inserirà come asserzioni di appartenenza del concetto implicato nella ABox. Per esempio, se la ABox è `(instance betty mother)`, allora verrà aggiunta anche `(instance betty woman)`, visto che woman è membro di `(atomic-concept-ancestors mother)`. In questo caso non fallirà più la query `(retrieve (?x) (?x woman))`, che risponderà correttamente, ma comunque fallirà la query `(retrieve (?x) (?x human))`. Inoltre si hanno a disposizione le regole nRQL per arricchire sintatticamente l'ABox. Per abilitare questo modo basta eseguire `(set-nrql-mode 1)`.
- ***Mode 2**: Interrogazione di informazioni note più informazioni della TBox utilizzate per tutti i concetti.* In questo caso i sinonimi e gli antenati di un concetto atomico saranno computati per asserzioni di appartenenza di concetto arbitrarie, non solo per asserzioni di appartenenza del concetto che si riferiscono ai nomi di concetto. In questo caso non avrebbe più nessun problema neanche la query `(retrieve (?x) (?x human))`. Per grandi ABox che contengono molte espressioni di concetto differenti, questo processo potrebbe richiedere un lungo tempo, visto che ognuna di queste espressioni di concetto devono essere classificate nella TBox. Per abilitare questo modo basta eseguire `(set-nrql-mode 2)`.
- ***Mode 3**: ragionamento su ABox completo*. E' quello che viene utilizzato per default e che abbiamo esaminato fino ad ora. Per ri-abilitarlo `(set-nrql-mode 3)`.
- ***Mode 4 e mode 5***. Per distinguere i due modi dobbiamo introdurre il *modo di processamento della query a due fasi*, che non è altro che uno speciale modo di processamento della query incrementale (tuple-at-a-time). Quindi se nRQL è usato in questo modo, la consegna delle tuple avverrà in due fasi

  a. Vengono restituite le *cheap tuple* (le tuple economiche) nella prima fase;

  b. A seguito, nella seconda fase, vengono restituite le *expensive tuple* (le tuple più costose.)

  Le prime vengono restituite senza utilizzare le funzioni di recupero dell'ABox di RacerPro. Le tuple che vengono consegnate nella prima fase di `(set-nrql-mode 4)` sono le stesse che vengono restituite da `(set-nrql-mode 1)`. La stessa relazione vale per `(set-nrql-mode 5) e (set-nrql-mode 2)`.

  Nella seconda fase, una volta terminate le tuple economiche, vengono recuperate attraverso le funzioni di recupero dell'ABox le cosiddette *tuple expensive*, che richiedono un tempo di computazione più lungo.

  E' da notare che questi modi, come anche il modo 3, sono completi, quindi l'insieme totale delle tuple consegnate è sempre lo stesso. L'unica differenza è che i modi 4 e 5 sono lazy e incrementali, ma comunque le tuple non verranno duplicate. Inoltre il

modo 5 ritornerà più tuple nella prima fase del modo 4, ma l'insieme totale sarà sempre lo stesso.

- ***Mode 6***: Anche se è un modo set at a time, utilizza il modo di computazione della tuple a due fasi per ridurre il numero di chiamate alle funzioni di recupero della ABox di RacerPro al minimo. Infatti nRQL assicura che un test dell'ABox costoso sulla tupla viene effettuato (nella seconda fase) se e solo la stessa tupla non è già stata computata nella prima fase, per evitare quando è possibile le funzioni costose di recupero dell'ABox. Quindi, visto che la prima fase è incompleta, si potrebbe verificare che non viene computata nessuna tupla nella prima fase, lasciando tutto il lavoro per la seconda fase. Tuttavia si potrebbe usare questo modo nel caso in cui si incontrano problemi di prestazioni con il modo 3.

3.10 LUBM Benchmark

Il cosiddetto Lehigh University Benchmark (LUBM) fu sviluppato per facilitare la valutazione e la comparazione dei repositories del web semantico in OWL. Esso consiste in un ontologia OWL che modella delle università (con concetti come persone, studenti, professori, pubblicazioni, corsi, etc., e le appropriate reazioni). Un generatore di riferimento scritto in Java è capace di generare *dati estensionali* corrispondenti a questa ontologia (per esempio un insieme di dipartimenti, professori, studenti, corsi, etc.). Sono state definite 13 query di riferimento, da quelle più semplici che vengono calcolate con tecniche che cercano relazioni chiare, fino a quelle che richiedono tecniche complesse di ragionamento OWL per essere calcolate completamente. Queste sono delle semplici query congiuntive che si riferiscono solo a concetti e a nomi di ruolo, e possono essere effettuate con nRQL.
Viene utilizzato un repository LUBM che comprende un'università con 14 dipartimenti. Le query vengono eseguite su repository di dimensione crescente (iniziando da un'università con un solo dipartimento, e aggiungendo un dipartimento alla volta). Il tempo di risposta (preso su una macchina P4 2.8 GHz 1 GB RAM con Linux), è preso 10 volte sui repository di dimensione incrementale, e viene registrato il tempo di risposa medio.
Il benchmark è stato eseguito in tre differenti setting:

1. la Abox non è stata prima realizzata.
2. L'ABox è stata realizzata (vengono aggiunti i congiunti implicati logicamente).
3. Modo di processamento a due fasi incrementale (tuple at a time).

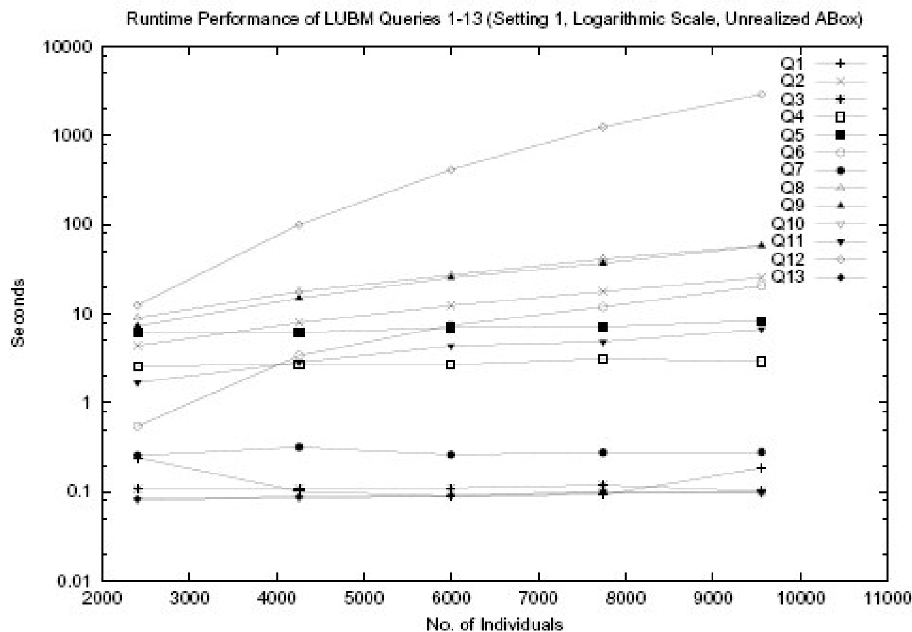
Con la versione 1.8 di RacerPro ancora non è possibile caricare un'intera università dal LUBM a causa del test iniziale di consistenza della ABox che deve essere eseguito prima di iniziare a rispondere alle query. Quindi ci si è dovuti fermare a 5 dipartimenti.

