UNIVERSITÀ DEGLI STUDI DI ROMA TOR VERGATA

FACOLTÀ DI SCIENZE MM.FF.NN
Corso di Laurea Triennale in Informatica

Progettazione e sviluppo di un *reasoner* configurabile in un *framework* di *ontology editing*

Relatore
Prof.ssa Maria Teresa Pazienza

Candidato
Giovanni Lorenzo Napoleoni

A.A 2013/2014

dedica A
dedica B
dedica C

Ringraziamenti

« Propongo di considerare questa domanda: "Le macchine sono in grado di pensare?" »
Alan Turing

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Nell'era odierna, l'uso di strumenti informatici per la ricerca di informazione nel web, ha permesso di rendere la conoscenza presente all'interno di internet alla portata di tutti. La necessità degli utenti di effettuare ricerche sulla rete internet ha reso il motore di ricerca *Google* il sito più visitato al mondo[**1**], dimostrando che una delle principali funzionalità della rete internet è la ricerca di informazione. Questo aspetto rispecchia in pieno il mutamento di pensiero che si è avuto nell'utilizzo del computer: dalla visione di un computer utilizzato per meri calcoli computazionali, si è passati ad una visione in cui il computer è

"an entry points to the information highways"[**10**].

Le ricerche, fondamentalmente,[1] vengono effettuate tramite l'ausilio di alcune parole chiavi: più parole chiavi si inseriscono, più si tenta di affinare la ricerca. Analizzando in modo molto semplicistico l'algoritmo utilizzato dal motore di ricerca di *Google* (per una spiegazione esaustiva si faccia riferimento a [**2**]), possiamo dire che i risultati visualizzati vengono scanditi e filtrati attraverso la corrispondenza delle parole chiavi inserite dall'utente e le parole chiavi all'interno dei testi presenti in internet. Utilizzando la terminologia propria dell' *information retrieval*[**3**], le ricerche basate su parole chiave hanno un *high recall* e una *low precision,* ossia a fronte di un grande numero di risultati, solamente pochi sono rilevanti ai fini della ricerca[**3**], è quindi compito dell'utente ricercare all'interno dei risultati restituiti l'informazione che più si avvicina alla sua ricerca iniziale. Ricerche del tipo:

" voglio tutti i vini che si abbinano con il pesce spada"

restituiscono risultati che si discostano molto da quelli aspettati: la ricerca che è stata appena espressa è di tipo semantico, ossia basata sul significato e non sulle parole chiavi. Infatti, i risultati restituiti dalla ricerca potrebbero contenere pagine web che trattano di ambienti marini (corrispondenza con la parola chiave "pesce spada") o pagine di ristoranti (corrispondenza con "vini" e "pesce") o, se siamo fortunati, esiste una pagina che contiene un determinato vino da abbinare con quel tipo particolare di pesce. Per eseguire ricerche come quelle descritte sopra, è necessario avere un collegamento e una interoperabilità dei dati presenti in internet: la semplice corrispondenza con la parola chiave non basta. Ad esempio, l'applicazione che effettua la ricerca, deve sapere che ci stiamo riferendo ad un pesce, non come un animale ma come un piatto di cucina e che non cerchiamo un ristorante in cui ci sia un buon vino e del buon pesce, ecc...ecc... . È necessario, quindi, per far fronte a ricerche più complesse,[2] che la rappresentazione dei dati all'interno del *web* sia contenuta in una determinata struttura che permetta all'applicazione di usare tecniche intelligenti per il *processamento* dei dati: se per esempio abbiamo un sito web di un hotel, definito tramite l'uso di particolari costrutti o annotazioni sui dati che specificano in maniera distinta il nome dell'hotel, dove è situato, il numero delle stanze ecc..ecc.., l'applicazione potrebbe gestire la ricerca dei dati puntando direttamente sull'annotazione che specifica il particolare dato, come il nome, piuttosto che ricercare all'interno del documento la corrispondenza tra la parola chiave inserita dall'utente e la parola (se esiste) all'interno di esso.

Fin ora si è fatto solo riferimento alla necessità di definire una struttura per la rappresentazione dei dati, la quale da sola non fornisce una soluzione completa al nostro problema. Deve quindi essere analizzato un altro aspetto fondamentale: la semantica dei dati. Ritornando alla nostra ricerca di prima, tramite la semantica possiamo distinguere se ci stiamo riferendo al pesce come animale o al pesce come piatto di cucina.

A questo punto è lecito chiedersi, come può una macchina (nel nostro caso intendiamo un'applicazione che opera su di essa) interpretare un significato? Una prima risposta risiede nell'applicazione in cui vengono gestiti i dati. Essa può associare a quel particolare dato un significato e delle azioni ad esso corrispondenti: ad esempio, quando viene incontrata la parola "pesce" viene associata automaticamente l'azione di cercare le informazioni all'interno di un database che contiene dei piatti a base di pesce. Questo

[1] Facendo sempre riferimento al servizio di ricerca messo a disposizione da Google

[2] Il concetto può essere esteso a tutte le applicazioni che devono effettuare ricerche sui dati, non solo su quelle effettuate nei motori di ricerca.

approccio, però, comporta che i dati sono dipendenti dall'applicazione ed è solamente questa a gestirli, rendendo difficile l'operazione di costruire un reticolato di informazioni tra un applicazione e un'altra. Per questo motivo, nel tempo, si è andata sviluppandosi una tecnologia che porta il nome di *semantic web* che sopperisse alla mancata gestione delle informazioni strutturate presenti in internet, creando delle semantiche distribuite nel web ed eliminando la dipendenza dei dati dalle applicazioni.

Lo scopo del *semantic web* è quello di trovare una comune rappresentazione dei dati presenti in internet attraverso l'inserimento di particolari *tag* o dati aggiuntivi(*meta dati*) specificandone il contesto semantico in un formato adatto all'interrogazione e interpretazione (motori di ricerca) e, in generale, all'elaborazione automatica[**7**]. Attraverso queste informazioni aggiuntive è possibile catturare i significati dei dati, costruendo una relazione tra di essi e permettendo agli utenti di trovare, condividere e combinare informazioni più facilmente e velocemente. Questo contesto è ottenuto usando delle strutture create *ad hoc* per l'organizzazione delle informazioni, strutture che vengono espresse mediante appositi linguaggi propri della famiglia dei *knowledge representation languages,* come il *Resource Description Framework* (RDF), *Resource Description Framework Schema*(RDFS) e il *Web Ontology Language* (OWL) definiti negli standard del *World Wide Web Consortium* (W3C). Tramite questi linguaggi è possibile costruire dei particolari *file* chiamati ontologie attraverso le quali è possibile definire un insieme di rappresentazioni primitive con cui modellare un dominio di conoscenza[**9**], permettendo di specificare un vocabolario con il quale produrre delle asserzioni che possono essere date in input o in output ad applicazioni software fornendo servizi come le risposta a *query* di ricerca (come la ricerca espressa sopra), la pubblicazione di basi di conoscenza riusabili e offrendo servizi per facilitare l'interoperabilità attraverso sistemi eterogenei. L'importanza delle ontologie è quella di fornire una rappresentazione dei dati ad un livello di astrazione che rende indipendente l'ontologia, e quindi i dati, dall'applicazione in cui viene usata: in questo modo i dati possono essere spostati , tradotti e ricercati indipendentemente dai sistemi sviluppati.

Per esprimere a pieno le potenzialità offerte dal *semantic web* è necessario fornire degli strumenti che permettano di aumentare il grado di relazione e di consistenza all'interno dei dati presenti in un ontologia. Si pensi alla situazione in cui stiamo ricercando un particolare tipo di macchina che abbia la proprietà di essere decapottabile. Se nella nostra ontologia fossero espresse solamente le relazioni, "le macchine decapottabili appartengono alla categoria auto sportive" e " le auto sportive appartengono alla categoria macchine", la ricerca restituirebbe un risultato nullo: da nessuna parte, all'interno dell'ontologia, è espresso che le macchine decapottabili appartengono alle categorie delle macchine, solamente affinando la ricerca con " voglio tutte le macchine che sono di tipo sportivo e sono decapottabili" avremmo come risposta un risultato accettabile. Invece, aggiungendo la relazione "le macchine decapottabili appartengono alla categoria macchine" la ricerca iniziale avrebbe un esito positivo. Questa nuova relazione, invece di essere aggiunta a mano, può essere derivata esaminando semplicemente le informazioni esistenti, ossia sapendo che le *macchine decapottabili* appartengono alla categoria delle *macchine sportive* e le macchine sportive appartengono alla categoria *macchina*, potremmo inferire che le macchine decapottabili facciano parte della categoria *macchina*. È auspicabile che le relazioni che possono essere derivate da quelle presenti all'interno di un ontologia, vengano esplicitate mediante strumenti automatizzati. Scopo di questa tesi sarà proprio quello di fornire una tecnologia che, a partire dalle informazioni esistenti, sia in grado di generare nuove relazioni all'interno di ontologia, tramite processi automatizzati di *reasoning*. Questa tecnologia chiamata *reasoner*, attraverso la specificazione di regole di inferenza in cui verranno definite le operazioni per derivare nuove informazioni, sarà in grado di utilizzare tali regole per portare a compimento l'obiettivo di rendere esplicite le conoscenze che sono contenute all'interno dell'ontologia e di far visualizzare all'utente i vari processi che hanno portato alla scoperta di questa nuove informazioni.

Nei prossimi capitoli verranno illustrate le motivazioni della costruzione di un nuovo *resoner*, i processi di implementazione dell'architettura *software*, le scelte implementative, l'analisi della *performance* ed infine l'integrazione del *resoner* all'interno di un sistema concreto di gestione della conoscenza sviluppato dal gruppo di intelligenza artificiale dell'università di Roma Tor Vergata.

Parte I Dominio Del Sistema

In questa parte verranno esaminati gli elementi che compongono il dominio dell'applicazione. Capire il motivo per cui viene progettato il sistema e definire l'insieme degli oggetti con il quale lavora aiuterà a delineare in maniera precisa l'architettura del sistema e di conseguenza la sua l'implementazione. Inoltre, senza una buona definizione del dominio dell'applicazione potrebbe essere difficile far sì che il sistema lavori in modo soddisfacente. Nel primo capitolo riguardante il cuore dell'applicazione verranno spiegate le operazioni per le quali il sistema viene progettato e successivamente saranno illustrate le motivazioni della creazione di un nuovo *reasoner* paragonandolo con i sistemi già disponibili in commercio. Nel secondo capitolo, invece, verranno esaminate, con una visione ad alto livello, le regole di inferenza con le quali il *reasoner* esegue le proprie operazioni.

CAPITOLO 1

Reasoner

1.1. Cosa è un *reasoner*

Un *reasoner* è un software in grado di inferire conseguenze logiche a partire da un insieme di assiomi o di fatti asseriti. Per comprendere meglio il lavoro svolto dal *reasoner* si consideri il seguente esempio:

Tutti gli uomini sono mortali.
Socrate è un uomo.

Da queste due asserzioni o premesse si può inferire che Socrate è mortale. Il compito del *reasoner* si riduce ad attuare questo processo di inferenza attraverso delle opportune regole che ne specificano il funzionamento, generando nuove informazioni[1]. Mentre con il nostro ragionamento siamo in grado di carpire le informazioni implicite nascoste all'interno delle premesse, una macchina, da sola, non avendo a disposizione gli strumenti necessari per eseguire determinati "ragionamenti", non è in grado di farlo. Con l'utilizzo del *reasoner* si cerca di fornire alla macchina quella serie di strumenti necessari ad esplicitare tutte le informazioni, che saranno così a disposizione per eseguire operazioni di ricerca. Quindi, se il *reasoner* ricevesse in input le asserzioni specificate nell'esempio, restituirebbe una nuova asserzione in cui verrà specificato che "Socrate è mortale", aggiungendola alle informazioni già in possesso della macchina; possiamo dire, in termini astratti, che la macchina, "ragionando" sulle informazioni a sua disposizione, è riuscita quindi a trovare nuove informazioni nascoste all'interno dei suoi dati, salvandole all'interno della propria memoria.

Nel contesto del *web semantico* e in particolare per aumentare il grado di relazione dei dati presenti all'interno delle ontologie, il processo di "ragionamento" apporta notevoli benefici in quanto permette di esplicitare quella conoscenza presente all'interno di un'ontologia che altrimenti resterebbe nascosta agli agenti che usano *una query* di ricerca su di essa; in questo modo, quando viene richiesta dell'informazione all'ontologia, l'ontologia è in grado di restituire i risultati corretti.

Come abbiamo detto il *reasoner* utilizza delle regole per scoprire nuove informazioni e nel capitolo 2 verranno analizzate in dettaglio queste regole e definiti i meccanismi che gestiscono i processi di inferenza. Ora è opportuno analizzare le motivazioni che hanno portato alla costruzione di un nuovo *reasoner*.

1.2. Perché costruire un altro *reasoner*

In commercio esistono molti *reasoner engine* (per una lista esaustiva, fare riferimento al link reasoners list) [2] e viene quindi da chiedersi perché sviluppare un *reasoner ex novo*. L'utilizzo di un *reasoner* piuttosto che un altro può dipendere da una serie di fattori che vengono valutati anche in base alle proprie esigenze. Ad esempio si consideri fattori come:

- *software open source* o *software cloused source*
- linguaggio di implementazione
- portabilità
- *knowledge representation languages* utilizzato
- *performance* del *reasoner*
- regole di inferenza usate

L'ideale sarebbe avere a disposizione un *reasoner* che racchiuda le proprietà descritte sopra: che sia *open source*, che utilizzi un linguaggio ampiamente utilizzato in ambito di sistemi *software* permettendone così un'alta portabilità, che permetta l'esecuzione del sistema indipendentemente dal *knowledge representation*

[1] D'ora in avanti con il termine "nuova informazione" o "nuova tripla" si intenderà (salvo quando è indicato diversamente) la trasformazione di un informazione implicita in una informazione esplicita. Nel nostro contesto, trasformare un informazione implicita in una esplicita consiste nel "mettere a conoscenza" la macchina di un nuovo dato che sì contenuto all'interno dei dati esistenti ma che il sistema non riusciva a vedere.

[2] All'interno del testo si farà sempre riferimento a software di tipo *open source* per via della loro accessibilità da parte di tutti

languages utilizzato per rappresentare i dati all'interno dell'ontologia[3], che offra buone prestazioni in termini di performance ed infine che dia la possibilità all'utente che utilizza il *reasoner* di specificare le proprie regole di inferenza. Purtroppo, ad oggi è difficile trovare dei *reasoners* che abbraccino appieno tutte le proprietà elencate. Ad esempio il *reasoner Fact++* (http://owl.man.ac.uk/factplusplus/) è *open source,* ma non si possono specificare le regole di inferenza o un altro esempio è *HermiT* (http://hermit-reasoner.com/) che permette di specificare le regole di inferenza, ma che obbliga l'utente a conoscere il linguaggio *SWRL* (http://www.w3.org/Submission/SWRL/) mentre il nostro obiettivo è rendere le regole di inferenza in un formato di facile scrittura e comprensibilità. Inoltre, sia *Fact++* sia *HermiT* e un altro *reasoner* che porta il nome di *Pellet*(http://clarkparsia.com/pellet/) si avvicinano abbastanza alle proprietà che abbiamo elencato sopra ma non danno la possibilità di "vedere" il processo di *reasoning*: viene solamente mostrato il risultato finale dell'operazione di *reasoning,* ossia le informazioni esplicitate, non permettendo all'utente di capire in quale maniera sono state generate. L'avere a disposizione la possibilità di vedere i passaggi che hanno portato alla scoperta di nuove relazioni, oltre a mostrare i processi di ragionamento effettuati dal *reasoner*, è importate al fine di implementare un processo di *debug* per il controllo della correttezza dell'ontologia sulla quale si sta operando: ad esempio, mentre ci si aspetta un certo risultato, dopo avere attuato il processo di *reasoning,* il risultato è un altro (si è commesso un errore nella specifica di un'informazione). Inoltre, i *reasoner* disponibili, anche se sono portabili, sono pensati per esprimere il massimo delle loro potenzialità all'interno di ambienti ben definiti, come ad esempio il *reasoner* disponibile per *Jena*; è importante, quindi, avere a disposizione un *reasoner* che sia in grado di mantenere un buon bilanciamento tra la performance, la funzionalità e l'ottimizzazione, sia all'interno di ambienti ottimizzati sia in ambienti esterni.

Come abbiamo detto, anche se alcuni *reasoner* permettono all'utente di specificare delle proprie regole di inferenza, esse dipendono comunque dal linguaggio utilizzato per implementarle e, talvolta, l'aggiunta di tali regole comporta una modifica al codice di programmazione con cui viene scritto il *reasoner* come avviene per il *reasoner* di *Jena* [**16**]. Inoltre non solo all'utente è richiesto la conoscenza del linguaggio con cui è implementato il *reasoner* o le regole di inferenza, ma è anche costretto a dovere ripetere tutto il processo di compilazione del sistema ogni volta che le regole di inferenza vengono modificate, operazione che, se non supportata da una buona documentazione, è molto complessa. Il nostro obiettivo, invece, è dare la possibilità all'utente di specificare le regole di inferenza, senza dovere mettere mano al codice di implementazione del *reasoner* e senza dovere conoscere alcun linguaggio di programmazione: esso dovrà solamente attenersi ad alcuni vincoli per la definizione della struttura delle regole.