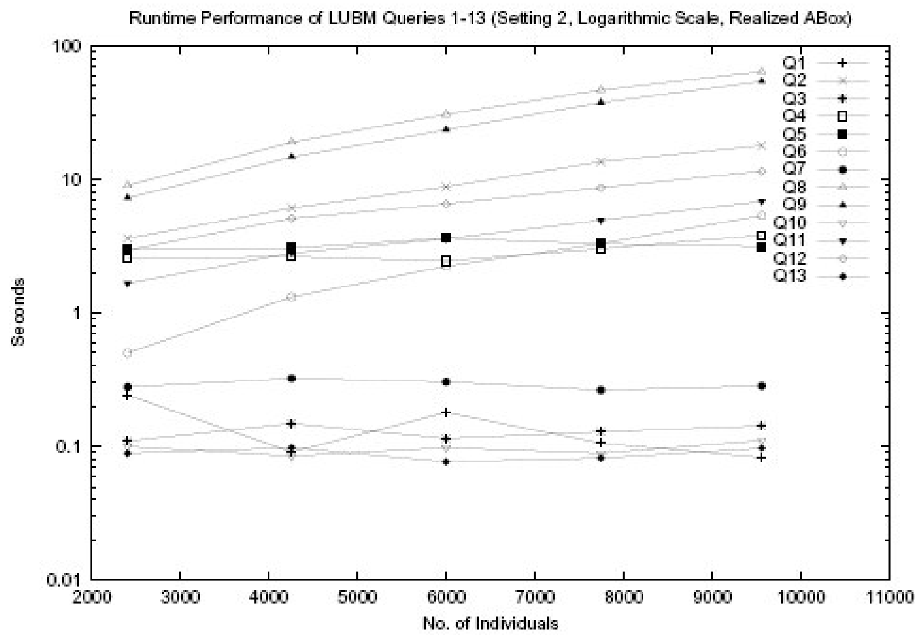
Nel setting 1 il sistema risponde alle query 1,3,7,10 e 13 in una frazione di secondo anche se sono caricati 5 dipartimenti. Queste query, cosi come anche la 4 e la 5, mostrano un comportamento costante nel tempo. Questo non è molto sorprendente, visto che si riferiscono a individui e quindi sono molto specifiche. Per le query 2, 8,9,11 e 12 il tempo di risposta si incrementa in maniera esponenziale. Comunque solo le query 8, 9, 12

richiedono 55 secondi quando ci sono 5 dipartimenti (la 12 richiede addirittura 2912 secondi). Queste sono le query hard nel primo setting.

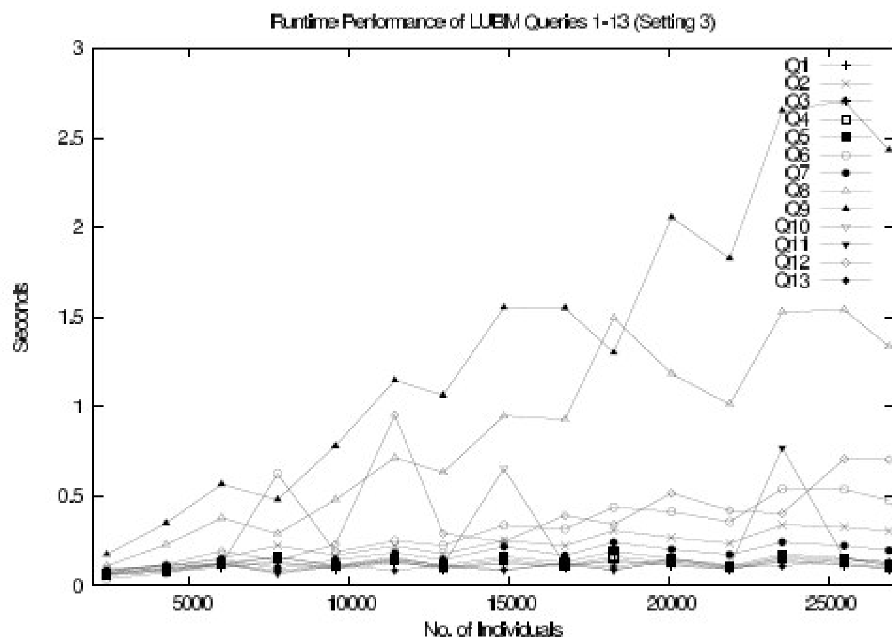
Nel setting 2, la situazione cambia: la query 12 in questo caso non richiede più di 12 secondi. Le query 1, 3, 4, 5, 7, 10 e 13 hanno sempre un andamento costante, ma in questo caso sono leggermente più veloci. L'incremento esponenziale delle query 2, 8, e 9 è leggermente alleviato. Comunque soltanto la query 2 beneficia realmente del secondo setting, in quanto il tempo di esecuzione di tutte le altre è leggermente più veloce. Quindi la realizzazione della ABox migliora il tempo di risposta della query 12, che non potrebbe essere risolta con un DLDB (sistema di databese relazionale come Microsoft Access arricchito con un motore per riscrivere le query DL-based). Inoltre le query 8 e 9 rimangono hard anche per questo setting e quindi sono tra le più hard per DLDB.

Per quanto riguarda il setting 3, l'applicazione client ha la possibilità di caricare gli elementi dell'insieme delle soluzioni in maniera incrementale, aggiungendo una tupla alla volta. Inoltre RacerPro può essere configurato in modo tale che possa rispondere alle query nRQL utilizzando algoritmi ottimizzati noti dai database relazionali. Quindi similmente a DLDB, RacerPro tratta un'ABox come un database e riscrive le query per supportare le informazioni della TBox per il recupero delle informazioni. Oltre alle funzioni del DLDB, RacerPro supporta anche i ruoli inversi in questo processo. Nella figura del setting 3 sono riportati i tempi per recuperare tutte le tuple. Notiamo che RacerPro supporta anche una funzione di query che indica quando terminano le le risposte *cheap.* Dopo questo punto si potrebbe configurare il sistema a restituire tutte le tuple rimanenti.





Runtime Performance of LUBM Queries 1-13 (Setting 2, Logarithmic Scale, Realized ABox)
Seconds
No. of Individuals
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9
Q10
Q11
Q12
Q13

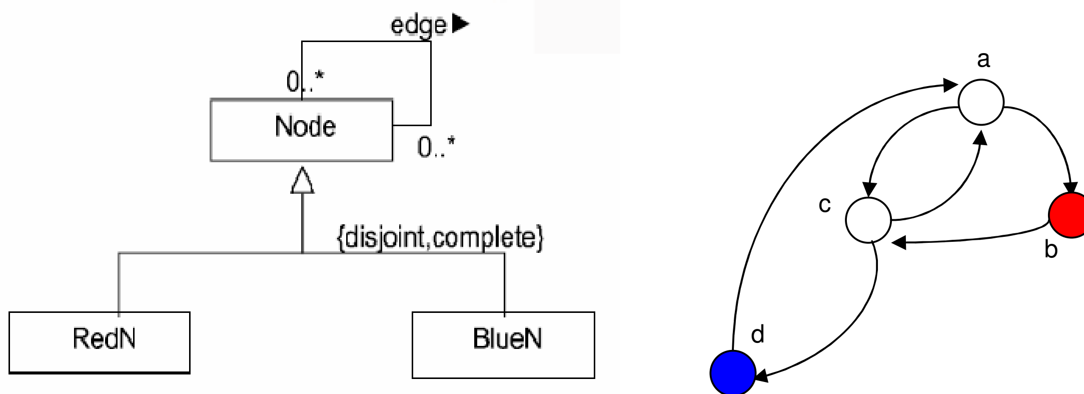


Runtime Performance of LUBM Queries 1-13 (Setting 3)
Seconds
No. of Individuals
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
Q9
Q10
Q11
Q12
Q13

CAPITOLO 4
Casi di studio

4.1 Prima base di conoscenza

La prima cosa che andrò ad analizzare sono le risposte alle query di una base di conoscenza tratta nelle slide dell'articolo *"Epistemic First-Order over Description Logic Knowledge Bases"* di Giuseppe De Giacomo (lavoro congiunto con Diego Calvanese, Domenico Lembo, Maurizio Lenzerini e Riccardo Rosati).
La TBox e la ABox della base di conoscenza che si sta esaminando sono rappresentate rispettivamente nel seguente schema UML e nel seguente grafo:



Base di conoscenza in RacerPro:

```
(in-knowledge-base grafo grafo-node-egde)
(signature :atomic-concepts (node redN blueN)
           :roles ((edge :inverse inverse-edge
                         :domain node
                         :range node))
           :individuals (a b c d))

(implies *top* (and (all edge node)(all inverse-edge node)))
(implies redN node)
(implies blueN node)
(implies redN (not blueN))
(implies node (or redN blueN))

(related a b edge)
(related b c edge)
(related c a edge)
(related c d edge)
```

```
(related d a edge)
(instance b redN)
(instance d blueN)
```

Quindi la base di conoscenza è costituita da quattro nodi (`node`), per due dei quali si conosce il colore, mentre gli altri due devono essere o rosso (`redN`) o blue (`blueN`). I nodi sono in relazione tra loro tramite degli archi (`edge`) .

Prima Query

q(x) :- $\exists$y, z, w. edge(x,y) $\wedge$ edge(y,z) $\wedge$ edge(z,w)

Trovare quei nodi x che possono collegati in catena ad altri tre nodi y, z e w (nella figura il nodo in nero è quello che viene restituito dalla query):

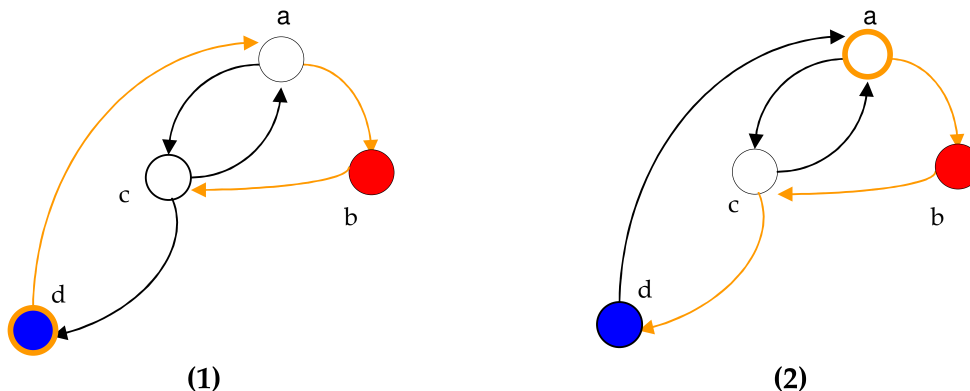
In nRQL (il simbolo '>' indica la risposta del sistema):

```
(RETRIEVE (?X) (AND (?X ?Y EDGE) (?Y ?Z EDGE) (?Z ?W EDGE)))
> (((?X D)) ((?X A)) ((?X C)) ((?X B))
```

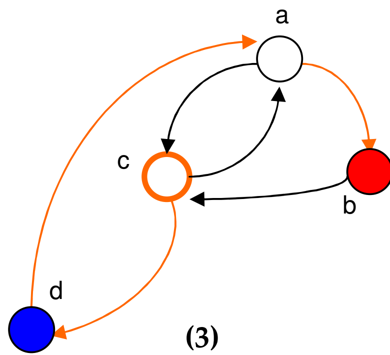
Quindi, in ogni modello, da ogni nodo parte una catena costituita da tre archi. Per osservare quali sono gli archi coinvolti mi faccio restituire l'ordine di tutti i nodi coinvolti in ogni singola catena:

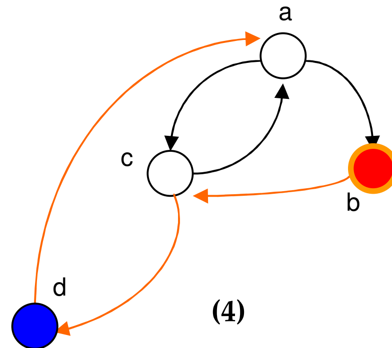
```
(RETRIEVE (?X ?Y ?Z ?W)(AND (?X ?Y EDGE)(?Y ?Z EDGE)(?Z ?W EDGE)))
> (((?X D) (?Y A) (?Z B) (?W C))              (1)
   ((?X A) (?Y B) (?Z C) (?W D))              (2)
   ((?X C) (?Y D) (?Z A) (?W B))              (3)
   ((?X B) (?Y C) (?Z D) (?W A)))             (4)
```

Gli archi che soddisfano la query sono evidenziati in arancione nelle seguenti figure:



(1) **(2)**

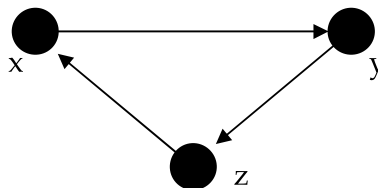




Seconda query

q(x,y,z) :- edge(x,y) ∧ edge(y,z) ∧ edge(z,x)

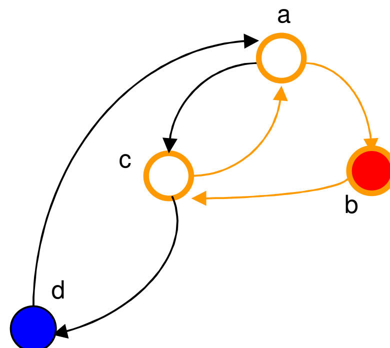
Restituisce tutte le triple di nodi che sono legati nel seguente modo:



In nRQL:

```
(RETRIEVE (?X ?Y ?Z) (AND (?X ?Y EDGE) (?Y ?Z EDGE) (?Z ?X EDGE)))
> (((?X A) (?Y B) (?Z C))                     (1)
   ((?X C) (?Y A) (?Z B))                     (2)
   ((?X B) (?Y C) (?Z A)))                    (3)
```

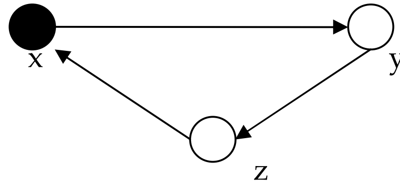
Il sistema fornisce tre liste di nodi differenti che soddisfano la query, che in realtà però individuano sempre lo stesso ciclo di nodi che è evidenziato in figura (differiscono soltanto per il mapping tra le variabili e i nodi del grafo).