L'obiettivo finale di questa tesi è la costruzione di un *reasoner* che riesca a contenere tutte le proprietà che sono state elencate all'inizio del paragrafo, cercando di trovare una giusta via di mezzo tra le performance e l'espressione delle funzionalità offerte dal sistema. Il *reasoner* costruito verrà poi inserito all'interno della piattaforma per la *knowledge acquisition and management*[**34**] realizzata da *ART Research Group*[**33**] che porta il nome di *Semantic Turkey*. Si deve ricordare che avendo l'utente la possibilità di specificare delle proprie regole di inferenza è impensabile avere la conoscenza di tutti i possibili ambienti definiti da tali regole, pertanto si cercherà di trovare un performance ottimale che valga per tutte le regole di inferenza. Per quanto riguarda la "visualizzazione" del processo di *reasoning*, si darà la possibilità all'utente di scegliere se visualizzarlo o no, in questo caso, in presenza di ontologie molto grandi, disattivando il processo di memorizzazione dei procedimenti di *reasoning* sarà possibile avere un aumento delle performance del sistema (nel capitolo 5 verranno analizzate i costi di computazione delle performance).

[3]Faremo sempre riferimento ai *knowledge representation languages* definiti negli standard del *World Wide Web Consortium* (W3C)

CAPITOLO 2

Le regole di inferenza

Come si è detto nella sezione 1.1, il *reasoner*, per potere funzionare correttamente, necessita di opportune regole in grado di attuare i processi di inferenza: in pratica, un *reasoner* senza regole di inferenza è come una macchina senza un motore. Le regole di inferenza sono il cuore di un *reasoner,* è tramite la loro applicazione che è possibile esplicitare le informazioni all'interno di un'ontologia. Una delle proprietà che deve esprimere il nostro sistema consiste nel dare la possibilità all'utente di aggiungere nuove regole di inferenza e occorre, quindi, definire un formato standard per la definizione delle regole in modo tale da adattare il *reasoner* a questa struttura. Tale definzione è necessaria affinché le regole siano specificate all'interno di un contesto che permetta al *reasoner* di poterle interpretare senza difficoltà. Prima di passare alla definizione di una struttura delle regole di inferenza comprensibili per il reasoner, è utile innanzitutto analizzare la natura delle regole di inferenza fornendo un modello di traduzione da un linguaggio naturale, in cui vengono espresse le regole, ad un linguaggio di più basso livello (come può essere un linguaggio di programmazione), modello che verrà poi utilizzato nella fase implementativa del sistema.

2.1. Le regole di Inferenza nel *semantic web*

Per comprendere la natura delle regole di inferenza, si faccia riferimento alla seguente definizione di inferenza:

"*something that you can find out indirectly from what you already know*"[**5**]

In particolare, l'inferenza all'interno del *semantic web*[1] consiste nel scoprire nuove relazioni o in generale nuove informazioni su proposizioni rappresentate attraverso particolari costrutti. Si consideri l'esempio fatto nella sezione 1.1:

Tutti gli uomini sono mortali e Socrate un uomo.

Supponiamo che la prima e la seconda proposizione siano vere[2], allora possiamo creare una terza proposizione (nuova informazione o relazione) in cui inferiamo che "Socrate è un uomo". Nel *semantic web* i dati, in questo caso le proposizioni, sono modellati attraverso un insieme di relazioni tra risorse, dove le risorse possono essere di qualsiasi natura, come una pagina Web o un elemento di un sorgente XML. Pertanto, l'inferire consiste in una procedura automatica, orchestrata dal *reasoner*, che genera nuove relazioni a partire dai dati esistenti attraverso un insieme di regole dette regole di inferenza. L'inferenza nel *semantic web* è uno degli strumenti che consentono di aumentare la qualità dell'integrazione dei dati, analizzando automaticamente il contenuto dei dati o gestendo la conoscenza del *web*.

Per avere una idea generale di come può funzionare un processo di inferenza all'interno di un ontologia si consideri il seguente esempio: supponiamo che all'interno dell'ontologia siano definite le seguenti relazioni (indipendentemente dal linguaggio di rappresentazione utilizzato):

```
Marco isBrother Fabio.
Fabio isBrother Paolo.
```

dove "*isBrohter*" definisce la relazione "essere fratello di", relazione che sarà presente all'interno del "vocabolario" definito nell'ontologia. Le due premesse implicano che Marco è anche fratello di Fabio, relazione non espressa all'interno dell'ontologia ma nascosta all'interno di essa. Nel caso in cui venisse richiesto all'ontologia di restituire "tutti i fratelli di Marco" tramite una *query* di ricerca, otterremo una risposta incompleta: la risposta sarà costituita solamente dalla relazione "Marco isBrother Fabio". Supponendo di avere a disposizione la regola di inferenza:

if X isBrhoter Y and Y isBrhoter Z then X isBrhoter Z.

[1] Si può facilmente generalizzare per tutti i contesti in cui viene usata l'inferenza

[2] Nella tesi si farà sempre riferimento ad un modo che segue le regole della OWA (*open world assuntion*)[**6**]

dove X,Y e Z rappresentano il soggetto o l'oggetto di una proposizione. Applicando la regola all'ontologia, inferiremo la relazione "Marco è fratello di Paolo" che si aggiungerà alle relazioni già presenti in essa. Questa volta, eseguendo la *query* di ricerca "restituire i fratelli di Marco" avremo finalmente la risposta completa: "Fabio e Paolo".

Le regole di inferenza possono essere rappresentate mediante una determinata struttura. Questa struttura verrà presa in prestito dalla logica formale che esprime i processi di inferenza[**8**] nel seguente modo:

$Premise\#1$
$Premise\#2$
...
$\underline{Premise\#n}$
$Conclusion$

Dove le "*Premise*" e la "*Conclusion*" vengono espresse mediante proposizioni. La regola di inferenza che abbiamo espresso prima può essere riscritta nel seguente modo:

$X\ isBroher\ Y$
$\underline{Y\ isBroher\ Z}$
$Y\ isBrohter\ Z$

Questa notazione implica che le premesse devono susseguirsi in condizioni logiche[**8**] ossia che la struttura appena definita è equivalente alla seguente struttura espressa in predicati logici: $\{P \wedge Q\} \rightarrow Z$. Dove P e Q corrispondono alle due premesse e Z alla conclusione. Naturalmente le proposizioni sono soggette ad una tavola di verità fatta nel seguente modo:

P	Q	$\{P \wedge Q\} \rightarrow Z$
V	V	V
V	F	F
F	V	F
F	F	F

TABELLA 1. Tavola verità

Una volta definita la struttura complessiva delle regole di inferenza, bisogna specificarne le strutture interne delle proposizioni e delle conclusioni.

2.1.1. Le proposizioni e le conclusioni. Dopo la definizione della generica struttura di una regola di inferenza è necessario analizzare i componenti interni alla regola: le premesse e le conclusioni. Questi oggetti fanno parte di uno stesso dominio: sia le premesse sia le conclusioni sono a tutti gli effetti delle proposizioni, cambia solamente il modo in cui vengono analizzate all'interno della regola di inferenza, di conseguenza definire una struttura per una proposizione equivale a definire una struttura per le premesse e per le conclusioni. A differenza della struttura globale delle regole di inferenza, che può essere usata in contesti diversi, la struttura interna delle proposizioni deve, in qualche modo, corrispondere agli oggetti del dominio sui cui operano le regole. Per fare un esempio con il mondo matematico è come se definiamo una funzione che opera sul dominio e codominio dei numeri naturali, e come valori della funzione utilizziamo dei reali. Una tale funzione può anche essere usata ma previa conversione dei reali in un corrispondente (approssimato) dei naturali. Nello stesso modo, essendo le proposizioni quegli oggetti che operano con gli elementi del dominio del sistema, è utile definire la loro struttura il più vicino possibile al dominio sul quale operano, in modo tale da non dovere aggiungere operazioni aggiuntive di conversioni tra gli oggetti utilizzati nella struttura e gli oggetti del dominio. Occorre, prima di definire la struttura interna dei componenti, analizzare quali sono gli oggetti del dominio su cui operano le regole di inferenza e di conseguenza l'intero sistema del *reasoner*.

Come è stato descritto nelle precedenti sezioni, il *reasoner* opera su documenti chiamati ontologie appartenenti all'insieme delle tecnologie utilizzate nel *semantic web*. Un'ontologia, usando la definizione di *Tom Gruber*, è "*a formal specification of a shared conceptualization* ", in altre parole viene utilizzata per descrivere un dominio con una lista di termini e di relazioni tra essi fornendo in pratica gli strumenti per descrivere gli elementi del domino stesso[**19**]. Un esempio di ontologia potrebbe essere la descrizione del dominio che concerne i "liquidi" , mentre un esempio della definizione di una classe di oggetti appartenenti al dominio dell'ontologia è descritta nel listato che identifica quella classe di oggetti che sono i liquidi potabili.2.1

LISTING 2.1. Esempio di classe in un ontologia

```
<owl:Class rdf:ID="PotableLiquid">
    ....
</owl:Class>
```

LISTING 2.2. Rappresentazione di uno *statement*

```
<rdf:Statement>
  <rdf:subject rdf:resource="Mela" />
  <rdf:predicate rdf:resource="onto;è" />
  <rdf:object rdf:resource="Frutto" />
</rdf:Statement>
```

Come tutti i testi scritti che usano un propria definizione di linguaggio (grammatica e sintassi) e un proprio vocabolario, anche le ontologie vengono redatte usando un *ontology language* e un *ontology vocabolary*: è quindi possibile avere una stessa ontologia descritta con diversi linguaggi e diversi vocabolari. A differenza dei linguaggi comuni, che in alcuni casi possono essere totalmente diversi (ad esempio come l'italiano e il giapponese), tutti i linguaggi usati per descrivere le ontologie usano uno stesso *core* [3]. Avere a disposizione una base comune per tutti gli *ontology languages* facilita di molto la definizione della struttura delle proposizioni delle regole di inferenza: una volta definita una struttura sulla base di questo *core,* le regole di inferenza potranno essere applicate indipendentemente dal linguaggio utilizzato nell'ontologia. Ogni linguaggio, che può estendere il *core* con funzioni aggiuntive*,* può essere sempre riportato o tradotto nella sua implementazione di base, comune a tutti gli *ontology languages*. Ma come è definito questo *core*? Il *semantic web*, nel corso degli anni, ha prediletto un linguaggio che porta il nome di *Resource Description Framework*(RDF)[**20**].

L'*RDF* è un linguaggio che appartiene alla famiglia dei linguaggi di tipo *data model* [**21**]ossia l'insieme di quei linguaggi che si occupano di modellare i dati. La sua struttura base è composta da un tripla *subject-predicate-attribute* chiamata *statement.* Senza entrare nel dettaglio delle specifiche del linguaggio (per una spiegazione esaustiva dell' RDF fare riferimento qui [**22**]), ogni informazione all'interno dell'ontologia è rappresentata tramite un insieme di *statements*. Ad esempio, se dobbiamo rappresentare l'informazione "una mela è un frutto" possiamo suddividere la nostra proposizione in:

$mela \rightarrow subject$
$è \rightarrow predicate$
$frutto \rightarrow attribute$

dove una sua possibile rappresentazione in RDF, espresso tramite i costrutti del linguaggio XML, è mostrata nel listato 2.2. L'RDF è indipendente dal dominio in cui opera, è compito dell'utente definire la terminologia del dominio nei cosiddetti *schema language* chiamati *RDFSchema*(*RDFS*). Questi schemi definiscono il vocabolario del linguaggio usato nei *data models* in RDF. Vediamo come questa struttura può essere applicata alle proposizioni della regola di inferenza.

Per comprendere meglio come è possibile applicare il modello dello *statement* all'interno delle nostre regole di inferenza, utilizziamo l'esempio fatto nella sezione 2.1 in cui si era creata una regola di inferenza di questo tipo:

$X\ isBrohter\ Y$
$\underline{Y\ isBrohter\ Z}$

$Y\ isBrohter\ Z$

Come si può notare le proposizioni e le conclusioni assomigliano alla definizione di *statment* data precedentemente. Le proposizioni all'interno della regola possono essere viste come:

[3]Questa asserzione deve essere valutata in base specifiche ufficiali delle tecnologie e metodi usati dal *semantic web*[**20**].

$X \rightarrow object$
$isBrohter \rightarrow attribute$
$Z \rightarrow value$
$Y \rightarrow object$
$isBrohter \rightarrow attribute$
$Z \rightarrow value$
$Y \rightarrow object$
$isBrohter \rightarrow attribute$
$Z \rightarrow value$

ossia come la suddivisione in triple composte da un soggetto, un predicato e un oggetto del predicato. Supponiamo che questa regola venga applicata alla ontologia definita nel listato 2.3:ogni proposizione della regola di inferenza corrisponderà al suo rispettivo *statement* (se esiste) nell'ontologia. Ad esempio, la proposizione $X\ isBrohter\ Y$ applicata all'ontologia ci restituisce il primo *statement* in cui si definisce che "Mario è fratello di Franco", mentre la seconda proposizione $Y\ isBrohter\ Z$ restituirà lo *statemen*t "Franco è fratello di Pippo" (la spiegazione esaustiva di come verrà effettuata la ricerca degli *statmen-ts* all'interno dell'ontologia sarà spiegata nella parte relativa all'implementazione del sistema, per ora è importante focalizzare l'attenzione sul comportamento generale delle regole di inferenza). Una volta trovato due *statment* che corrispondono alle proposizioni definite nella regola di inferenza (ricordiamo che per applicare una conclusione di una regola di inferenza tutte le premesse devono essere verificate), allora potrà essere applicata la conclusione $Y\ isBrohter\ Z$ sull'ontologia, che produrrà un nuovo *statement* rappresentato nel listato 2.4.

Generalizzando la costruzione delle premesse e delle conclusioni, possiamo dire che quando definiamo una nuova proposizione essa sarà composta da un *subject*, *predicate* e un *attribute*. La proposizione esprime, quindi, la ricerca di tutte le triple all'interno dell'ontologia che corrispondono agli elementi definiti nella proposizione. Definendo le regole in questo modo si rende più facile, in fase di applicazione delle regole di inferenza, tradurre una regola scritta in un linguaggio testuale con degli oggetti che hanno sì lo stesso dominio, ma che sono contenuti in strutture dati utilizzati dalla macchina. Se ad esempio definissimo la proposizione $X\ isBrohter\ Y$ in un linguaggio più naturale come " X deve avere la proprietà di essere fratello di Y", non solo avremo il problema di eseguire una scansione dell'intera proposizione per cercare le parole chiavi come "proprietà","fratello di" ecc... ma successivamente, la proposizione, dovrà essere riportata in un oggetto che assomigli il più possibile agli oggetti del dominio. Deve essere sottolineato che l'uso di una struttura *object-predicate-subject* è limitativa perchè si hanno a disposizione solamente tre "atomi" per definire le nostre proposizioni: nel caso di regole complesse non è molto efficiente come struttura. D'altro canto, gli oggetti dell'ontologia condividono lo stesso problema ed è proprio per questo motivo che sono stati creati dei linguaggi che estendono le funzionalità dell'RDF in modo tale da rendere più espressiva la definizione dei termini dell'ontologia. Siccome dobbiamo essere in grado di applicare le regole di inferenza anche su linguaggi che estendono l'RDF, l'utilizzo di una tale struttura è necessaria per rendere indipendente l'esecuzione del *reasoner* dal linguaggio di rappresentazione utilizzato per descrivere l'ontologia.

LISTING 2.3. Esempio di un'ontologia

```
<rdf:Statement>
  <rdf:subject rdf:resource="Mario" />
  <rdf:predicate rdf:resource="onto;isBrother" />
  <rdf:object rdf:resource="Franco" />
</rdf:Statement>
<rdf:Statement>
  <rdf:subject rdf:resource="Franco" />
  <rdf:predicate rdf:resource="onto;isBrother" />
  <rdf:object rdf:resource="Pippo" />
</rdf:Statement>
```

LISTING 2.4. Statement prodotto dalla applicazione di una regola di inferenza

```
<rdf:Statement>
  <rdf:subject rdf:resource="Mario" />
  <rdf:predicate rdf:resource="onto;isBrother" />
  <rdf:object rdf:resource="Pippo" />
</rdf:Statement>
```

Parte II Architettura e implementazione del *reasoner*

In questa parte verrà illustrato il processo di creazione dell'architettura *software* del sistema e la definizione dei moduli che ne faranno parte. Definire una buona architettura permette di rendere il sistema indipendente dall'implementazione usata per svilupparlo e aiuta a comprendere quali sono le funzionalità che il sistema dovrà esprimere. Inoltre pone il sistema nella condizione in cui possono essere attuati processi di modifica o inserimento di nuovi moduli senza dovere alterare l'architettura stessa. Successivamente saranno mostrate le implementazioni dei moduli definiti nella fase architetturale ponendo particolare attenzione alle relazioni con gli altri moduli del sistema.

Nel primo capitolo dopo avere individuato i requisiti funzionali e del dominio del sistema, verranno illustrate le scelte architetturali fornendo una serie di diagrammi per capire la relazioni, ad alto livello, dei moduli del sistema. Nel capitolo 4, sarà posta l'attenzione sull'implementazione dei vari moduli e sulle loro operazioni, mostrando il processo che li lega allo sviluppo dei requisiti del sistema. Per l'implementazione di ogni modulo saranno descritte le particolari tecnologie utilizzate ed eventuali alternative che sono state prese in considerazione durante la fase implementativa. Infine, nel capitolo 5, saranno analizzate le performance del sistema in base a delle opportune configurazioni dell'ambiente di esecuzione.

CAPITOLO 3

Definizione dell'architettura

In questo capitolo verrà definita l'architettura del sistema e i principali moduli che ne faranno parte. Questo processo verrà gestito tenendo a mente i requisiti funzionali e di dominio del sistema.

3.1. I requisiti del sistema

Prima di iniziare lo sviluppo dell'architettura del sistema è necessario comprendere e definire in maniera chiara e precisa quali sono i requisiti che il sistema deve soddisfare per ridurre gli errori durante la fase progettuale dell'architettura. Avere una precisa dimensione dei requisiti del sistema obbliga l'architetto a valutare in modo efficiente i suoi punti critici e a scegliere un'architettura che, qualora dovessero essere apportati dei cambiamenti durante la fase di sviluppo del software o dopo che l'applicazione fosse rilasciata, supporti, a basso costo, operazioni di modifica del sistema stesso.