Terza query

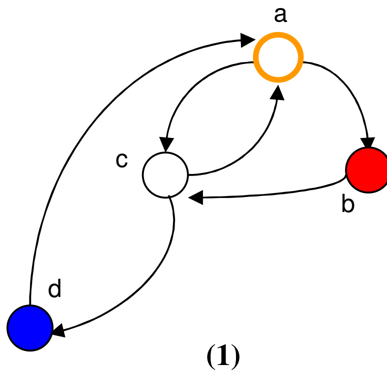
$q(x) \text{ :- } \exists y, z.\ edge(x,y) \wedge edge(y,z) \wedge edge(z,x)$

Restituisce la lista di tutti i nodi da cui partono delle catene cicliche costituite da tre archi:

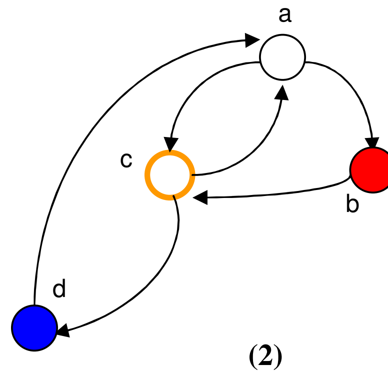


In nRQL:

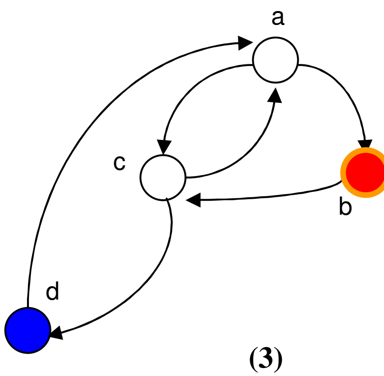
```
(RETRIEVE (?X) (AND (?X ?Y EDGE) (?Y ?Z EDGE) (?Z ?X EDGE)))
> (((?X A))                    (1)
   ((?X C))                    (2)
   ((?X B)))                   (3)
```



(1)



(2)

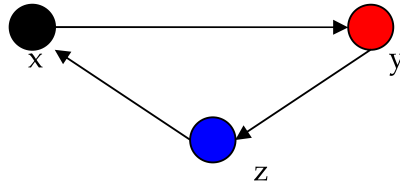


(3)

Quarta query

q(x) :- ∃y, z. edge(x,y) ∧ RedN(y) ∧ edge(y,z) ∧ BlueN(z) ∧ edge(z,x)

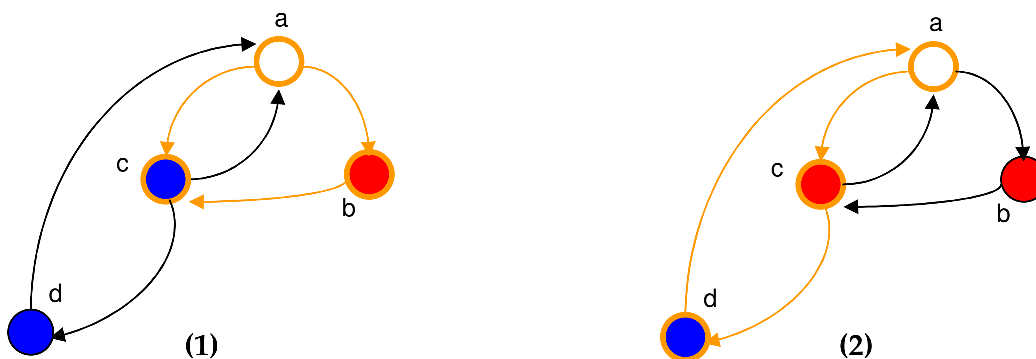
E' simile alla query precedente, con la differenza che il nodo y deve essere rosso mentre quello z deve essere blue.



In nRQL:

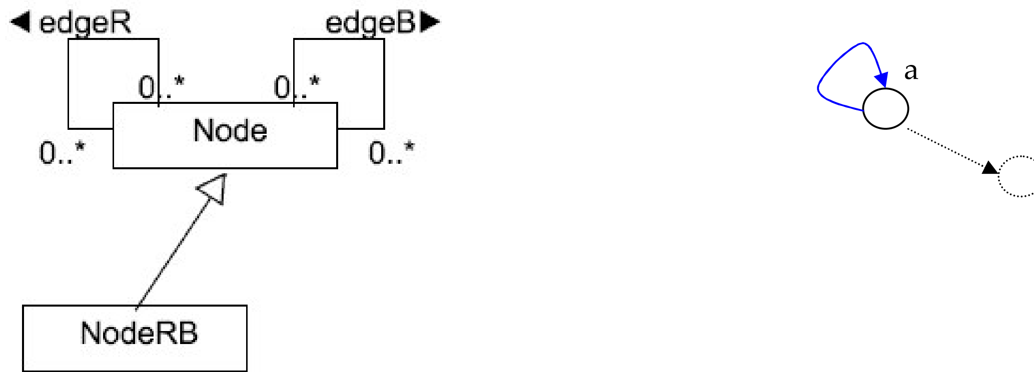
```
(RETRIEVE (?X)  (AND (?X ?Y EDGE) (?Y REDN) (?Y ?Z EDGE)
                     (?Z BLUEN) (?Z ?X EDGE)))
> NIL
```

In questo caso il sistema RacerPro non è in grado di dimostrare l'esistenza di tale cammino. Possiamo però notare che assegnando un colore ai nodi a cui non è stato ancora associato (in particolare al nodo *c*), possiamo creare i diversi modelli che costituiscono la nostra base di conoscenza e osservare che esiste un nodo *x* che soddisfa la query precedente in ogni modelli. Infatti, indipendentemente dal colore del nodo *a,* esiste sempre un cammino come quello richiesto nella query (evidenziati in arancione). E quindi il nodo *a* soddisfa la query in tutti i modelli ma il sistema non è in grado di rilevarlo.



4.2 Seconda base di conoscenza

La seconda base di conoscenza, tratta sempre dallo stesso articolo, è costituita dalla Tbox e dalla Abox riportata in figura:



Base di conoscenza in RacerPro:

```
(in-knowledge-base grafoII grafoII-node-egde)

(signature :atomic-concepts (node nodeRB)
           :roles ((edgeR :inverse inverse-edgeR
                          :domain node
                          :range node)
                   (edgeB :inverse inverse-edgeB
                          :domain node
                          :range node))
           :individuals (a))

(implies (some inverse-edgeR node) node)
(implies (some edgeR node) node)
(implies (some inverse-edgeB node) node)
(implies (some edgeB node) node)
(implies nodeRB (some edgeB node))
(implies nodeRB (some edgeR node))

(related a a edgeB)
(instance a nodeRB)
```

Quindi la base di conoscenza è costituita da un `nodeRB` *a* che ha un arco blue (`edgeB`) verso se stesso, e inoltre deve avere obbligatoriamente almeno un altro arco rosso (`edgeR`).

Prima query

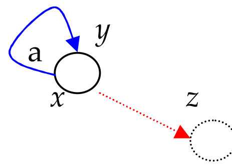
q(x):- $\exists$y, z. edgeB(x,y) $\wedge$ edgeR(x,z) $\wedge$ edgeR(y,z)

Restituire il nodo x che ha un `edgeB` verso un nodo y e un `edgeR` verso un nodo z, e i nodi y e z sono legati tra loro tramite un `edgeR`.

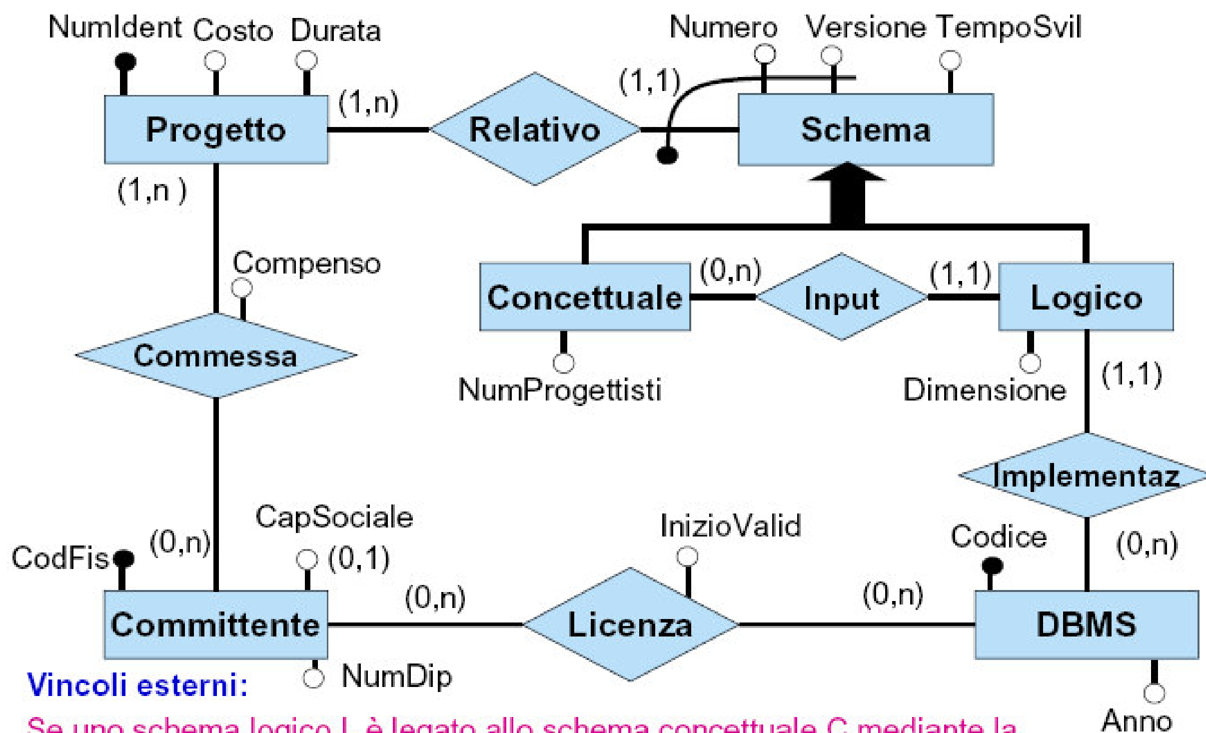
In nRQL:

```
(RETRIEVE (?X) (AND (?X ?Y EDGEB) (?X ?Z EDGER) (?Y ?Z EDGER)))
> NIL
```

Neanche in questo caso il sistema è in grado di dimostrare che il nodo *a* soddisfa la query in ogni modello. Infatti il nodo *a* (a cui associamo la variabile *x*) ha un `edgeB` verso se stesso (quindi anche alla variabile *y* può essere associato il nodo *a*), ma nello stesso tempo deve avere anche un `edgeR` verso un altro nodo (anche verso se stesso), al quale viene associata la variabile *z*, essendo un `nodeRB` (in ogni modello). Quindi poiché anche *y* coincide con il nodo *a*, l'edgeR condiviso da *x* e *z*, sarà condiviso anche da *y* e *z*, come è evidente in figura.



4.3 Esame del 19/12/2002 – Compito A



Prima query

Per ogni schema concettuale che ha richiesto più di 30 giorni per lo sviluppo, si vogliono conoscere i dati relativi al progetto, al numero, e alla versione.

```
(RETRIEVE
  (?X
    (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#schemeNumber| ?X))
    (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#version| ?X))
   ?Y
    (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#identifyNumber| ?Y))
    (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#cost| ?Y))
    (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#duration| ?Y)))
  (AND (?X |http://www.owl-ontologies.com/unnamed.owl#Conceptual|)
     (?X (min |http://www.owl-ontologies.com/unnamed.owl#developmentPeriod| 30))
     (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#related|)))

> (((?X |http://www.owl-ontologies.com/unnamed.owl#Conceptual_12|)
      ((:TOLD-VALUE
        (|http://www.owl-ontologies.com/unnamed.owl#schemeNumber|?X))(234))
      ((:TOLD-VALUE
        (|http://www.owl-ontologies.com/unnamed.owl#version| ?X) ("uml"))
    (?Y |http://www.owl-ontologies.com/unnamed.owl#Project_9|)
      ((:TOLD-VALUE
        (|http://www.owl-ontologies.com/unnamed.owl#identifyNumber|?Y))(456))
      ((:TOLD-VALUE
        (|http://www.owl-ontologies.com/unnamed.owl#cost| ?Y)) (30000.0))
      ((:TOLD-VALUE
        (|http://www.owl-ontologies.com/unnamed.owl#duration|?Y))("16:27:26"))))
```

Notare che in questo caso è stato utilizzato l'operatore `MIN` perché `developmentPeriod` è un datatype di tipo `int`.

Seconda query

Per ogni schema logico di dimensione maggiore di 100, si vogliono conoscere il tempo di sviluppo, ed il costo del relativo progetto.

```
RETRIEVE
 (?X
  (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#developmentPeriod|?X))
  ?Y
  (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#cost| ?Y))))
(AND (?X |http://www.owl-ontologies.com/unnamed.owl#Logical|)
     (?X (MIN |http://www.owl-ontologies.com/unnamed.owl#size| 100))
     (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#related|)))

> (((?X |http://www.owl-ontologies.com/unnamed.owl#Logical_10|)
((:TOLD-VALUE(|http://www.owl-ontologies.com/unnamed.owl#developmentPeriod| ?X))
                                                                          (24))
    (?Y |http://www.owl-ontologies.com/unnamed.owl#Project_3|)
((:TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#cost| ?Y))(15000.0))))
```

Anche in questo caso il datatype di selezione è `developmentPeriod`, quindi l'operatore è `MIN`.

Terza query

Fornire la lista dei progetti per i quali è stato prodotto almeno uno schema logico implementato in un DBMS per il quale almeno un committente del relativo progetto non ha la licenza.

```
(RETRIEVE
   (?X)
   (AND (?X |http://www.owl-ontologies.com/unnamed.owl#Logical|)
```

```
        (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#layout|)
        (?X ?Z |http://www.owl-ontologies.com/unnamed.owl#related|)
        (?W ?Z |http://www.owl-ontologies.com/unnamed.owl#order|)
        (NEG (?W ?Y |http://www.owl-ontologies.com/unnamed.owl#permission|))))
> (((?X |http://www.owl-ontologies.com/unnamed.owl#Logical_13|)))
```