I requisiti principali che il sistema deve soddisfare sono i seguenti:

Requisiti Funzionali

- Il sistema deve essere in grado di effettuare operazioni di *reasoning* su un'ontologia data in input al sistema.
- Il sistema deve essere in grado di usare regole di inferenza specificate dall'utente. Regole che verranno usate per l'operazione di *reasoning*.
- Le regole specificate dall'utente devono essere redatte in un file di testo che successivamente verrà passato al *reasoner*.
- Il sistema deve essere in grado di produrre un output in cui vengono memorizzate le informazioni prodotte dal processo di *reasoning*.
- Il sistema deve avere la possibilità di essere configurabile: l'utente può scegliere le regole che devono essere applicate, se l'output deve essere prodotto e quante volte deve essere eseguita l'operazione di *reasoning* con i parametri specificati dall'utente.

Requisiti di Dominio

- I dati primitivi su cui deve operare il *reasoner* e il modello di ontologia sul quale deve effettuare il *reasoning* devono appartenere agli standard definiti nelle API OWL-ART api

Vediamo ora in dettaglio i requisiti espressi nei precedenti punti. Il compito principale del *reasoner* è quello di effettuare operazioni di *reasoning* all'interno di un'ontologia ricevuta in input. Con operazione di *reasoning* si intende una serie di procedure che, utilizzando delle regole di inferenza specificate dall'utente, siano in grado di inferire nuove informazioni a partire da quelle contenute nell'ontologia (come descritto nel capitolo 2) e restituirle in output rispettando i vincoli imposti nell'uso dei dati primitivi del sistema. Tuttavia, trovandoci in processi di inferenza è possibile che tutte le informazioni inferite siano già presenti nell'ontologia e quindi non restituite in output o che sia necessario eseguire l'operazione di *reasoning* più volte per scoprire nuove informazioni. Per avere una visione più chiara dell'operazione di *reasoning* si faccia riferimento alla figura 3.1.1

L'utente ha la possibilità di inserire, in base agli obiettivi descritti nella sezione 1.2, delle regole di inferenza personalizzate in base alle proprie esigenze. Come è stato accennato, la stesura di queste regole dovrà rispettare una struttura che verrà definita in fase di implementazione del sistema. Queste regole, una volta scritte, dovranno essere date in input al *reasoner* che a sua volta provvederà ad interpretarle ed ad usarle nell'operazione di *reasoning*. Durante il processo di interpretazione si controllerà che il documento sia scritto con la giusta struttura, altrimenti l'operazione non potrà proseguire. Il documento contenete le regole di inferenza potrà essere fornito solamente prima dell'inizio dell'operazione di *reasoning* e non potrà essere modificato finché il *reasoner* non avrà concluso il suo *task*. Altro elemento essenziale per il sistema è l'ontologia sul quale effettuare il *reasoning*: l'ontologia dovrà essere inoltrata al *reasoner* rispettando il vincolo dell'utilizzo delle API di OWL-ART.

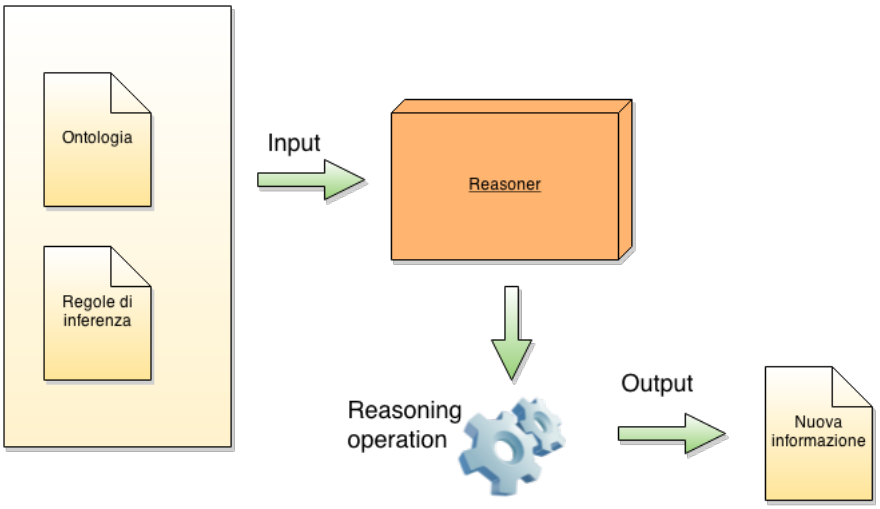


FIGURA 3.1.1. Operazione di *reasoning*

Durante il processo di *reasoning* il sistema deve essere in grado di memorizzare tutte le informazioni che hanno portato alla generazione della nuova informazione. Se ad esempio è stata scoperta un nuova tripla, il *reasoner* dovrà memorizzare tutti i passaggi che hanno portato alla sua scoperta. Più dettagliatamente, nello storico dell'informazione della tripla saranno presenti tutte le informazioni, contenute all'interno dell'ontologia, da cui è stata generata; ogni qual volta verrà eseguito il *reasoner*, lo storico di una precedente operazione di *reasoning* verrà sostituito con quello della nuova operazione e non sarà possibile recuperarlo. Ad operazione conclusa, l'utente sarà in grado di visualizzare lo storico con le informazioni prodotte durante il processo e se lo storico dovesse risultare vuoto significherà che nessuna nuova informazione è stata trovata.

Infine, il *reasoner*, è reso personalizzabile in base a dei parametri di configurazione che l'utente potrà modificare solamente prima dell'avvio dell'operazione di *reasoning*: al suo termine si potranno modificare nuovamente i parametri o lasciarli inalterati. Se l'utente non modifica alcun parametro, il reasoner utilizzerà dei parametri di *default* che verranno caricati prima dell'esecuzione dell'operazione. I parametri configurabili sono i seguenti:

- **Regole da applicare**: l'utente potrà scegliere quali regole di inferenza applicare durante le operazioni di *reasoning*. La scelta di queste regole dovrà essere coerente con quelle passate in input al *reasoner*, non sarà quindi possibile specificare regole al di fuori di quelle inserite nel sistema. Se l'utente non specifica alcuna regola allora verranno applicate tutte le regole specificate nel file delle regole di inferenza.
- **Produzione output**: l'utente può decidere se l'output del processo di *reasoning* dovrà essere prodotto oppure no. In caso negativo, alla fine del processo di *reasoning* non potrà visualizzare come le informazioni esplicitate sono state generate.
- **Numero di esecuzioni del *reasoner*** : l'utente potrà scegliere quante volte eseguire sulla stessa ontologia, con le stesse regole di inferenza e con gli stessi di parametri di configurazione l'operazione di *reasoning*. Ripetendo più volte l'operazione di *reasoning* sulla stessa ontologia è possibile trovare le nuove triple che derivano dalle informazioni che sono state aggiunte nelle precedenti iterazioni dell'operazione.

Per quanto riguarda i requisiti del dominio del sistema, abbiamo un unico punto che vincola il sistema ad utilizzare determinati tipi di dati per gestire le informazioni contenute nell'ontologia, dati specificati nelle OWL-ART-API. L'uso di questo vincolo è dovuto al fatto che, tramite le api messe a disposizione da OWL-ART, è possibile operare con diversi tipi di *ontology languages* senza dovere creare delle strutture *ad hoc* per ogni tipo di linguaggio utilizzato per le ontologie. Inoltre, tramite l'uso di queste API, l'inserimento del *reasoner* all'interno della piattaforma di *Semantic Turkey* non necessiterà di una conversione per le strutture che gestiscono l'interazione con l'ontologia: sia il *reasoner* si *Semantic Turekty* si basano su l'utilizzo delle OWL-ART-API. Infine, tramite l'uso di queste API, vengono messe a disposizione dello sviluppatore alcune funzionalità che rendono l'interazione con l'ontologia molto più semplice rispetto a dovere implementare *ex nihil* l'interfacciamento con l'ontologia.

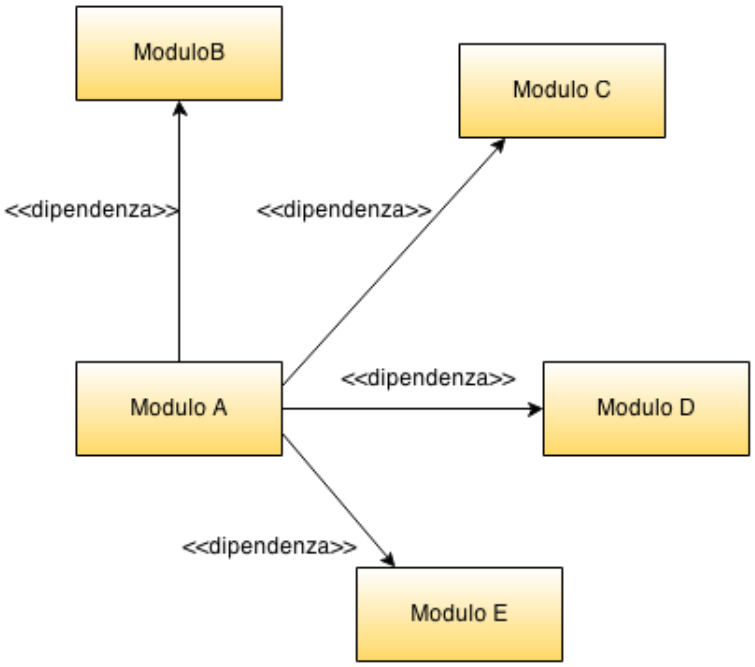


FIGURA 3.2.1. Dipendenza tra moduli

3.2. Un approccio modulare

Una volta che sono stati definiti i requisiti del *reasoner* è possibile iniziare a specificare l'architettura del sistema. Per la costruzione dell'architettura è stato scelto un approccio che ne garantisse la modularità dei vari componenti in modo tale da renderli il più possibile indipendenti l'uno dall'altro. Questo principio, riportato nella programmazione ad oggetti, corrisponde alla *loose coupling property* [**23**]. Sviluppare l'architettura di un sistema con un approccio *loosely coupled* permette di progettare dei componenti che hanno o che fanno uso di una parte o di nessuna informazione riguardo alla definizione degli altri componenti del sistema [**24**]. Sviluppare il sistema secondo questa linea guida permette di creare dei moduli dove ognuno di essi ha il compito di eseguire solamente un insieme di operazioni che appartengono allo stesso insieme di funzioni collegate logicamente: in questo modo, la modifica o il cambiamento di un modulo avrà un basso impatto sulle dipendenze del modulo stesso e sull'intero sistema. Riassumendo, questa scelta architetturale oltre a definire in ambienti separati le funzionalità che devono essere espresse dal sistema, permette un isolamento dei componenti che, se in un eventuale futuro dovessero essere cambiati o modificati, comporterebbe un spreco minimo di risorse.

Per capire meglio il funzionamento di questo approccio si pensi ad un modulo *A* che è dipendente dai moduli B, C, D e E, come descritto in figura 3.2.1 . Se il modulo A, oltre ad essere dipendente dai vari moduli, è dipendente dalla loro implementazione interna e non dalle interfacce che essi espongono, qualora si dovesse trovare nella situazione in cui i moduli B, C, D, E cambino la loro struttura interna o vengano totalmente sostituiti con altri componenti, A sarà costretto a rivedere l'utilizzo dell'uso delle sue dipendenze: operazione che dovrà essere fatta ogni qual volta i moduli da cui dipende cambiano. È importante, quindi, oltre a definire i moduli in modo indipendente, valutare con attenzione come vengono gestite le varie dipendenze. Se il modulo A, ha dipendenza dal modulo B, esso non deve dipendere dall'implementazione del modulo B, ma dall'interfaccia che B espone, in maniera tale che se in futuro il modulo B cambiasse implementazione interna, A non dovrà essere modificato. Di conseguenza, un ulteriore principio dello sviluppo dell'architettura consiste nel definire parallelamente alle definizione dei moduli le interfacce che essi espongono all'interno del sistema stesso. In figura 3.2.2 è mostrato un esempio di alta dipendenza tra i moduli di un sistema. Il modulo A che usa una classe interna del modulo B e del modulo C è strettamente dipendente dalla loro implementazione. Il metodo *OrderTotal()* è altamente dipendente dalle classi *CartEntry* e *CartContents*: se ad esempio volessimo inserire una logica che permettesse di calcolare gli sconti, dovremmo modificare tutte e tre le classi o se cambiassimo il tipo di lista della classe *CartContents* obbligati a modificare la classe *Order*. In figura 3.2.3 è illustrato come rendere le classi meno dipendenti. Come si può vedere, ogni modulo gestisce la propria logica riguardante

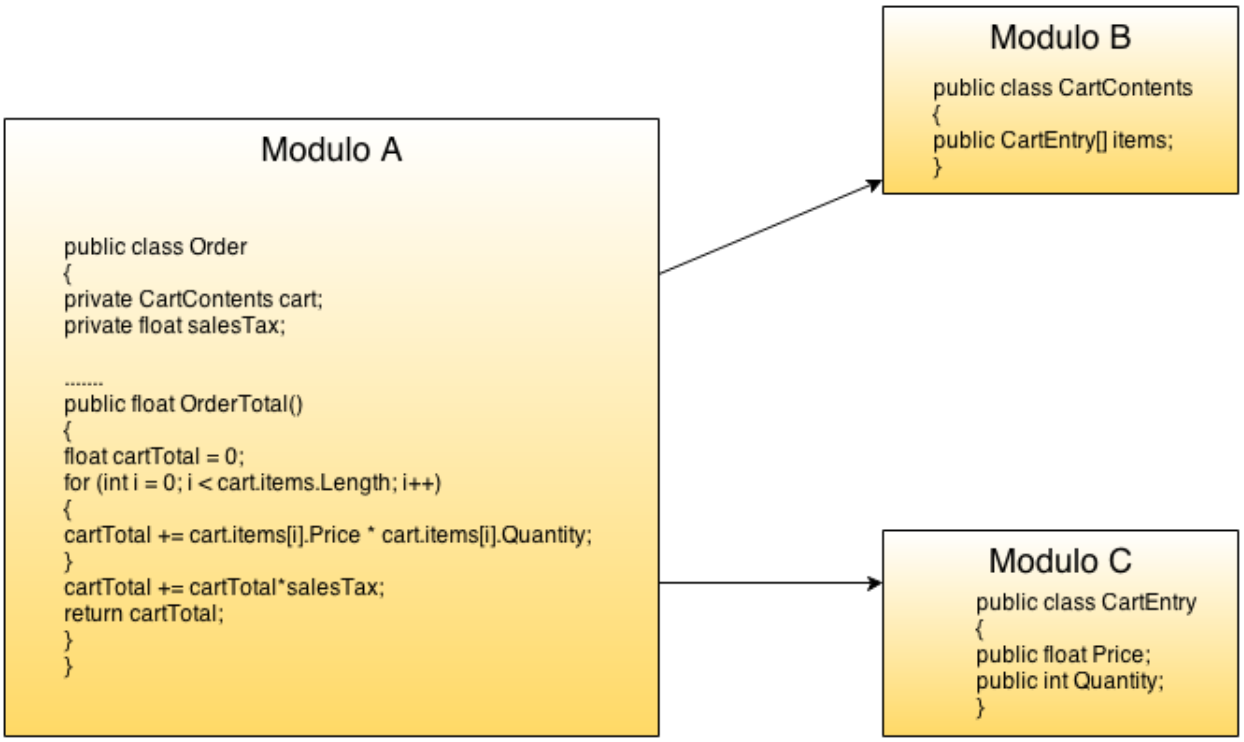


FIGURA 3.2.2. Alta copulazione tra moduli di un sistema

l'operazione sui propri dati e allo stesso tempo si occupa di gestire le proprie informazioni in maniera tale che il modulo A non dipenderà più dalla loro implementazione interna.

Per l'architettura del *reasoner* sono stati individuati i moduli rappresentati in figura 3.2.4 , dove per ogni modulo sono presenti delle frecce che indicano la dipendenza dagli altri moduli del sistema. I moduli presenti nello schema forniscono le seguenti funzionalità:

- ***Inference Rules Handler*** *:* Modulo per la gestione delle regole di inferenza. Questo modulo si occupa di effettuare le operazioni di lettura del *file* delle regole di inferenza specificate dall'utente e di conversione di queste regole in oggetti utilizzabili all'interno del dominio del sistema.
- ***Configurations Handler*** *:* Modulo per la gestione dei parametri di configurazione del sistema. Attraverso questo modulo verranno gestiti i parametri scelti dall'utente per effettuare le operazioni di *reasoning*.
- ***Execute Query Handler*** *:* Modulo per la gestione dell'esecuzione delle *query* per la ricerca di informazione all'interno dell'ontologia usata dal *reasoner*. In particolare, il modulo si occuperà di tradurre le regole di inferenza, che sono state precedentemente tradotte in oggetti appartenenti al dominio del sistema, in *query* eseguibili sull'ontologia.
- ***Output Handler***: Modulo per la gestione dell'output delle operazioni di *reasoning*. L'uso di questo modulo è attivato quando l'utente sceglie di produrre l'output del processo di *reasoning* memorizzando tutte le nuove informazioni.
- ***Reasoner Module***: Modulo principale del sistema. In questo modulo verrà eseguita l'operazione di reasoning.

Come si può vedere dalla figura 3.2.4, il modulo che gestisce l'operazione di *reasoning* è il componente che ha più dipendenze rispetto agli altri moduli del sistema. Questo scenario non deve indurre a pensare che il processo di scomposizione delle funzioni logiche sia sbagliato, poiché il modulo del *reasoning* è comunque quello che utilizzerà le funzioni messe a disposizione degli altri moduli per potere eseguire le proprie funzionalità. Sarebbe sbagliato, ad esempio, per diminuire la dipendenza, inserire dentro al

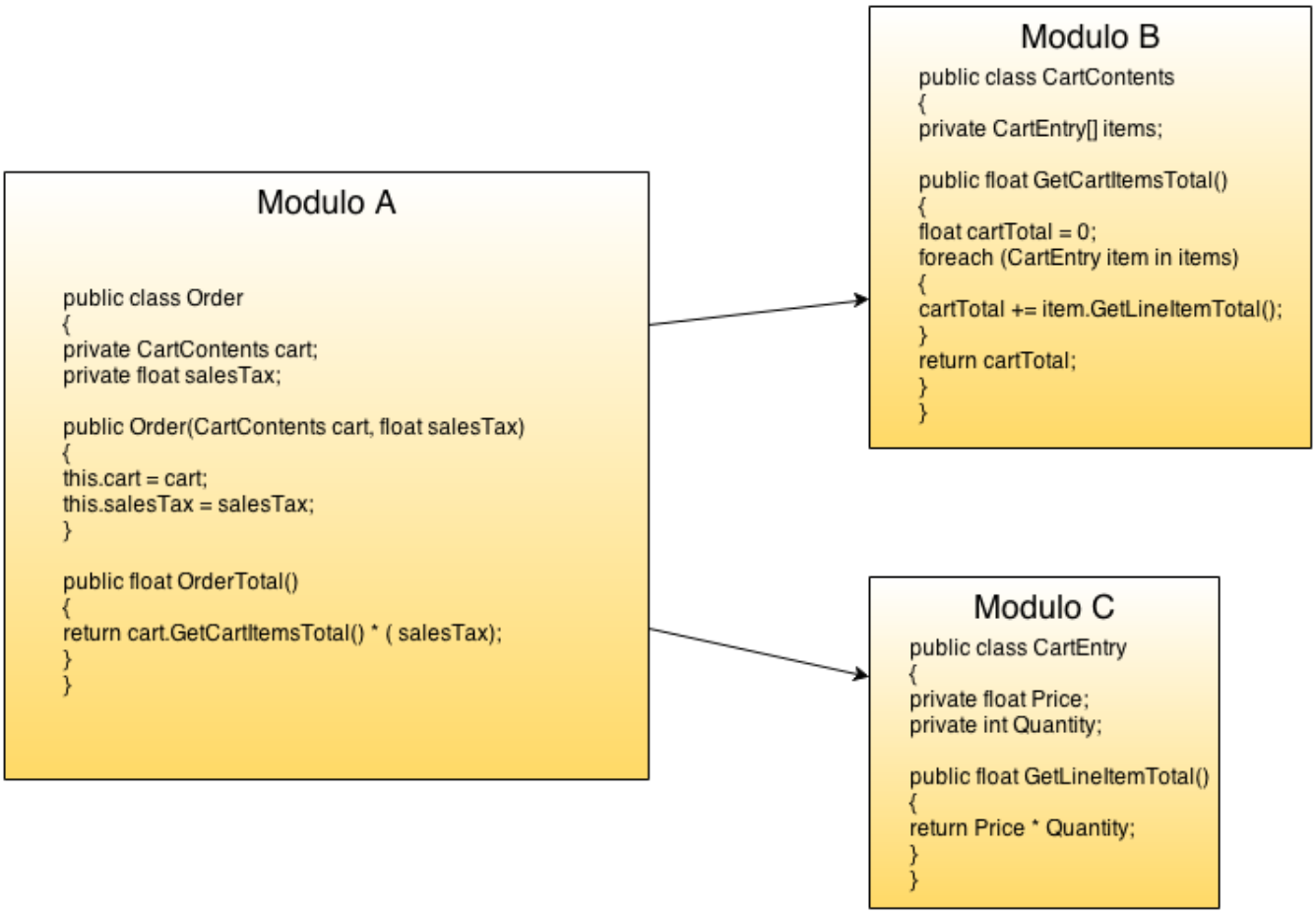


FIGURA 3.2.3. Sistema con bassa copulazione

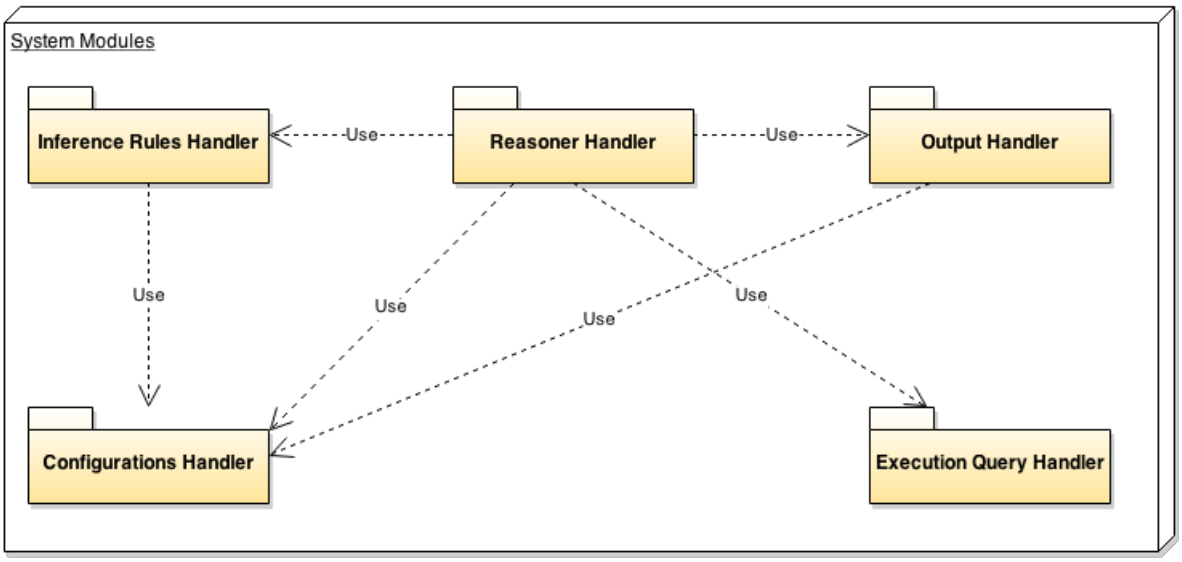


FIGURA 3.2.4. Moduli di sistema del *reasoner*

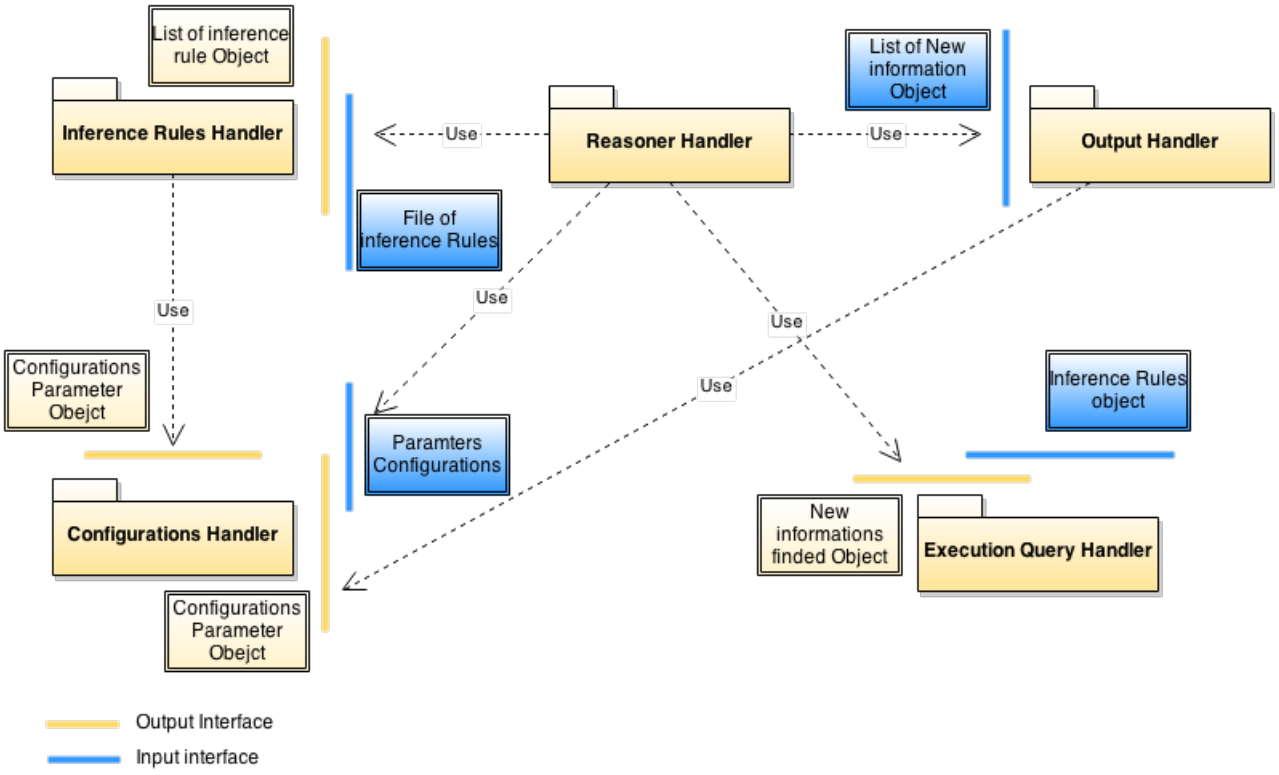


FIGURA 3.2.5. Interfacce esposte dai moduli del sistema

modulo del *reasoner* la funzione che gestisce la lettura delle regole di inferenza poichè, all'interno dello stesso modulo, non possono coesistere funzioni che non appartengono allo stesso insieme logico: il *reasoner* deve solamente eseguire l'operazione di *reasoning* e non si deve occupare di leggere le regole di inferenza.

Come abbiamo detto precedentemente, un'operazione che deve essere compiuta parallelamente alla definizione dei moduli del sistema è quella di specificare le interfacce con cui i moduli si presentano all'interno di esso. In questo modo, la modifica e la sostituzione di un modulo non comportano l'alterazione dei moduli che sono dipendenti da esso: l'importante è che i moduli oggetto della modifica o della sostituzione mantengano la stessa interfaccia che è stata definita nella costruzione dell'architettura. Nel caso della modifica dell'interfaccia si dovranno modificare anche i moduli che dipendono da essa.

Le interfacce dei moduli corrisponderanno agli oggetti che verranno restituiti o che devono essere inviati quando sarà usata una determinata dipendenza. In questo *modus operandi* i moduli che hanno una dipendenza si aspetteranno che un determinato oggetto venga loro restituito, se la dipendenza necessita di un oggetto in output, in input o entrambi. Questo sistema garantisce che i moduli non dipendano dall'implementazione di tali oggetti, ma solamente dall'oggetto stesso.

Per il nostro sistema sono state individuate le interfacce descritte in figura 3.2.5 (sono state rappresentate solamente le interfacce più significative). Nella figura, i moduli che restituiscono oggetti in output hanno un'interfaccia rappresentata da una linea colorata di giallo, mentre per i moduli che richiedono un oggetto in input, l'interfaccia ha una la linea colorata di blu. Ancora una volta, sottolineiamo l'importanza di avere creato l'architettura dei moduli del sistema seguendo due specifiche linee guida: la prima definisce un modulo come un insieme di funzioni correlate tra loro e la seconda ci obbliga a definire delle interfacce con cui i moduli devono interagire. Queste linee guida, oltre a rendere i moduli indipendenti dall'implementazione della conoscenza dei componenti da cui dipendono, isolano il modulo nel suo contesto operativo, cosicché i moduli possono essere sostituiti o modificati senza stravolgere l'architettura del sistema e possono essere utilizzati in altri sistemi, con il vincolo di costruire delle interfacce per interagire con tali moduli.

3.3. Diagrammi UML

Per definire meglio l'architettura del sistema è utile fornire alcuni diagrammi di interazione tra i moduli, presi in prestito dai metodi dell'ingegneria del software che facilitano una produzione di software di

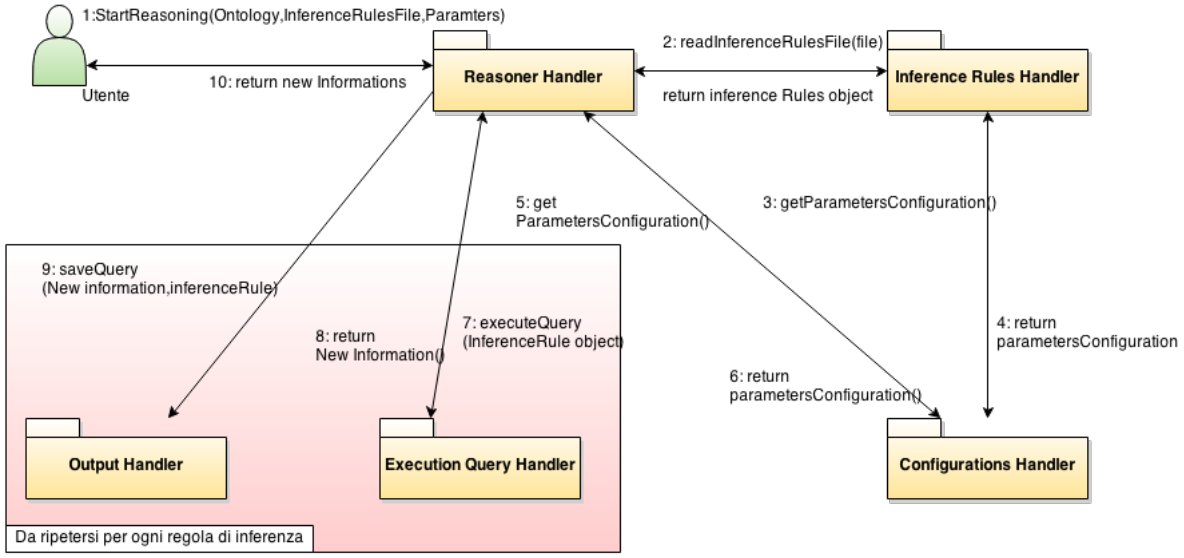


FIGURA 3.3.1. *Comunication Diagram*

alta qualità [**25**]. I metodi utilizzati per la creazione di questi digrammi affrontano i problemi attraverso un approccio orientato al paradigma *obejct oriented*(OO) utilizzando un linguaggio di modellazione che porta il nome di *Unified Model Language*(UML). La scrittura di questi diagrammi aiuta a definire, in fase implementativa del sistema, quali saranno le classi e metodi che si dovranno sviluppare per rispettare l'architettura del sistema. L'aspetto più importante risiede nella focalizzazione dell'interazione che avviene tra i vari moduli obbligando lo sviluppatore a pensare ad una struttura che interagisca rispettando le interfacce che sono state definite in fase architetturale. Il primo diagramma che verrà illustrato viene chiamato *Comunication Diagram* che presta l'attenzione sullo scambio di oggetti e le relazioni tra i partecipanti allo scambio [**26**]. In figura 3.3.1 è illustrato il *comunication diagram* relativo allo scambio di messaggi per l'esecuzione dell'operazione di *reasoning*. Vediamo nel dettaglio i messaggi che vengono scambiati tra i vari moduli del sistema:

(1) *StartReasoning(Ontology,InferenceRulesFile,Paramters)*: l'utente richiede al modulo del *reasoner handler* di iniziare le operazioni di *reasoning* con l'ontologia con il file delle regole di inferenza e i parametri che sono stati passati in input dall'utente.
(2) *readInferenceRulesFile(file)*: Il modulo del *reasoner*, una volata ricevuta la richiesta di *reasoning*, chiede al modulo che gestisce le regole di inferenza di leggere, dal file che è stato dato in input dall'utente, le regole di inferenza.
(3) *getParametersConfiguration()*: Il gestore delle regole di inferenza richiedere al gestore dei parametri di configurazione di avere i parametri specificati dall'utente per l'operazione di reasoning. Da questi parametri recupera quali regole di inferenza devono essere applicate, in modo tale da estrarle dal file delle regole di inferenza.
(4) *return parametersConfiguration()*: il gestore delle configurazioni restituisce l'oggetto contenente i parametri dell'operazione di *reasoning*.
(5) *return Inference Rules Object*: Una volta lette le configurazionoi, l'*Inference Rules Handler* legge le regole di inferenza e le trasforma in oggetti utilizzabili all'interno del sistema, successivamente passa tali oggetti al modulo del *reasoner*.
(6) *getParametersConfiguration()*: Il *reasoner Handler* chiede al gestore dei parametri di configurazione di avere i parametri specificati dall'utente per l'operazione di *reasoning*. Da questi parametri recupera il dato che specifica quante volte eseguire l'operazione di *reasoning* e se deve produrre l'output.
(7) *return parametersConfiguration()*: il gestore delle configurazioni restituisce l'oggetto contenente i parametri dell'operazione di *reasoning*.
   (a) Per ogni regola di inferenza invia i seguenti messaggi:
(8) *executeQuery(InferenceRule object)*: Il *reasoner Handler* invia un regola di inferenza, da applicare sull'ontologia, al modulo che si occupa di eseguire le *query* di ricerca.
(9) *return NewInformation*: L*'Execute Query Hanlder* esegue la regola di inferenza che ha ricevuto