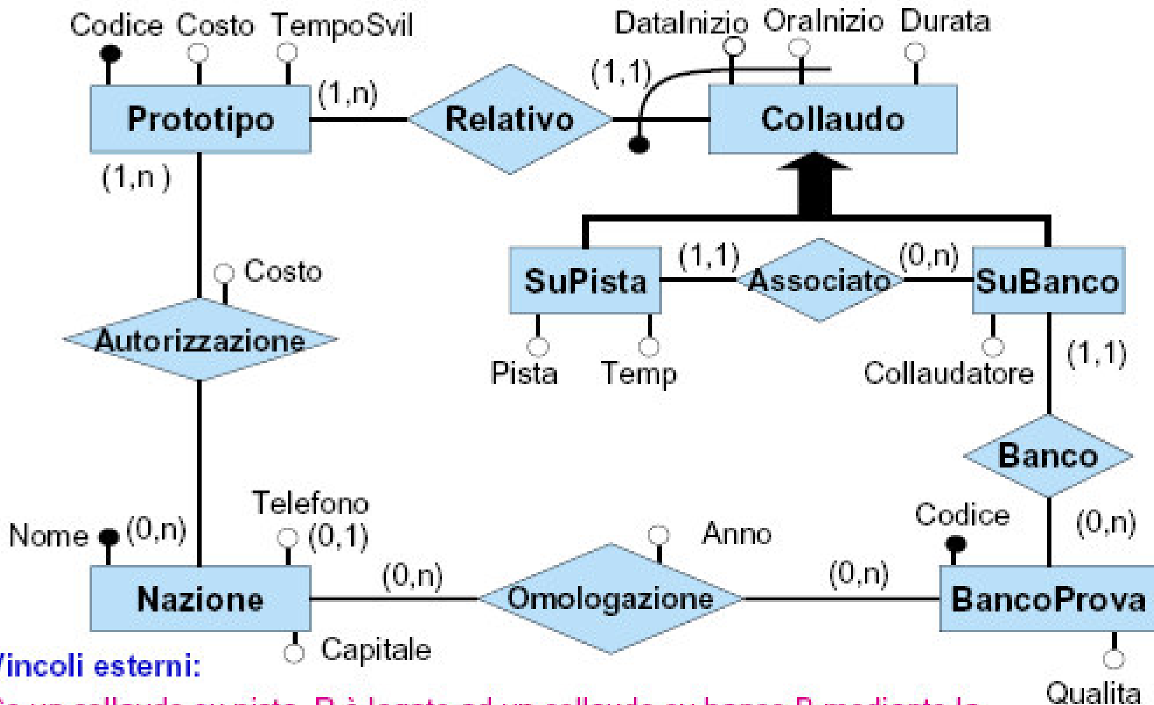
In questo caso è stata utilizzata la negazione come semantica di fallimento.

Quarta query

Produrre la lista di tutti i progetti il cui costo è inferiore al compenso totale che hanno determinato per l'azienda, dove il compenso totale che un progetto determina per l'azienda è semplicemente la somma dei compensi erogati dai relativi committenti per quel progetto.

Non è possibile effettuare la somma dei compensi erogati dai relativi committenti per calcolare il compenso totale di un determinato progetto.

4.4 Esame del 19/12/2002 – Compito B



Prima query

Per ogni collaudo più lungo di 60 minuti, si vogliono conoscere il prototipo collaudato e la data e l'ora di inizio del collaudo.

```
(RETRIEVE
   (?Y ?X
      (TOLD-VALUE
       (|http://www.owl-ontologies.com/unnamed.owl#testBeginningDate| ?X))
      (TOLD-VALUE
       (|http://www.owl-ontologies.com/unnamed.owl#testBeginningHour| ?X)))
   (AND (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#relatedTo|)
```

```
        (?X (> |http://www.owl-ontologies.com/unnamed.owl#lengthTest| 60))))

> (((?Y |http://www.owl-ontologies.com/unnamed.owl#Prototype_3|)
    (?X |http://www.owl-ontologies.com/unnamed.owl#OnCircuit_1|)
      ((:TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#testBeginningDate| ?X))
            ("2007-02-28"))
      ((:TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#testBeginningHour| ?X))
            (14.0)))
   ((?Y |http://www.owl-ontologies.com/unnamed.owl#Prototype_3|)
    (?X |http://www.owl-ontologies.com/unnamed.owl#OnBench_2|)
      ((:TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#testBeginningDate| ?X))
            ("2007-06-28"))
      ((:TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#testBeginningHour| ?X))
            (16.0)))
  ((?Y |http://www.owl-ontologies.com/unnamed.owl#Prototype_8|)
   (?X |http://www.owl-ontologies.com/unnamed.owl#OnCircuit_13|)
      ((:TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#testBeginningDate| ?X))
            ("2007-03-16"))
      ((:TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#testBeginningHour| ?X))
            (10.0))))
```

In questo caso il datatype sul quale viene effettuato l'operazione di selezione è un float, e per questo motivo viene utilizzata l'operatore > .

Seconda query

Per ogni collaudo su pista effettuato ad una temperatura minore di 0, si vogliono conoscere la durata ed il tempo di sviluppo del prototipo collaudato.

```
(RETRIEVE
(?X
(TOLD-VALUE(|http://www.owl-ontologies.com/unnamed.owl#lengthTest| ?X))
 ?Y
(TOLD-VALUE(|http://www.owl-ontologies.com/unnamed.owl#prototypeDevelopmentTime|
                                                                       ?Y)))
      (AND (?X |http://www.owl-ontologies.com/unnamed.owl#OnCircuit|)
           (?X (< |http://www.owl-ontologies.com/unnamed.owl#temperature| 0))
           (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#relatedTo|))

> (((?X |http://www.owl-ontologies.com/unnamed.owl#OnCircuit_1|)
      ((:TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#lengthTest| ?X))(140.0))
    (?Y |http://www.owl-ontologies.com/unnamed.owl#Prototype_3|)
      ((:TOLD-VALUE
      (|http://www.owl-ontologies.com/unnamed.owl#prototypeDevelopmentTime| ?Y))
                                                                    (2)))
   ((?X |http://www.owl-ontologies.com/unnamed.owl#OnCircuit_7|)
      ((:TOLD-VALUE
      (|http://www.owl-ontologies.com/unnamed.owl#lengthTest| ?X))(50.0))
    (?Y |http://www.owl-ontologies.com/unnamed.owl#Prototype_3|)
      ((:TOLD-VALUE
      (|http://www.owl-ontologies.com/unnamed.owl#prototypeDevelopmentTime| ?Y))
                                                                     (2))))
```