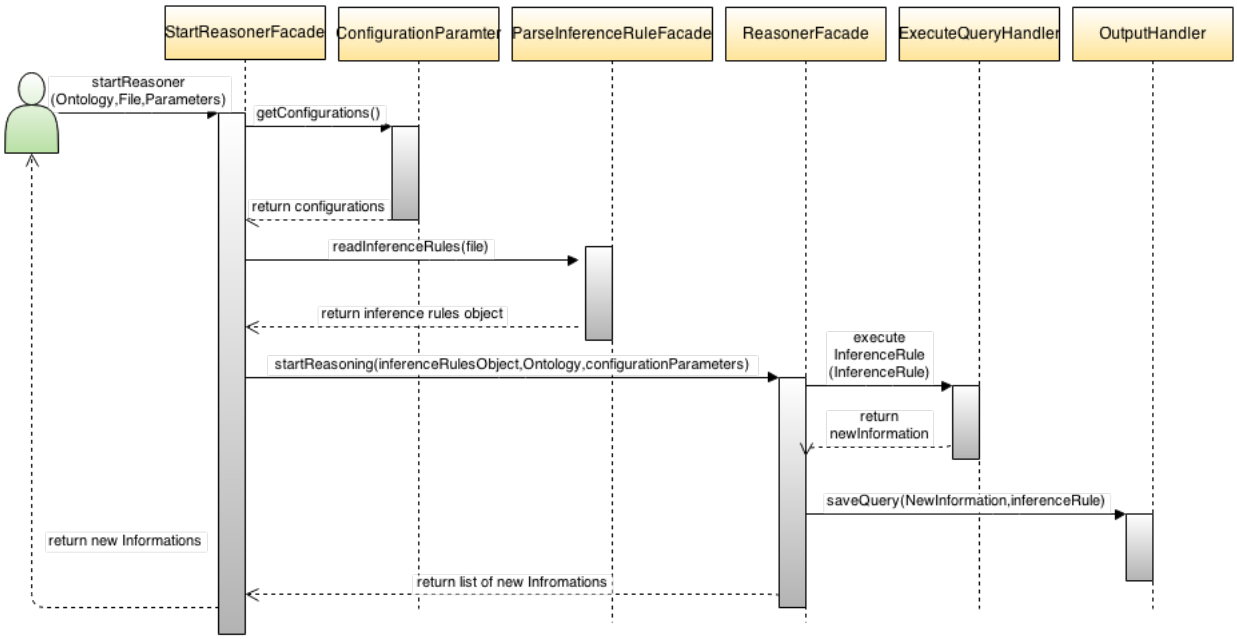


FIGURA 3.3.2. *Sequence Diagram*

in input e restituisce il risultato al gestore dell'operazione *reasoning.*

(10) *saveQuery(NewInformation,InferenceRule)*: il *reasoner*, se è stata abilitata l'opzione di produrre l'output, invia la nuova informazione che è stata trovata applicando la regola di inferenza sull'ontologia, al gestore dell'output corredata dalla regola di inferenza che l'ha generata.
   (a) Fine ciclo esecuzione regole di inferenza

(11) *return New Informations*: il *reasoner Handler* ritorna tutte le nuove informazioni trovate all'utente che aveva richiesto di eseguire l'operazione di *reasoning.*

Un'altra classe di diagrammi utili per la definizione dell'architettura del sistema e per porre le basi dell'implementazione vera e propria è la classe dei diagrammi di sequenza o *Sequence diagram.* I *sequence diagram* focalizzano l'attenzione sull'interazione degli oggetti (moduli) che collaborano per realizzare un fine comune[**27**] che nel nostro caso coincide con l'operazione di *reasoning.* Le interazioni che avvengono in questi tipi di diagrammi servono per ragionare sul sistema che si sta progettando e per identificare le classi e i metodi che ne costruiranno la soluzione. La figura 3.3.2 mostra il *sequence diagram* relativo all'esecuzione dell'operazione di *reasoning.*

Come si può vedere dalla figura, sono state create delle classi che marcano un altro aspetto fondamentale del processo implementativo del sistema ossia la distinzione netta tra le classi che eseguono la logica del processo delle operazioni del sistema e quelle che eseguono materialmente le operazioni. Per capire meglio questo concetto si pensi all'analogia con un concerto di musica classica. L'obiettivo finale dell'orchestra è quello di suonare un determinato brano musicale, obiettivo che è portato a termine da un insieme di esecutori ai quali è stata assegnata una singola parte. Invece di suonare secondo le proprie regole, tutti gli esecutori sono vincolati ad eseguire le proprie operazioni seguendo le istruzioni del direttore d'orchestra, il quale fornisce il momento in cui tali esecuzioni devono essere eseguite: un esecutore non partirà ad eseguire la propria parte finché non avrà ricevuto "l'ordine" dal direttore d'orchestra. Allo stesso modo funzionerà il nostro sistema. Ogni modulo (esecutore) ha un proprio compito da svolgere (suonare la parte), invece di scegliere per proprio conto come dirigere le operazioni per eseguire il proprio *task,* aspetterà che un componente esterno (il direttore) creato *ad hoc*, gli comunichi quando eseguire il suo compito. A differenza dell'orchestra, il sistema ha più direttori, i quali risponderanno successivamente ad un direttore di più alto livello (creando una gerarchia di direttori) che coordinerà l'intera gestione del sistema. In particolare, saranno presenti i componenti per la gestione della logica di lettura delle regole di inferenza, per la gestione del processo di *reasoning* ed infine un controllore generale che gestisce le operazioni minori e i controllori più piccoli come quelli che abbiamo appena elencato.

Quali sono i vantaggi di questa scelta implementativa? Il motivo che ha portato a questa preferenza architetturale è il seguente: inserire la logica operativa delle funzionalità che il sistema deve esprimere

dentro ai moduli che si occupano di eseguire le operazioni materiali del sistema comporta una serie di svantaggi. Il primo di essi consiste nell'accorpare la logica funzionale con la logica implementativa: come è stata definita una separazione dei moduli in funzionalità correlate logicamente, allo stesso modo, ad un livello più basso, deve essere definita una distinzione tra le classi che eseguono le operazioni e le classi che coordinano la logica delle operazioni. Un secondo svantaggio è quello di cablare la logica delle operazioni all'interno delle classi costringendo a riscrivere tutta la logica dei processi qualora una classe venga eliminata per essere sostituita da una nuova. In sostanza, bisogna adottare i principi di sviluppo che sono stati definiti per i moduli del sistema portandoli al livello implementativo delle classi. Inserire la logica delle operazioni in classi separate, corrisponde a creare delle "facciate" all'interno dell'insieme delle classi, in questo modo se una classe che esegue le operazioni cambia implementazione, viene sostituita o modificata, non ci sarà bisogno di cambiare la classe che si occupa di gestire l'esecuzione delle operazioni. Analogamente, se viene cambiata la logica del processo delle operazioni è necessario modificare solamente le classi facciate del sistema, senza intaccare gli esecutori delle operazioni. In conclusione, questo approccio cerca di isolare il più possibile i componenti interni del sistema, slegandoli dal flusso di esecuzione delle operazioni di *reasoning*, ossia eliminando la logica del processo dai singoli moduli per trasferirla nei moduli il cui unico compito è quello di attuare tale logica.

Nel *sequence diagram* in figura 3.3.2 sono evidenziate le classi principali che si occupano di gestire l'esecuzione della logica delle operazioni per portare a compimento l'operazione di *reasoning*. Le classi "facciata" sono le seguenti:

- **StarReasonerFacade**: questa classe è il gestore di tutte le operazioni del sistema. Tramite essa vengono controllate le altre classi "facciate" e gestite le operazioni di modifica dei parametri di configurazione, inserimento dell'ontologia e visualizzazione dell'output (se è stato prodotto).
- **ParseInferenceRuleFacade**: questo controllore si occupa di gestire tutte le operazioni per la lettura del file delle regole di inferenza, della loro validazione e della trasformazione in oggetti del dominio di sistema
- **ReasonerFacade**: si occupa di gestire l'operazione di *reasoning* vera e propria. Gestisce l'esecuzione delle regole di inferenza delegando i compiti ai moduli dell'esecutore delle regole di inferenza e , per il loro salvataggio, utilizza il modulo per la gestione dell'input.

Ancora una volta è utile evidenziare come queste classi hanno la funzione di delegare a quelle sottostanti i vari compiti che devono essere eseguiti per portare a termine una determinata operazione.

CAPITOLO 4

Implementazione del sistema

In questo capitolo verranno descritte le varie implementazioni delle funzionalità espresse dal sistema all'interno dei moduli dell'architettura. Il linguaggio di programmazione utilizzato appartiene alla famiglia dei linguaggi basati sul paradigma OO (*Objected Oriented*) e corrisponde a *Java v1.6* [**28**], scelta dettata dalla necessità di utilizzare un linguaggio di programmazione che implementasse il concetto di scambi di messaggi e dell'interazione tra oggetti all'interno di un sistema in un ambiente ben definito.

4.1. Dominio del sistema

Prima di passare ad analizzare le varie implementazioni delle operazioni è necessario definire l'implementazione degli oggetti del dominio sui quali il sistema lavora. Gli oggetti atomici sul quale verranno basate le regole di inferenza sono definiti con il nome di "tripla" e non sono nient'altro che una rappresentazione della struttura *subject-predicate-object* dell'RDF descritta nella sezione 2.1.1 . La sua implementazione è la seguente:

```
....
public class Triple{

....

private String subject;
private String predicate;
private String object;
....
```

In accordo alla definizione di regole di inferenza che abbiamo dato nella sezione 2.1.1 la loro implementazione consta di una lista di premesse seguite da una lista di conclusioni che sono rappresentate nel seguente modo:

```
public class InferenceRule{

private int inferenceRuleID;
private String inferenceRuleName;

private List<Triple> premisis;

private List<Triple> conclusions;
```

4.2. Implementazione del gestore delle configurazioni

Una delle funzionalità offerte dal sistema consiste nel dare la possibilità all'utente di scegliere i parametri di configurazione per l'operazione di *reasoning*. Nello specifico i parametri sono:

- Specifica delle regole di inferenza che devono essere applicate
- Produzione dell'output
- Quante volte deve essere eseguita l'operazione di *reasoning* sull'ontologia.

Come è stato espresso nei requisiti di sistema, l'utente può modificare i parametri di configurazione solamente prima dell'avvio dell'operazione di *reasoning*: infatti non è concessa la modifica dei parametri mentre l'operazione è in esecuzione. La gestione dei parametri è stata implementata tramite il meccanismo messo a disposizione dalle API di *Java* per la scrittura e lettura dei file di *properties* [**29**]. Questi particolari file permettono di memorizzare al loro interno dei record di tipo *chiave/valore* che possono essere letti e

scritti in maniera semplice all'interno delle classi *Java*. Il file di configurazione *config.properties* relativo alla gestione del nostro sistema conterrà le seguenti coppie di chiavi/valore (i valori inseriti forniscono un un esempio di valori):

```
ouput.produce = true
number.of.cycle.resoning = 2
which.rule.execute = 1,2
```

Tali valori possono essere letti e scritti usando i metodi messi a disposizione della classe che si occupa di gestire la configurazione:

```
public class ConfigurationParameter {
....
private ConfigurationParameter(){
    //Create new Properties Object
   prop = new Properties();
   //Load properties
     prop.load(
     getClass().getClassLoader().getResourceAsStream(CONFIGURATION_FILE_NAME));
    ...
}

public static ConfigurationParameter getIstance() {
      if(istance == null){
         istance = new ConfigurationParameter();
      }
      return istance;
}

public String getProperty(String propertyName){
      return prop.getProperty(propertyName);
   }

public void modifyProperties(String key,String value) {
 //Get properties file
 File f=new File(getClass().getClassLoader().
   getResource(CONFIGURATION_FILE_NAME).getFile());
 //Creat a file stream
 FileOutputStream out = new FileOutputStream(f);
 //Update the property value
 prop.setProperty(key, value);
 //save update
 prop.store(out, null);
 //close file stream
 out.close();
}
```

Nel dettaglio, il metodo *getProperty* recupera il valore di un parametro mentre il *modifyProperties* ne modifica il valore. Come si può vedere dalla definizione della classe, essa usa un *pattern singleton*[**30**] per essere *istanziata*: scelta motivata dal fatto che i parametri del sistema sono uguali per tutto l'arco dell'esecuzione dell'operazione di *reasoning* in quanto creare istanze diverse della classe risulterebbe inutile e aumenterebbe l'uso della memoria.

4.3. Implementazione della gestione delle regole di inferenza

La definizione di una struttura per la scrittura delle regole di inferenza ci aiuta ad aumentare la leggibilità delle regole stesse e, in fase di esecuzione delle regole all'interno dell'ontologia, ad ottenere una facile conversione con gli oggetti del dominio del sistema: l'obiettivo che vogliamo ottenere è trasformare gli elementi appartenenti alle regole di inferenza, scritti in forma testuale, in oggetti del dominio di tipo *InferenceRule*. Vediamo come è possibile realizzare questa conversione.

4.3.1. XML. Una prima implementazione potrebbe essere ottenuta attraverso l'uso dell' *eXtendible Marckup Language*(XML) con il quale possiamo definire una struttura simile alla definizione formale delle regole di inferenza con l'aggiunta di ulteriori informazioni utili al processo di *reasoning* (che non verranno mostrate poiché hanno poca importanza nel nostro contesto). Come abbiamo detto nella sezione 2.1.1 le regole di inferenza sono rappresentate da un insieme di premesse e di conclusioni, dove ogni premessa

e conclusione ha al suo interno la struttura base *subject-predicate-object,* traslando questa struttura in XML otteniamo il seguente risultato:

```
.....
<inference-rule> //start tag nuova regola
        <premisis> // definizione delle premesse
                ....
                <premise> // definizione di una premessa
                        <subject>...</shubject>
                        <predicate>...</predicate>
                        <object>...</object>
                </premise>
                ....
        </premisis>
        <conclusions> // definizione delle conclusioni
                ....
                <conclusion> // definizione di una conclusione
                        <subject>...</shubject>
                        <predicate>...</predicate>
                        <object>...</object>
                </conclusion>
                .....
        </conclusion>
</inference-rule> // fine della regola
```

Come si può vedere è stata mantenuta fedelmente la struttura delle regole di inferenza, per ogni regola che si vuole inserire basterà aggiungere nuovamente l'intera struttura definita sopra.

Il passaggio successivo da parte del sistema consiste nel processare il documento contenente le regole espresse in XML. Esso si dovrà occupare di tradurre le regole di inferenza in oggetti del dominio del sistema, ossia in oggetti *Java*: la libreria *JAXB*[**11**], messa a disposizione dal pacchetto *Java Web Services Development Pack* (JWSDP), permette di effettuare l'operazione di *binding* tra un *xml* e una classe *Java.* Utilizzando delle apposite *annotation* posizionate all'interno della classe *Java* che definisce le regole di inferenza è possibile creare automaticamente la traduzione da un *file xml* ad un oggetto *Java.* La classe *InferenceRule* verrà modificata inserendo le seguenti *annotation* [1]:

```
....

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "inference-rule")
public class InferenceRule{

....

private List<Triple> premisis;

private List<Conclusion> conclusions;
```

Una volta che è stato eseguito il *binding* tra l'*XML* contenente la definizione delle regole di inferenza e le classi *Java* che le rappresentano, occorrerà validare le regole di inferenza per poterle usare all'interno del *reasoner*: ad esempio dentro il *subject* di una premessa o conclusione non potrà essere presente una stringa di soli caratteri numerici, pena l'annullamento della regola di inferenza. Il processo di validazione verrà considerato nelle successive sezioni, per ora ci limitiamo solamente a definire un'implementazione del processo di acquisizione delle regole di inferenza all'interno del sistema.

4.3.2. ANTRL. Un'ulteriore implementazione del processo di acquisizione delle regole di inferenza può essere ottenuta attraverso la costruzione di un *parser* ossia la creazione di un programma che sia in grado di riconoscere un certo tipo di *linguaggio.* L'idea generale è quella di scrivere una grammatica

[1] Va osservato che l'oggetto di dominio deve essere indipendente dalla tecnologia usata per rappresentare le regole di inferenza.

formale che definisca un insieme di regole sintattiche per la generazione di determinate *stringhe* rappresentati le regole di inferenza. Questa grammatica verrà poi utilizzata dal *parser* al fine di effettuare il controllo sulle regole di inferenza ricevute in input e verificare che la loro struttura grammaticale corrisponda alle regole contenute nella grammatica formale: al termine di questa operazione, se andata a buon fine, verrà effettuata una corrispondenza tra gli elementi riconosciuti all'interno della regola di inferenza e gli oggetti del dominio.

Sostanzialmente l'operazione di *parsing* consiste in una prima fase denominata analisi lessicale[**49**], in cui l'input (le regole di inferenza) viene suddiviso, in base a degli opportuni pattern, in uno o più *token* ossia *sottostringhe* di caratteri consecutivi che formano un'unità logica[**48**]. La lista dei *token* che viene generata dall'analisi lessicale viene poi usata nella seconda fase dell'operazione di *parsing* che si occupa di svolgere l'analisi sintattica: durante questa fase si controllerà che i *token* formino espressioni corrispondenti a quelle definite all'interno della grammatica e tale processo verrà rappresentato attraverso la creazione di un albero di *parsing*. La terza ed ultima fase, corrisponde all'analisi semantica: verrà controllato che i valori dei *token* situati all'interno dell'albero siano consistenti con i valori definiti all'interno della grammatica (per una spiegazione esaustiva della creazione di grammatica formale e di *parser* associato fare riferimento qui[**12**] e qui[**13**]).