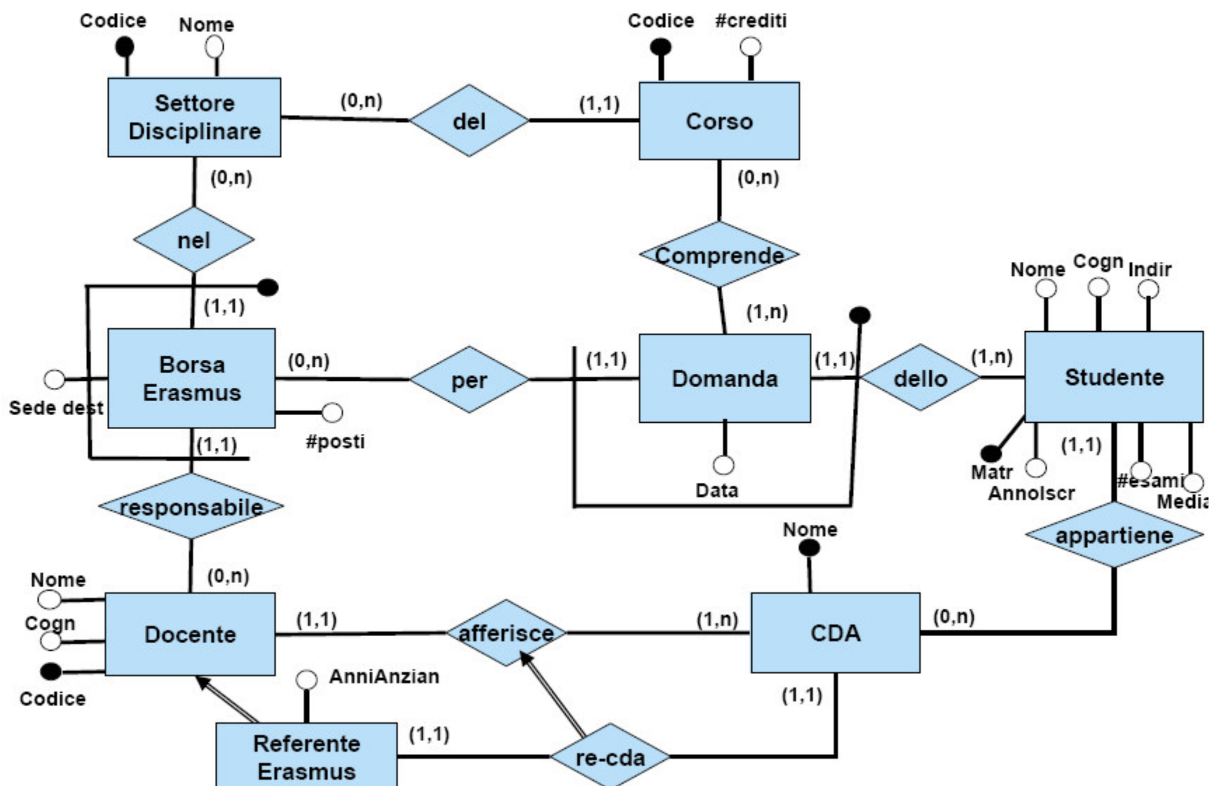
Terza query

Restituire i codici dei prototipi collaudati su almeno un banco che non è omologato in almeno una nazione che ha dato l'autorizzazione al collaudo del prototipo.

```
(RETRIEVE
      ((TOLD-VALUE
            (|http://www.owl-ontologies.com/unnamed.owl#prototypeCode| ?X)))
      (AND (?Y |http://www.owl-ontologies.com/unnamed.owl#OnBench|)
           (?Y ?X |http://www.owl-ontologies.com/unnamed.owl#relatedTo|)
           (?X ?W |http://www.owl-ontologies.com/unnamed.owl#authorization|)
           (?Y ?Z |http://www.owl-ontologies.com/unnamed.owl#bench|)
           (NEG
            (?Z ?W |http://www.owl-ontologies.com/unnamed.owl#typeApproval|))))
```

```
> ((((:TOLD-VALUE
     (|http://www.owl-ontologies.com/unnamed.owl#prototypeCode| ?X)) ("786"))))
```

4.5 Esame del 19/12/2005 – Compito A



Questo schema ER, come tutti quelli utilizzati in seguito, è stato tradotto in un ontologia OWL da Emma Di Pasquale e Teresa Raguso tramite il sistema Protegè. In queste ontologie, io mi sono limitato ad inserire degli individui per poi fare delle query tramite il linguaggio nRQL.

Prima Query

Per ogni domanda, restituire la matricola, la media, il numero di esami superati, l'anno di iscrizione ed il consiglio d'area dello studente che l'ha presentata.

```
(RETRIEVE
((TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#matriculationRoll| ?Y))
 (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#average| ?Y))
 (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#matriculationYear| ?Y))
 (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#examinationNumber| ?Y))
 (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#cdaName| ?Z)))
 (and (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#relatedTo|)
      (?Y ?Z |http://www.owl-ontologies.com/unnamed.owl#belongs|)))

>((((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#matriculationRoll| ?Y)) (793689))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#average| ?Y)) (27.0))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#matriculationYear| ?Y))  (2005))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#examinationNumber| ?Y)) (14))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#cdaName| ?Z)) ("Ingegneria
       Informatica")))
 (((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#matriculationRoll| ?Y)) (794546))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#average| ?Y)) (27.5))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#matriculationYear| ?Y)) (2005))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#examinationNumber| ?Y)) (14))
  ((:TOLD-VALUE
   (|http://www.owl-ontologies.com/unnamed.owl#cdaName| ?Z)) ("Ingegneria
       Informatica"))))
```

Seconda query

Restituire le domande per le quali tutti i corsi che lo studente intende frequentare sono nel settore disciplinare della borsa alla quale si riferisce la domanda.

In questo caso non è possibile indicare il fatto che "tutti" i corsi presenti nelle domande devono essere nel settore disciplinare della borsa alla quale si riferisce la domanda. Quindi basta che ci sia un corso che appartiene allo stesso settore, per fare in modo che la 'domanda' venga inserita nella risposta alla query, anche se nella stessa 'domanda' ci sono corsi che non appartengono a quel settore. Infatti nella risposta alla query viene riportato anche `Demand_1` che tra i suoi corsi riporta anche `Informatica_Teorica` che appartiene al settore `Robotica`, oltre che `Gestione_di_Base_di_Dati` e `Seminario_di_Ingegneria_del_SW` che appartengono allo stesso settore della domanda (`Web_semantico`).

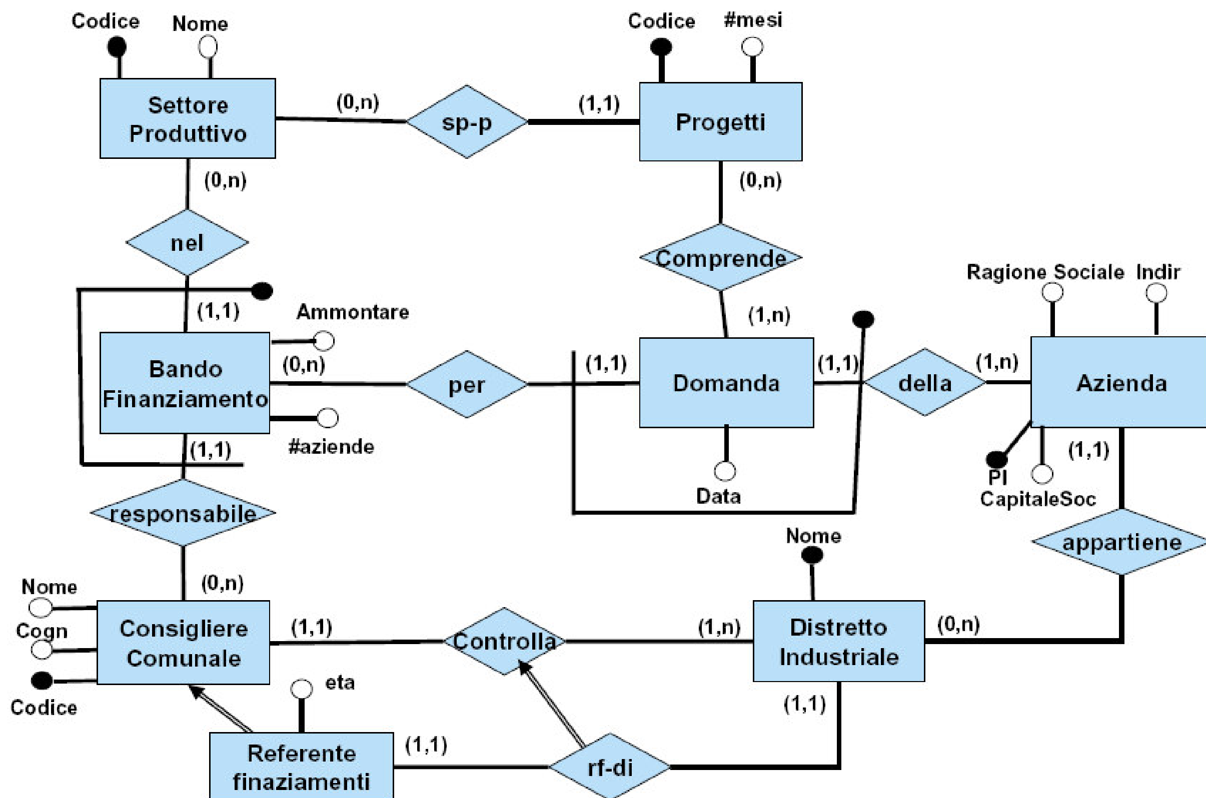
```
(RETRIEVE
      (?Y)
      (AND (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#includes|)
           (?Y ?Z |http://www.owl-ontologies.com/unnamed.owl#for|)
           (?Z ?W |http://www.owl-ontologies.com/unnamed.owl#into|)
           (?W ?X http://www.owl-ontologies.com/unnamed.owl#of|)))

 > (((?Y |http://www.owl-ontologies.com/unnamed.owl#Demand_2|))
    ((?Y |http://www.owl-ontologies.com/unnamed.owl#Demand_1|)))
```