Gli strumenti che servono per definire una grammatica formale e un *parser* vengono messi a disposizione da un *tool* che porta il nome di ANTLR: *Another Tool for Language Recognition.* ANTRL oltre a fornire questa serie di strumenti, mette a disposizione dello sviluppatore degli oggetti chiamati "*tree walker*" con i quali è possibile effettuare delle visite degli alberi di *parsing* ed eseguire del codice specifico ad ogni visita di un nodo. Un esempio di una grammatica con la quale è possibile definire il linguaggio che genera le regole di inferenza è il seguente:

inferenceRule → **newRule premisis conclusions**
newRule →′ *nuova regola*′
premisis → **premisis** | **premise**
conclusions → **conclusions** | **conclusion**
premise → **startPremise triple**
startPremise →′ *premessa*′
triple →′ *subject*′ **value**′*predicate*′ **value**′*object*′ **value**
value → **value** | *a*.....*z* | 0....9
conclusion → **startConclusion triple**
startConclusion →′ *conclusione*′

Le regole di inferenza che possono essere generate dalla grammatica descritta sopra sono:

nuova regola
premessa subject Pluto predicate Amico object Pippo
conclusione subject Pippo predicate Amico object Pluto

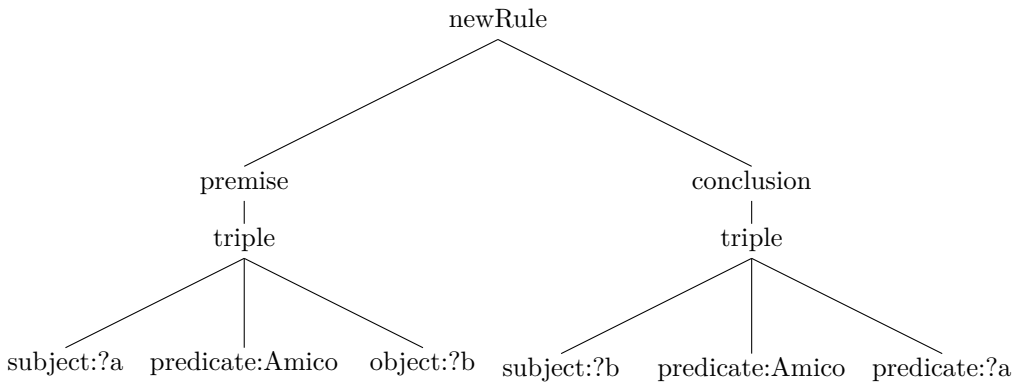
La grammatica che è stata appena definita, per essere utilizza in ANTRL, deve essere modificata in accordo alle linee guida del *tool*. Di seguito viene riportata la grammatica (con alcune aggiunte, ma il cuore della grammatica rimane lo stesso) utilizzata all'interno del sistema che rispetta i vincoli imposti da ANTRL.

```
grammar InferenceRulesGrammar;
options {
// Otuput Language
language = Java;
}
//Entry parser rule. A rule must have an information section (about rule) almost one premisie and almost one conclusion
parseInferenceRule: (new_rule)+;
new_rule : NEW_RULE (rule_information) ((premise))+ ((conclusion))+;
//New rule token
NEW_RULE : 'new rule :' | 'new rule:' ;
//Rule information. Each rule must have a name and a id identfication
rule_information : RULE_NAME LetterAndSymbol RULE_ID NUMBER ;
//Information rule tokens
RULE_NAME : 'name : ' |'name: '; RULE_ID : 'id: ' | 'id : ';
//Premise parser rule. Eache premise must have a triple.
premise : (START_PREMISE triple);
//Start premise token
START_PREMISE : 'premise : ' | 'premise: ';
//Conclusion parser rule. Each concluisone must have a triple.
conclusion : (START_CONCLUSION triple );
/Start conclcusion token
START_CONCLUSION : 'conclusion: ' | 'conclusion : ';
//Triple parse rule. Each triple is composed by subject,object and predicate with relative values
triple: ('subject: '|'subject : ')(value)( 'predicate : ' |'predicate: ')(value)('object: '|'object : ')(value);
.....
WS : [ \t\n]+ -> skip ; // skip spaces, tabs, newlines
```

Una volta definita la grammatica, ANTRL genererà le corrispondenti classi *Java* per la creazione del *parser* e per la relativa operazione di *parsing* delle regole di inferenza. Successivamente, dato in input il file con le regole di inferenza, si potrà avviare il *parsing* attraverso il quale verrà verificato che le regole corrispondano alle stringhe del linguaggio generate dalla grammatica. Un esempio di stringa generata dal linguaggio espresso dalla grammatica scritta per ANTRL è la seguente:

new rule :
name: newRule
id: 1
premise: subject: ?a predicate: Amico object: ?b
conclusion: subject: ?b predicate: Amico object: ?a

dalla quale ANTRL genererà il seguente albero (semplificato) di *parsing*:



L'albero di *parsing,* come detto precedentemente, si potrà visitare con l'ausilio del *tree walker* in modo

tale da effettuare un *mapping* tra le regole di inferenza all'interno dell'albero e gli oggetti del dominio rappresentati medianti classi *Java*.

4.3.3. XML vs ANTRL. Ora che sono state illustrate le due implementazione per la gestione delle regole di inferenza, verranno analizzati i pro e i contro delle due tecnologie.

XML:

- PRO: facilità per lo sviluppatore nell'effettuare il *mapping* tra il file XML contenente le regole di inferenza e gli oggetti di dominio del sistema.
- PRO: *low coupling*. Se bisogna cambiare le regole di inferenza, lo sviluppatore dovrà modificare solamente le *annotations*(e scrivere i corrispondenti nuovi *tag* nel file XML).
- CONTRO: poca "leggibilità" delle regole di inferenza. Avendo una struttura imposta dallo standard XML, la leggibilità delle regole di inferenza, da parte dell'utente, potrebbe risultare difficile.
- CONTRO: l'utente può trovarsi in difficoltà nella stesura delle regole di inferenza: poiché devono essere scritte seguendo gli *standard* XML, se le regole di inferenza sono complesse, potrebbe risultare difficile scriverle.
- CONTRO: necessità, da parte dello sviluppatore, di creare un meccanismo per la validazione delle regole di inferenza. L'utente potrebbe immettere valori come "1111111" all'interno di una regola.

ANTRL:

- PRO: regole di inferenza *human readable*. Le regole di inferenza vengono espresse in costrutti semplici e quindi di facile lettura da parte dell'utente.
- PRO: facilità di scrittura. Le regole vengono scritte in un linguaggio quasi naturale rendendone la stesura molto semplice da parte dell'utente.
- PRO: lo sviluppatore non è obbligato ad effettuare il controllo del contenuto delle regole di inferenza. Realizzando una grammatica si possono ottenere delle regole che vincolano l'utente a scrivere solamente determinate tipologie di stringhe all'interno delle regole.
- CONTRO: lo sviluppatore potrebbe avere difficoltà nello scrivere la grammatica per le regole di inferenza. Una grammatica inesatta porta a generare regole di inferenza inesatte.
- CONTRO: alta copulazione. Se in un futuro cambiasse la struttura delle regole di inferenza, lo sviluppatore si trova costretto a dovere cambiare la grammatica, il *parser* e la lettura del l'albero di *parsing* generato da *ANTRL*.

Per lo sviluppo del *reasoner* si è scelto di adottare come implementazione delle regole di inferenza la tecnologia di ANTLR. Nonostante questa tecnologia soffra del problema dell'alta copulazione, che in un approccio nei sistemi modulari è altamente sconsigliato, permette di esprime le regole di inferenza in un linguaggio quasi naturale e quindi comprensibile all'utente che lo scrive. Questa qualità per le finalità del progetto è molto importante: per l'utente che utilizza il *reasoner* assicura una maggiore comprensione del processo di inferenza applicato sull'ontologia e per questo motivo la scelta di utilizzare ANTRL è sembrata la via migliore.

Deve essere sottolineato che, qualora si volesse cambiare approccio per l'implementazione delle regole di inferenza, l'uso di un sistema modulare concede allo sviluppatore la totale libertà d'azione nel modificare i componenti purché l'obiettivo finale sia sempre quello di restituire una lista di regole di inferenza espresse negli oggetti di dominio del sistema (questa scelta è obbligata dall'architettura del sistema).

4.3.4. Classi per la gestione delle regole di inferenza. La gestione delle regole può essere divisa nei seguenti *task:*

- Acquisizione del file
- Lettura del file
- Conversione degli elementi letti precedentemente in oggetti del dominio del sistema

Lo schema in figura 4.3.1mostra l'insieme delle classi fondamentali per l'esecuzione dell'operazione di lettura del file delle regole di inferenza. Come si può vedere in figura è presente una classe che si occupa di gestire l'operazione logica di lettura delle regole, delegando alle altre classi il compito di svolgere le esecuzione materiale dell'operazione. Nello specifico, una volta ricevuto in input il file delle regole di inferenza, la classe *ParseInferenceRulesFacade* gestirà le operazioni come descritto nel seguente modo (listato 4.1): verrà inizializzato un nuovo oggetto di tipo *Parser* e successivamente verrà eseguito il metodo *parsingInferenceRulesFile* che legge le regole di inferenza dal file ricevuto in input. La lettura, invece, avviene secondo le modalità descritte nell'analisi dell'utilizzo della libreria di ANTRL. Una volta

LISTING 4.1. classe *ParseInferenceRulesFacade*

```
public InferenceRules doHandler(File inferenceRulesFile) {
  List<InferenceRule> inferenceRules = new ArrayList<InferenceRule>();
 //Create new Parser
 parser = new Parser();
 //Parsing inference rules file
 inferenceRules = parser.parsingInferenceRulesFile(inferenceRulesFile);
 Validator validator = new Validator(configurationParameter);
 //Validate inferenceRules
 boolean passValidation = validator.validate(inferenceRules);
 ....
 return inferenceRulesObject;
}
```

restituiti gli oggetti del dominio essi vengono validati usando l'uso del *validatore*, il quale, in base alle regole di inferenza che ha scelto l'utente, le scremerà da quelle ottenute dalla lettura del file. Se la validazione va a buon fine verrà restituito un oggetto contenente le regole di inferenza, altrimenti verrà inviato un errore.

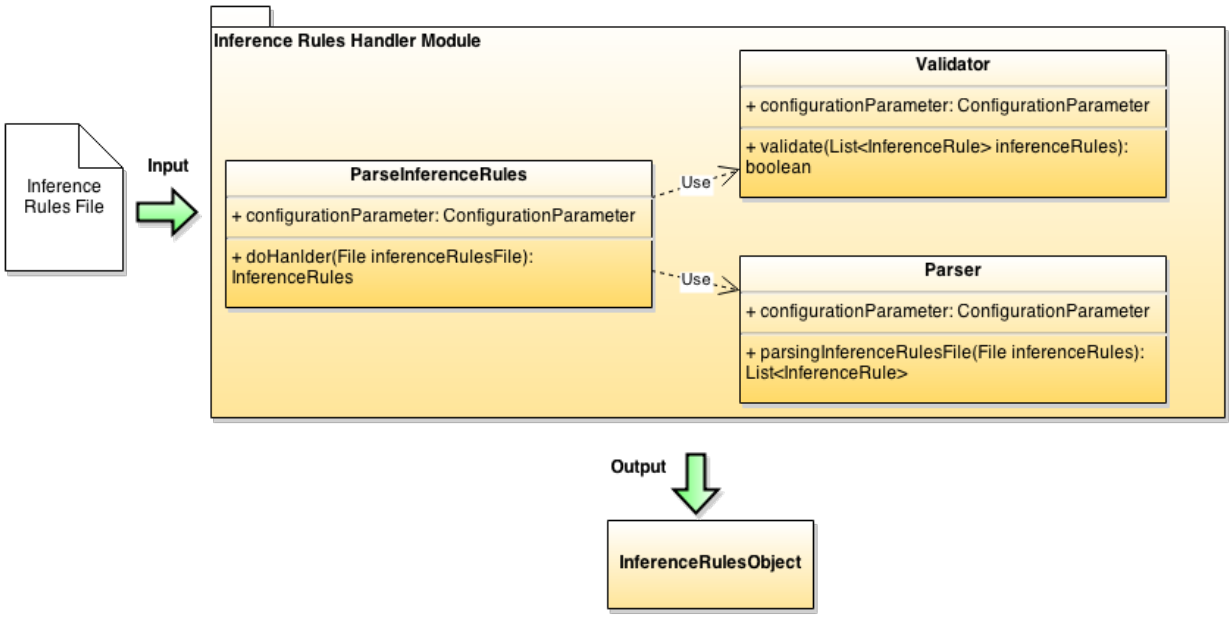


FIGURA 4.3.1. *Inference Rules Handler module*

4.4. Implementazione dell'esecuzione delle regole di inferenza

Il *reasoner*, una volta acquisite le regole di inferenza, dovrà applicarle sull'ontologia per potere inferire nuova conoscenza. Le API di OWL-ART mettono a disposizione dello sviluppatore una rappresentazione delle informazioni contenute all'interno dell'ontologia attraverso l'utilizzo di una base dati chiamata *RDF Store* e un meccanismo di interrogazione al pari di comune basi dati. Per accedere alle informazioni contenute nell'RDF *Store* è necessario tradurre le premesse e le conclusioni, contenute all'interno di una regole di inferenza, in *query* da eseguire sulla base dati. La traduzione avverrà attraverso un modulo sviluppato *ad hoc* (rispettando il principio di bassa copulazione) che, ricevendo in input le premesse e le conclusioni di una regola di inferenza e una volta trasformate in *query* eseguibili, restituirà sotto forma di un oggetto di domino, il risultato delle *query* applicate. Il linguaggio utilizzato per effettuare le interrogazioni sull'ontologia porta il nome di SPARQL[**15**] che ha a disposizione varie opzioni per eseguire una ricerca mirata in base alle esigenze del sistema: una di queste è lo SPARQL CONSTRUCT. Vediamo

il suo funzionamento. Supponiamo di avere una regola di inferenza che definisce il concetto di transitività:

```
new rule :
name: RegolaTransitiva
id:1
premise: subject:?p predicate: type object: transitiveProperty
premise: subject: ?a predicate: ?p object: ?b
premise: subject: ?b predicate: ?p object: ?c
conclusion: subject: ?a predicate: ?p object: ?c
```

nelle premesse esprimiamo di cercare tutte le entità che hanno una proprietà di tipo transitivo e tutte le entità che usano quella proprietà con altre entità della relazione. Se, eseguendo la regola di inferenza, vengono trovate delle entità che rispecchiano i vincoli espressi nelle premesse, allora possiamo creare una nuova informazione seguendo le specifiche contenute nella conclusione. La regola di inferenza appena descritta può essere espressa interamente nel costrutto CONSTRUCT:

```
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT {
?b ?a ?d
}
WHERE {
    ?a rdf:type owl:TransitiveProperty .
    ?b ?a ?c .
    ?c ?a ?d
}
```

Il CONSTRUCT permette di eseguire la regola di inferenza in un "colpo solo". L'unica difficoltà, usando questo metodo, rimane solamente la traduzione della regola di inferenza nella sintassi del COSTRUCT, ossia trasformare ogni singola premessa e conclusione in variabili conformi al linguaggio SPARQL (questa trasformazione è eseguita in un modulo a parte). La facilità di utilizzo del CONSTRUCT sembrerebbe portare ad adottare questa scelta come scelta definitiva per l'esecuzione delle regole di inferenza, ma non sarà così.

L'uso del CONSTRUCT nasconde un insidia: lede uno degli obiettivi espressi nel paragrafo 1.2 per cui viene sviluppato un nuovo *reasoner*. Utilizzando il CONSTRUCT perdiamo tutti i passaggi che portano alla generazione della nuova informazione poiché il risultato finale della *query* restituirà solamente i valori delle variabili espresse nel clausola del *construct* della *query*. Una soluzione a questo problema potrebbe essere quella di aggiungere all'interno della clausola del CONSTRUCT anche le premesse:

```
PREFIX owl:<http://www.w3.org/2002/07/owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT {
        ?b ?a ?d.
        ?b ?a ?c .
        ?c ?a ?d
}
WHERE {
    ?a rdf:type owl:TransitiveProperty .
    ?b ?a ?c .
    ?c ?a ?d
}
```

soluzione che comporta lo sviluppo di un processo per risalire a tutte le informazioni che hanno portato al risultato. Anche inserendo le premesse all'interno del *construct*, si verificheranno casi in cui non sarà possibile risalire alla costruzione della nuova informazione. Si consideri un ontologia con le seguenti triple:

```
primo parente secondo
secondo parente terzo
parente rdf:type owl:TransitiveProperty
alpha parente beta
beta parente gamma
```

se applichiamo la regola di inferenza della transitività, otteniamo il seguente risultato:

```
primo parente terzo
alpha parente beta
primo parente secondo
secondo parente terzo
parente rdf:type owl:TransitiveProperty
alpha parente beta
beta parente gamma
```

che costituisce, di fatto, un risultato sbagliato.

Rimanendo nello stesso ambiente, ma abbandonando l'uso del CONSTRUCT, si può utilizzare una SELECT SPARQL. La regola di inferenza sulla transitività verrebbe riscritta nel seguente modo:

```
SELECT
?a ?b ?c ?d
WHERE {
?a rdf:type owl:TransitiveProperty .
?b ?a ?c .
?c ?a ?d
}
```

che applicata all'ontologia definita prima restituirebbe i seguenti risultati:

```
parente primo secondo terzo
parente alpha beta gamma
```

basterà prendere, per ogni riga del risultato, i valori associati alle variabili contenute all'interno della *query* per ottenere il risultato finale e le triple da cui è stato generato.

In figura 4.4.1 è evidenziato il processo di esecuzione delle regole di inferenza con le classi che partecipano a questa operazione. Il processo consiste, una volta ricevuto in input una regola di inferenza, nel convertire la regola dall'oggetto di dominio *InferenceRule* ad un formato adatto ad un *query* di ricerca tramite il metodo *createSPARQLQueryFromInferenceRule*. Successivamente, la regola convertita in una *query sparql*, viene eseguita sull'ontologia e le *tuple* (i risultati della ricerca) restituiti dalla *query* vengono a loro volta riconvertiti in oggetti utilizzabili nel sistema attraverso il metodo *tupleToArtStatament* che ritorna un oggetto di tipo ARTStatement, tipo di dato utilizzato dalle OWL-ART per rappresentare gli *statements* di un ontologia. L'insieme di questi oggetti verrà poi restituito come output dell'operazione.

4.5. Implementazione della gestione dell'output

Se l'utente che utilizza il *reasoner* sceglie di attivare la memorizzazione del processo di generazione delle nuove informazioni, allora, durante l'operazione di *reasoning*, le nuove triple che verranno trovate dovranno essere inviate al gestore dell'output che provvederà a salvare nelle proprie strutture dati. L'utente, al termine dell'operazione di *reasoning*, potrà chiedere al gestore dell'output di visualizzare le informazioni che ha raccolto durante l'operazione. Come si può vedere in figura 4.5.1, ci sono due diverse modalità con le quali vengono salvate le nuove informazioni generate:

- Salvataggio tramite una lista: ogni elemento della lista è rappresentato da un oggetto che al suo interno contiene la singola nuova informazione generata con una lista delle informazioni che l'hanno generata e la regola di inferenza dalla quale è stata trovata la nuova informazione.
- Salvataggio su un grafo.

L'ultimo punto merita un spiegazione più approfondita. Durante l'operazione di *reasoning* può avvenire che la scoperta di una nuova informazione (ossia l'esplicitazione di una informazione che era implicita all'interno dell'ontologia) contribuisca a sua volta, in nuova esecuzione dell'operazione di *reasoning* (ricordiamo che l'utente può specificare quante volete eseguire il *reasoning* all'interno di una singola esecuzione del sitema) nell' esplicitamento di una nuova tripla: in questo modo abbiamo che una nuova informazione

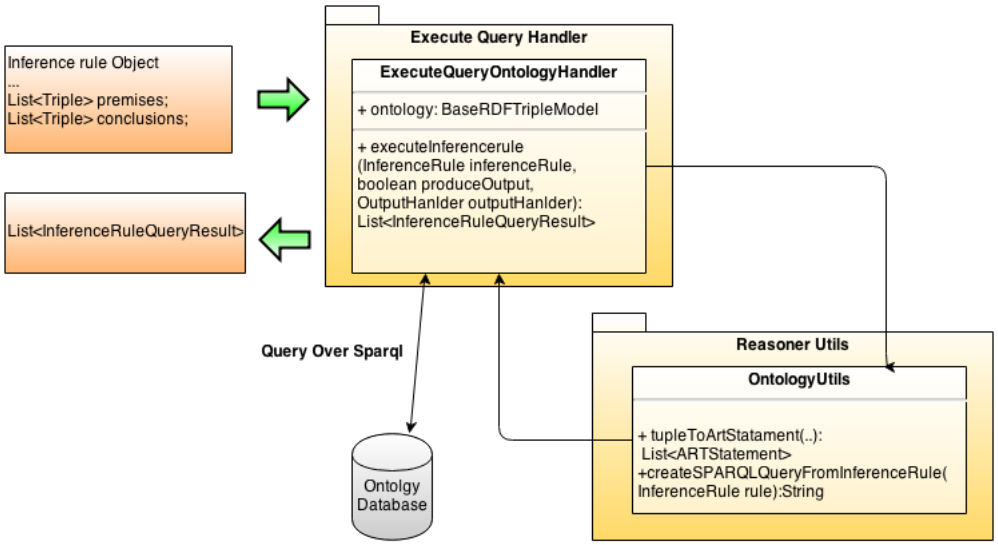


FIGURA 4.4.1. Esecuzione *query*

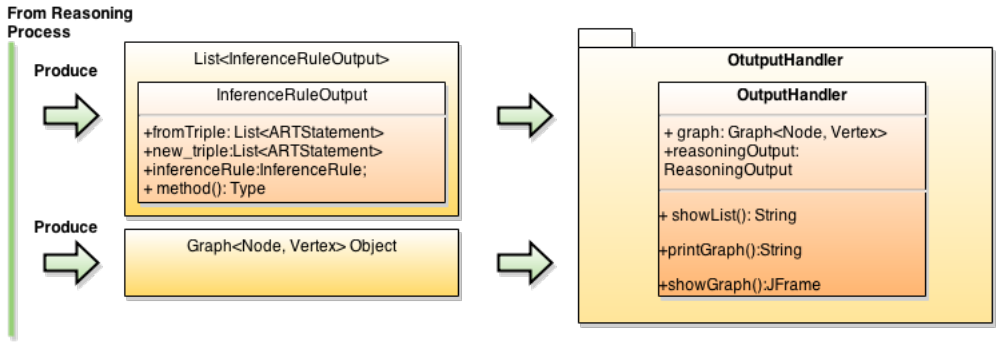


FIGURA 4.5.1. Salvataggio dell'output

che è stata trovata ad una generica operazione di *reasoning* apre la strada per trovare un ulteriore nuova tripla in un esecuzione successiva. È utile, quindi, avere un meccanismo che ci permette di risalire, data una nuova informazione, a tutte le triple che l'hanno generata. Nell'implementazione attraverso la lista questo procedimento non è possibile in quanto la nuova tripla è corredata solamente dagli ultimi dati da cui è stata generata: se ad esempio all'interno di questi dati c'è una tripla che è stata trovata durante l'esecuzione di una precedente operazione di *reasoning*, la sua storia sarà contenuta in un altro elemento della lista. Con l'ausilio di un grafo è possibile interconnettere le nuove informazioni (nodi del grafo) attraverso una serie di relazioni (archi del grafo) che rappresentano la relazione "generata da:", in questo modo è possibile ottenere, per ogni nodo del grafo, tutte le informazioni che hanno generato quel nodo. Vediamo un esempio. Supponiamo di avere l'ontologia, espressa nel listato 4.2, sulla quale vogliamo applicare la regola di inferenza che "scova" le relazioni transitive (mostrata nel listato 4.3): Scegliendo di eseguire per due volte l'operazione di *reasoning*, il grafo risultante sarà costruito nel modo descritto in figura 4.5.2 . Quando l'utente chiederà si visualizzare le nuove triple inferite attraverso il grafo, potrà vedere tutte le informazioni da cui sono state generate.

Il grafo utilizzato per visualizzare l'output è stato utilizzando usando le classi della libreria JUNG Graph che mette a disposizione dello sviluppatore numerosi costrutti per la costruzione di un grafo diretto. Inoltre, sono disponibili alcune classi che permettono la visualizzazione del grafo basate sulla libreria grafica di *Java Swing*[**32**]. Combinando in modo opportuno le classi per la visualizzazione del grafo si è riusciti ad ottenere il risultato mostrato in figura figura 4.5.3 . Con questa visualizzazione è possibile modificare il *layout* del grafo, eseguire lo zoom, ruotare il grafo ed infine spostare i nodi del grafo. Mediante questi strumenti l'utente può ottenere una visualizzazione del grafo in base alle proprie esigenze. Cliccando su un nodo del grafo si ottengono (sempre sotto forma di grafo) le triple che hanno

LISTING 4.2. Esempio di un'ontologia

```
....
<Nazione rdf:ID="Italia" />
<Regione rdf:ID="Lazio">
        <locatedIn rdf:resource="#Italia"/>
</Regione>
<Provincia rdf:ID="Roma">
<locatedIn rdf:resource="#Lazio"/>
</Provincia>
<Citta rdf:ID="Ariccia">
        <locatedIn rdf:resource="#Roma"/>
</Citta>
....
```

LISTING 4.3. Regola di inferenza per le relazioni transitive

```
new rule :
name: RegolaTransitiva
id:1
premise: subject:?p predicate: type object: transitiveProperty
premise: subject: ?a predicate: ?p object: ?b
premise: subject: ?b predicate: ?p object: ?c
conclusion: subject: ?a predicate: ?p object: ?c
```

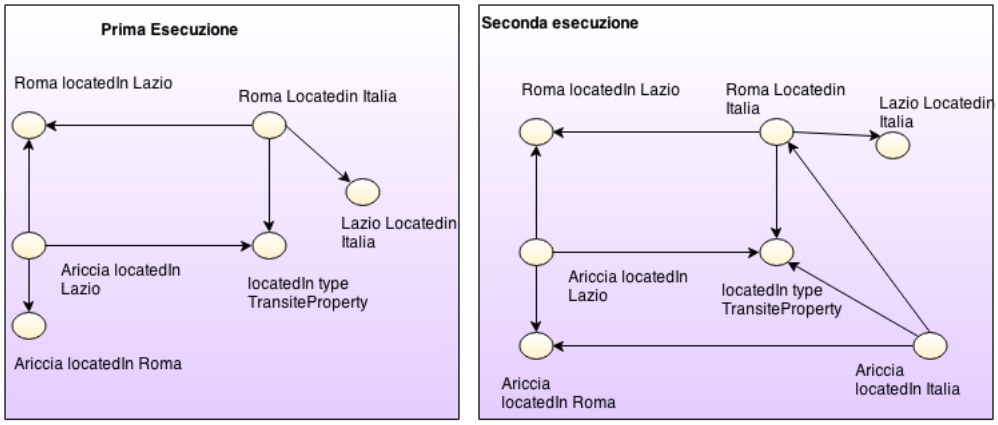


FIGURA 4.5.2. Costruzione del grafo dell'output

generato la tripa rappresentata del nodo del grafo: questa funzionalità è stata ottenuta scrivendo una procedura che implementasse una visita in ampiezza sul grafo delle nuove informazioni.

4.6. Implementazione dell'operazione di reasoning

Ora che tutti i componenti che cooperano all'operazione di *reasoning* sono stati definiti è giunta l'ora di mostra l'implementazione del cuore dell'applicazione: il modulo del reason*ing*. Questo modulo usa tutti gli alti componenti del sistema come base per eseguire le proprie funzioni, vediamo in dettaglio come vengono implementate (listato 4.4) : nella prima parte della funzione vengono inizializzati i cicli per l'applicazione delle regole di inferenza in base ai parametri di configurazione scelti dall'utente. In particolare il primo ciclo itera sul numero delle volte che le regole di inferenza devono essere applicate, mentre il secondo ciclo itera sulle regole di inferenza che devono essere eseguite. All'interno del secondo ciclo viene richiamato, attraverso la classe *executeQueryOntologyHandler* (definita nella sezione 4.4), il modulo per l'esecuzione delle regole di inferenza che converte le regole di inferenza in *query* da applicare

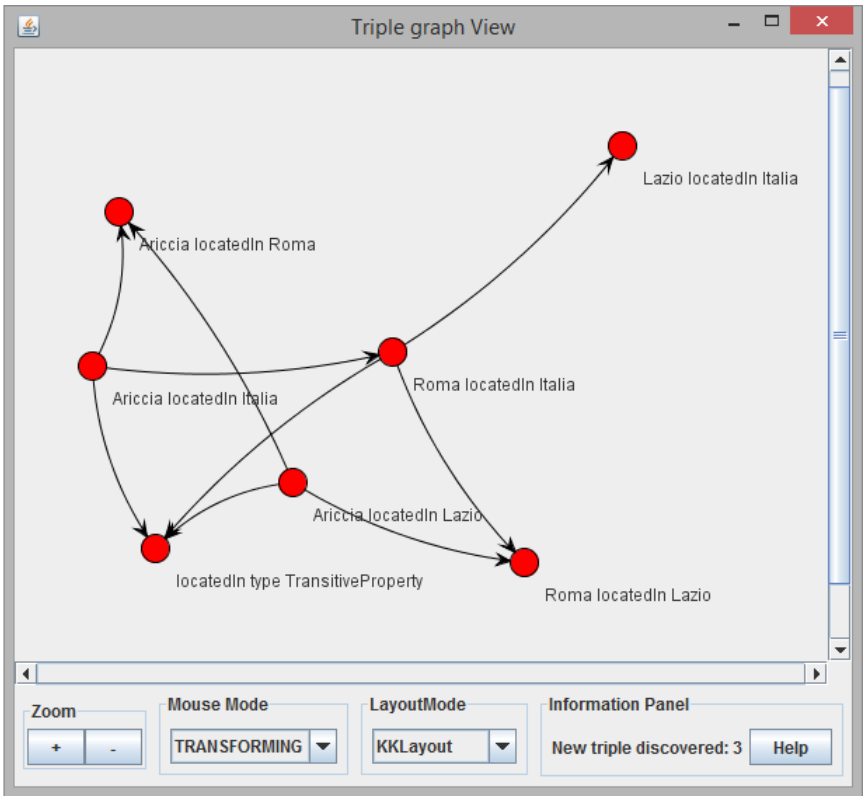


FIGURA 4.5.3. Esempio di grafo che utilizza la libreria Jung Graph

LISTING 4.4. Metodo per l'esecuzione dell'operazione di *reasoning* - prima parte

```
public class reasoner {
....
 public List<ARTStatement> reasoning() {
 //Apply all inference rules on ontology
 //as many times as expressed by the configuration parameters
 for(int count=0;count <numberOfExecution;count++){
  //Execute each single inference rule on ontology
  for(InferenceRule inferenceRule: inferenceRules.getInferenceRules() ){
   //Get the result of execution of query.
   List<InferenceRuleQueryResult> inferenceRuleQueryResults =
   executeQueryOntologyHandler.executeInferencerule(inferenceRule);
   ....
   ....
    }
   }
  }
}
```

sull'ontologia e restituisce i risultati come una lista di oggetti di tipo *InferenceRuleQueryResult.* Successivamente vengono eseguite le linee di codice rappresentate nel listato 4.5. Per ogni nuova informazione che è stata trovata si controllerà, attraverso il metodo *isStatementisOnOntology,* che essa non sia già presente all'interno dell'ontologia *:* in caso positivo verrà aggiornata la lista che memorizza le nuove triple trovate e verrà creato un oggetto da inviare al gestore dell'output. Il metodo *produceOutput* si occuperà di controllare la produzione dell'output e ,in caso positivo, creerà un nuovo oggetto contenente l'informazione da invierà al modulo dell'output e aggiungerà la nuova informazione all'interno del grafo dell'output. Una volta terminati i rispettivi cicli delle regole di inferenza e sul numero di volte che devono essere eseguite, rimarrà un ultimo passaggio che consiste nelle linee di codice mostrate nel listato 4.6; le informazioni esplicitate vengono inviate al gestore dell'output e restituite alla *facade* che gestisce l'operazione di *reasonng.* Il processo di *reasoning* è descritto in figura 4.6.1.

LISTING 4.5. Metodo per l'esecuzione dell'operazione di *reasoning* - seconda parte

```
public List<ARTStatement> reasoning() {
....
for(InferenceRuleQueryResult inferenceRuleQueryResult:inferenceRuleQueryResults){
  if( !isStatementisOnOntology(inferenceRuleQueryResult, baseRDFTripleModel)){
  //Add all new triples on output result.
  results.addAll(inferenceRuleQueryResult.getTupleToConclusion());
  //Create new object to store the output of reasoning operation
  InferenceRuleOutput inferenceRuleOutput =
    produceOutput(
        produceOutput,inferenceRuleQueryResult,inferenceRule,stringGraph);
        //Add the output on the list of output
        inferenceRulesOutuput.add(inferenceRuleOutput);
         }
  }
...
}
```

LISTING 4.6. Metodo per l'esecuzione dell'operazione di *reasoning* - terza parte

```
public List<ARTStatement> reasoning() {
...
...
//Save the results of reasoning operation
inputOutputHanlder.saveQuery(inferenceRulesOutuput);
//Save graph
inputOutputHanlder.setGraph(stringGraph);
//Return resoning result
return results;
}
```

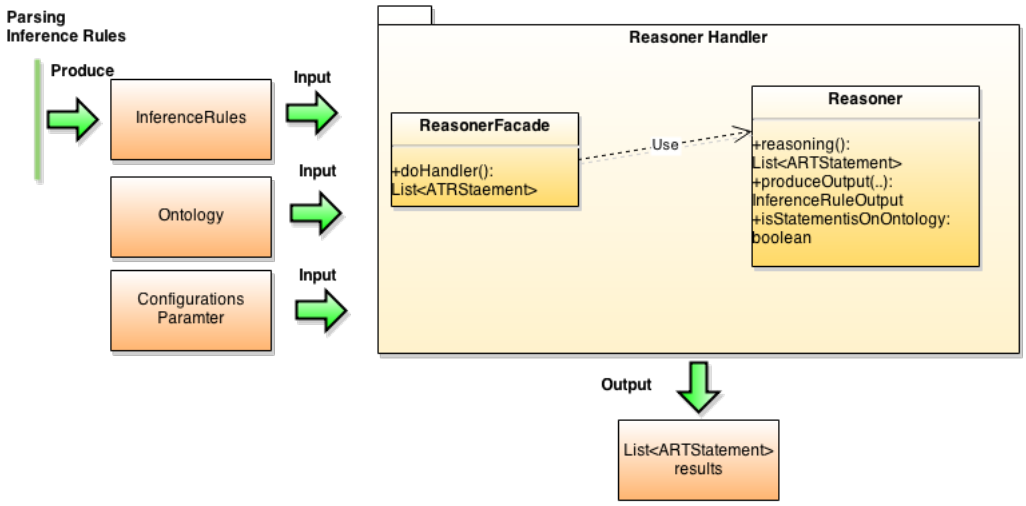


FIGURA 4.6.1. *Reasoner module*

Nella sezione 3.2 abbiamo descritto l'esistenza di alcune classi che orchestrano le varie operazioni logiche del sistema. Queste classi rispondono ad una classe superiore che "comanda" loro di eseguire le operazioni per cui sono state sviluppate. La classe di cui parliamo è l'*entry point* del sistema ovvero la classe con cui l'utente è costretto ad interagire per usufruire delle funzionalità offerte dal *reasoner*. Il nome di questa classe è *StartReasonerFacade* che si occupa di delegare i vari *task* ai moduli che hanno le competenze per eseguirli: in sostanza mette insieme tutti i moduli del sistema per raggiungere l'obiettivo finale di creare un operazione di *reasoning*. Analizziamo brevemente questa classe (mostrata nel listato 4.7) : il metodo principale della classe ha come input l'ontologia e il file delle regole di inferenza specificato dall'utente. Come primo passo vengono *istanziati* gli oggetti relativi alla gestione dei parametri di

LISTING 4.7. *Facade* per la gestione dell'interno processo di *reasoning*

```
public List<ARTStatement> startReasoner(BaseRDFTripleModel inputOntology,
                                        File inferenceRuleFile){

 List<ARTStatement> newInferredTriples = new ArrayList<ARTStatement>();
 //Create new configuration Parameter
 ConfigurationParameter configurationParameter =ConfigurationParameter.getIstance();
 //Create Input/Output handler
 inputOutputHanlder = new OutputHanlder();
 //Create new parserFacade
 ParseInferenceRulesFacade parseInferenceRuleFacade =
            new ParseInferenceRuleFacade(configurationParameter,inferenceRuleFile);
    //Parse inference rules from file
 InferenceRules inferenceRules = parseInferenceRulesFacade.doHandler(inferenceRuleFile);
    //Create ReasonerFacade
            ReasonerFacade reasonerFacade = new ReasonerFacade();
    //reasoning on ontology
 newInferredTriples=
    reasonerFacade.doHandler(
    inferenceRules,configurationParameter,inputOutputHanlder,inputOntology);
        //return new triple
        return newInferredTriples;
    }
```

LISTING 4.8. *Facade* per la gestione dell'interno processo di *reasoning*

```
public void setParameter(String key,String value) {
                ....
        }

public ReasoningOutput getOutputList(){
            ...
}

public void showGraphOnWindow(){
                ...
        }
```

configurazione e alle gestione dell'output, successivamente viene creato il gestore dell'operazione di *parsing* delle regole di inferenza e viene delegato alla *ParseInferenceRuleFacade* di iniziare tale operazione. Con le regole ottenute dalle lettura del file viene creata la *ReasonerFacade* per la gestione dell'operazione di *reasoning* e consecutivamente viene richiesta di attuare l'operazione di *reasoning* sull'ontologia restituendo il risultato dell'operazione all'utente. Inoltre, sono a disposizione dell'utente i metodi espressi nel listato 4.8: *setParameter* permette all'utente di modificare i parametri di configurazione dell'operazione di *reasoning*, *getOutputList* restituisce la lista delle nuove triple trovate (se si è scelto di produrre l'output) in accordo con la specifiche descritte nella sezione 4.5 e *showGraphOnWindow* mostra all'utente il grafo creato durante l'operazione di *reasoning*.