Terza query

Un consiglio d'area viene detto "rilevante per Erasmus" se sono state presentate almeno 10 domande di borse Erasmus da studenti appartenenti al consiglio d'area stesso. Per ciascun consiglio d'area rilevante per Erasmus, calcolare il numero di domande presentate da studenti ad esso appartenenti.

In questo caso non siamo in grado di esprimere il fatto che ci devono essere almeno 10 domande affinché un consiglio sia rilevante per Erasmus. Questo perché il sistema non ci permette di contare le domande.

4.6 Esame del 19/12/2005 – Compito B



Prima query

Per ogni domanda di risposta ad un bando, restituire la partita IVA, il distretto industriale di appartenenza ed il capitale sociale dell'azienda che l'ha presentata.

```
(RETRIEVE
   ((TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#pi| ?Y))
    (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#nameDI| ?Z))
    (TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#socialCapital| ?Y)))
    (AND (?X ?Y |http://www.owl-ontologies.com/unnamed.owl#of|)
         (?Y ?Z |http://www.owl-ontologies.com/unnamed.owl#belongs|)))

>((((:TOLD-VALUE
        (|http://www.owl-ontologies.com/unnamed.owl#pi| ?Y)) (4.3255434E13))
   ((:TOLD-VALUE
        (|http://www.owl-ontologies.com/unnamed.owl#nameDI| ?Z)) ("distretto2"))
```

```
  ((:TOLD-VALUE
       (|http://www.owl-ontologies.com/unnamed.owl#socialCapital| ?Y))
                                                            (500000.0)))
 (((:TOLD-VALUE
       (|http://www.owl-ontologies.com/unnamed.owl#pi| ?Y)) (1.2434355E18))
  ((:TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#nameDI| ?Z))
                                                           ("distretto1"))
  ((:TOLD-VALUE (|http://www.owl-ontologies.com/unnamed.owl#socialCapital| ?Y))
                                                            (1500000.0))))
```

Seconda query

Restituire le domande di risposta a bandi per le quali tutti i progetti indicati dall'azienda proponente sono nel settore produttivo di interesse del bando di finanziamento alla quale si riferisce la domanda.

Non è possibile indicare che "tutti" i progetti indicati devono essere nello stesso settore produttivo del bando a cui si riferisce la domanda. Infatti anche in questo caso basterebbe la presenza di un solo progetto con tale caratteristica per far restituire la domanda nella risposta, anche se contiene altri progetti che non sono nello stesso settore.

Terza query

Un distretto industriale viene detto "rilevante per i finanziamenti"se sono state presentate almeno 10 domande di finanziamento da aziende appartenenti al distretto industriale stesso. Per ciascun distretto industriale rilevante per i finanziamenti, calcolare il numero di domande presentate da aziende ad esso appartenenti.

Non è possibile contare il numero di domande.

Conclusioni

Il lavoro svolto è stato utile per analizzare il potere computazionale di RacerPro e il potere espressivo di nRQL. Abbiamo osservato che RacerPro è in grado di processare documenti OWL-DL, che garantiscono oltre ad un potere espressivo maggiore di quello dei documenti OWL-Lite (presentano solo gerarchie di classi e vincoli poco complessi), anche la completezza computazionale e la decidibilità. Inoltre implementa la logica descrittiva *ALCQHIR+* (*ALC* con restrizioni di numero qualificative, gerarchie di ruolo, ruoli inversi e transitivi).

Abbiamo focalizzato l'attenzione sulla potenza espressiva del linguaggio di query nRQL, che permette di esprimere delle *conjunctive query* e altre query complesse a partire da semplici atomi e utilizzando diversi operatori come `AND`, `UNION`, `INV`, `NOT`, `PROJEC-TO`. Prevede inoltre due tipi di negazione: la classica negazione del vero e la *negazione come semantica di fallimento* (*NAF*), che non è supportata da nessun linguaggio OWL per motivi di decidibilità. Quando si utilizza questa negazione in una query, si opera sotto l'assunzione *Local-Closed World*. Proprio per questa funzionalità, nRQL non è realmente un sotto-insieme di OWL-QL (versione più elevata di OWL). nRQL supporta anche ricerche complesse sui valori degli attributi di un determinato individuo tramite i predicati di dominio concreto. Inoltre permette di fare delle espressioni complesse su documenti RDF e OWL. Caratteristica che rende il linguaggio molto più espressivo dei tipici linguaggi di query RDF, e in certi aspetti anche dei semi-ufficiali linguaggi di query OWL.

Nei diversi casi di studio sono state espresse diverse conjuctive query che hanno evidenziato il limite computazionale di RacerPro, che in alcune circostanze non è in grado di dimostrare la validità di una query. Esso infatti ritorna NIL, che per l'*open world assumption* sta a significare che il sistema non è in grado di provare la validità della query per l'informazione data al sistema stesso. Ciò potrebbe essere dovuto al fatto che le variabili possono essere associate soltanto a individui modellati esplicitamente nella corrente ABox (*semantica del dominio attivo*).

Infine ho notato che il test iniziale di consistenza dell'ABox, che deve essere effettuato prima di computare la risposta ad una query, limita il sistema che non è in grado di utilizzare delle KB molto grandi (non più di 10.000 individui). Nel caso in cui sarà necessario utilizzare un base di conoscenza di dimensioni maggiori, potrebbe essere utile configurare il sistema per lavorare in *modo incompleto*, dando così la possibilità all'applicazione client di recuperare degli individui della tassonomia.

Riferimenti bibliografici

- G. De Giacomo, *dispense del corso di "Seminari di ingegneria del software"* anno accademico 2005-2006, http://www.dis.uniroma1.it/~degiacomo.

- Racer Systems GmbH & Co. KG, *"RacerPro User's Guide"* Version 1.9 8 Dicembre 2005, http://www.racer-systems.com

- Racer Systems GmbH & Co. KG, *"RacerPro Reference Manual"* Version 1.9 8 Dicembre 2005, http://www.racer-systems.com

- G. De Giacomo, *"Epistemic First-Order Queries over Description Logic Knowledge Bases"* 1 Giugno 2006, Internetional Workshop on Description Logics

- Volker Haarslev, Ralf Moller, and Michael Wessel, *"Querying the Semantic Web with Racer + nRQL"*

- Volker Haarslev, Ralf Moller, and Michael Wessel, *slide dell'articolo "Querying the Semantic Web with Racer + nRQL"*

- E. Di Pasquale, T. Raguso, *"Traduzione di diagrammi ER in Protegè"* anno accademico 2005/2006, Seminario di Ingegneria del Software