L'avere implementato il sistema seguendo le linee guida stabilite in fase progettuale ha garantito che l'accesso alle funzionalità offerte dal sistema non avvenisse interagendo direttamente con le classi interessate a svolgere queste funzioni ma attraverso l'utilizzo di una classe intermedia che si occupa di recuperare le informazioni per l'utente. Questo meccanismo rispetta a pieno gli standard dello sviluppo di applicazioni che usano il paradigma orientato agli oggetti poiché si è mantenuto un basso livello di accoppiamento delle classi (le implementazioni delle classi sono nascoste dalle facciate) e un alta coesione tra le classi (ogni classe esegue azioni appartenenti solo al proprio dominio di azione). In particolare, l'alta coesione garantisce all'applicazione le caratteristiche di robustezza, affidabilità e riusabilità delle classi sviluppate: slegando la logica del processo dalle classi che la eseguono, è possibile rimuovere i vari componenti del sistema ed integrarli in altri progetti senza stravolgere la loro implementazione o senza costruire sovrastrutture per adattarle a sistemi preesistenti.

CAPITOLO 5

Analisi Performance

In questo capitolo verranno mostrati alcuni esempi di esecuzione del sistema su diversi tipi di ontologia e con diversi tipi di configurazione dei parametri dell'operazione di *reasoning*, prestando particolare attenzione sulle risorse utilizzate durante l'operatività del sistema. Le risorse oggetto dell'analisi delle performance sono:

- *Heap Memory Usage* : parametro che riflette l'uso della memoria usato dagli oggetti che vengono istanziati dal linguaggio *Java* all'interno della *JavaVirtualMachine*(ambiente di esecuzione del linguaggio Java).
- *Thread number*: numero dei *thread* creati durante l'esecuzione del software.
- *Classes*: il numero delle classi che vengono create durante tutta l'esecuzione del software.
- *CPU Usage*: l'utilizzo della CPU durante l'esecuzione del programma

L'analisi delle performance sono state eseguite su tre tipi di configurazione del sistema: la prima, simula l'uso del programma in ambienti con un basso carico di operazioni, la seconda con un medio carico di operazioni e la terza un alto carico di operazioni. Per tutti e tre gli ambienti verrà usato lo stesso file delle regole di inferenze (a seconda dell'ambiente verranno scelte diverse regole da eseguire) contenente quelle regole che rappresentano alcune *property characterstic* descritte nella *OWL Web Ontology Language Guide* e nell'*OWL 2 Web Ontology Language Profiles* (http://www.w3.org/TR/owl2-profiles/#Reasoning_in_OWL_2_RL_and_RDF_Graphs_using_Rules) redatte dalla W3C, nello specifico :

(1) *TransitiveProperty*: se una proprietà P è specificata come transitiva allora per ogni *x,*y e z vale $P(x,y) \wedge P(y,z) \longrightarrow P(x,z)$
(2) *SymmettricProperty:* se una proprietà P è specificata come simmetrica allora per ogni *x,y* vale $P(x,y) \Longleftrightarrow P(y,x)$
(3) *inverseOfProperty:* se una proprietà *P1* è specificata come *inverseOf* di *P2* allora per ogni *x,y* vale$P1(x,y) \Longleftrightarrow P2(y,x)$
(4) *InverseFunctionalProperty:* se una proprietà P è specificata come *inverseFunctional* allora per ogni *x,y* e z vale $P(x,y) \wedge P(z,x) \rightarrow y = z$
(5) *FunctionalProperty:* se una proprietà P è specificata come funzionale allora per ogni *x,y* e z vale $P(x,y) \wedge P(x,z) \longrightarrow y = z$
(6) *EquivalentProperty:* se una proprietà *P1* è specificata come *equivalentProperty* allora per ogni *x,y* vale $P1(x,y) \longrightarrow P2(x,y) \wedge P2(y,z)$ dove *P2* è una proprietà specificata come *subPropertyOf.*
(7) *SubClassProperty:* se una proprietà P è specificata come *subClassOf* allora per ogni *x,y* e z vale $P(x,y) \wedge P(y,z) \longrightarrow P(x,z)$

Nella tabella 1 sono elencati i parametri di configurazione per i tre ambienti in cui verranno effettuati i test. Oltre ai parametri di configurazione dell'operazione di *reasoning*, sono presenti i dati riguardanti

(A)		(B)		(C)	
produce output	false	produce output	true	produce output	true
number of execution	1	number of execution	5	number of execution	10
which rule execute	1,2,7	which rule execute	1,2,3,7	which rule execute	1,2,3,4,5,6,7
numero triple ontologia	3483	numero triple ontologia	5941	numero triple ontologia	7847

Tabella 1. Configurazione ambienti di esecuzione

il numero delle triple o *statements* presenti all'interno dell'ontologia usata negli ambienti di esecuzione delle prove. Le ontologie usate sono state reperite ai seguenti link:

- www.cs.man.ac.uk/∼stevensr/ontology/family Ontologia usata nel primo ambiente
- http://owl.cs.manchester.ac.uk/co-ode-files/ontologies/pizza Ontologia usata nel secondo ambiente
- www.w3.org/TR/owl-guide/wine Ontologia usata nel terzo ambiente

5.1. Esecuzione Test

Per monitorare le attività e l'uso delle risorse durante l'esecuzione del programma è stato utilizzato un software presente nativamente all'interno del *Java development Kit* chiamato *JConsole* [**44**]. *JConsole* è basato sulla strumentazione JMX(*Java Management Extensions*[**45**]) ed è presente all'interno della *Java virtual machine* (JVM); esso fornisce informazioni sulla performance e sulle risorse consumante durante il *running* delle applicazioni all'interno della piattaforma *java* [**44**].

Nella figure 5.1.1, 5.1.2 e 5.1.3 vengono illustrati, per ogni ambiente di esecuzione, i grafici che mostrano l'utilizzo delle risorse elencate nella precedente sezione. Invece, in figura 2 sono mostrate il numero

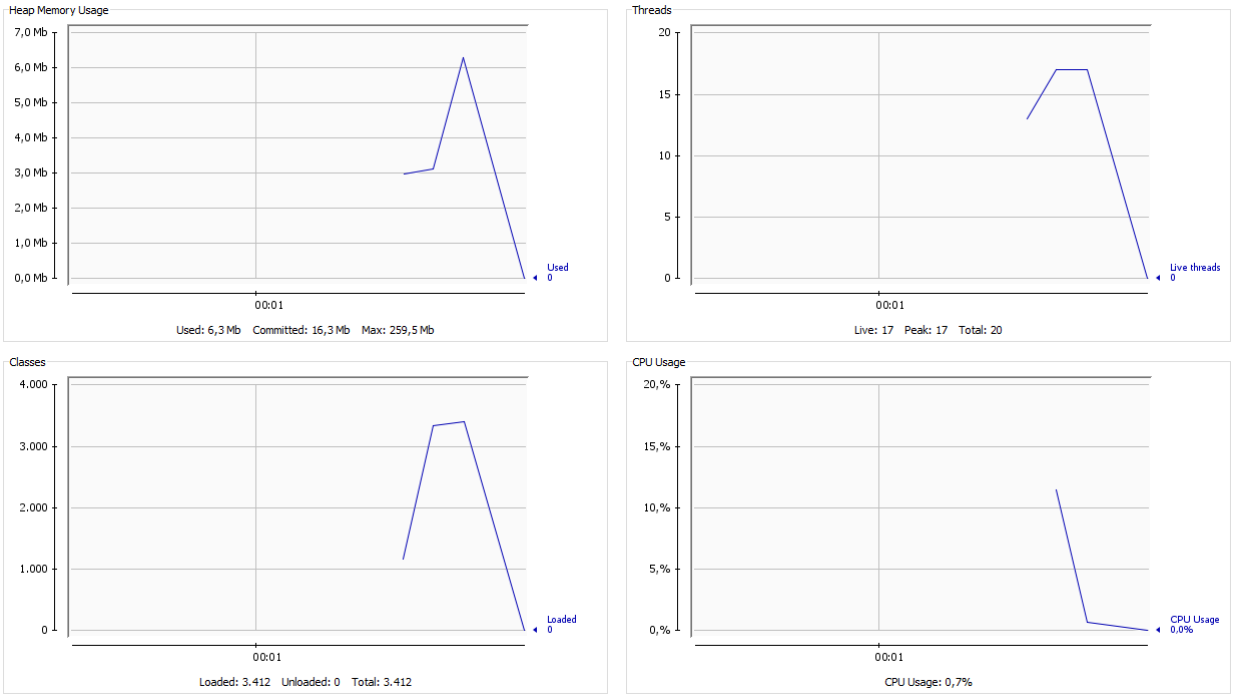


FIGURA 5.1.1. Grafici relativi al primo ambiente di esecuzione dei test

delle nuove informazione che sono state trovate durante l'esecuzione dei test. Come si può osservare

Test	Triple trovate
Primo	6
Secondo	62
Terzo	88

TABELLA 2. Numero di nuove informazioni trovare durante l'esecuzione dei test

dalle figure precedenti, l'utilizzo delle risorse è strettamente dipendente dai parametri di configurazione del *reasoner* e dal numero delle triple contenute all'interno dell'ontologia utilizzata nei vari ambienti di esecuzione dei test. Infatti, mentre nel primo grafico otteniamo dei valori dell'utilizzo delle risorse abbastanza bassi, nei grafici che rappresentano gli scenari dei due rimanenti test si ha un graduale aumento del consumo delle risorse utilizzate durante l'esecuzione dell'operazione di *reasoning.* Nonostante si abbia un aumento del carico di lavoro che deve essere svolto dal *reasoning* tra il secondo ambiente di esecuzione e il terzo, i valori delle risorse consumate cresce in modo relativamente basso. A fronte di un numero di triple che passa da 5941 nel secondo ambiente ad un numero di 7847 nel terzo e dall'applicazione dell'operazione di *reasoning* che passa da 5 volte a 10 volte tra i due ambienti, si ha un aumento delle

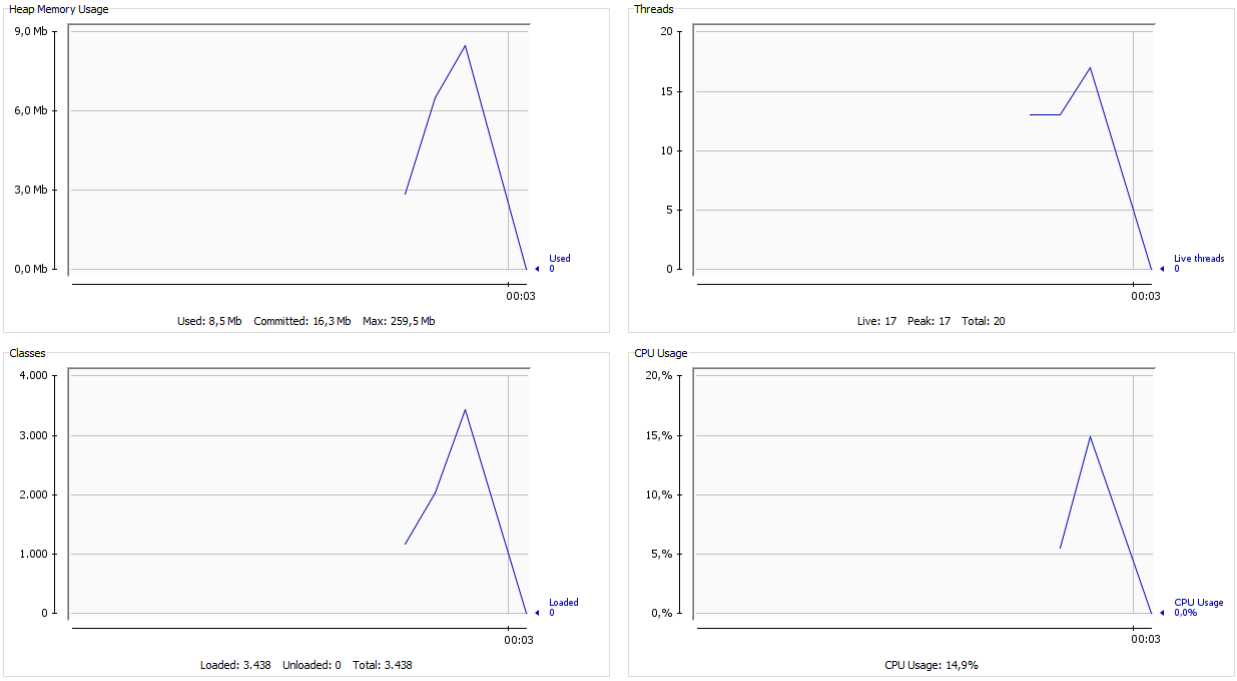


FIGURA 5.1.2. Grafici relativi al secondo ambiente di esecuzione dei test

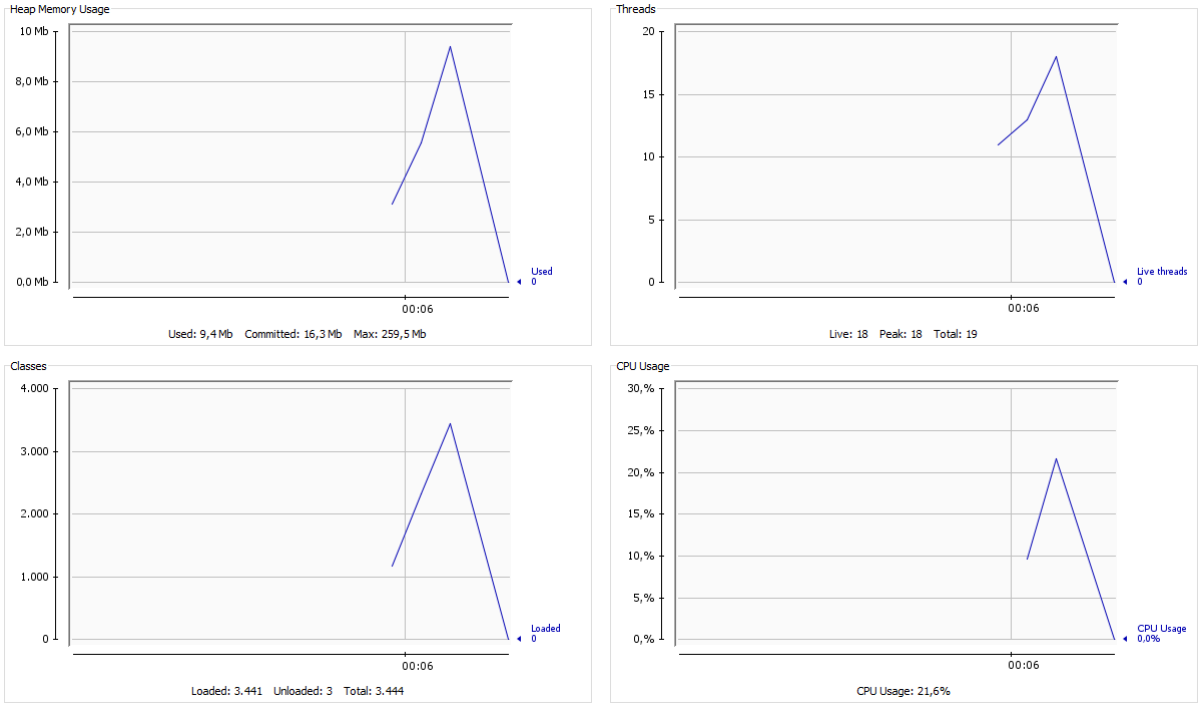


FIGURA 5.1.3. Grafici relativi al terzo ambiente di esecuzione dei test

risorse utilizzate tra il secondo test e il terzo test che varia tra un *range* del 2%-3% per quanto riguarda l'*Heap memory usage*, i *threads* e le classi e del 10% per l'utilizzo della *CPU* del sistema. Nella tabella 3 vengono messe in evidenza le differenze tra i valori delle risorse utilizzate nei tre ambienti di esecuzione.

5.2. Considerazioni sull'analisi delle performance

Il tempo di esecuzione dell'operazione di *reasoning*, oltre a dipendere dal numero delle volte che l'intera operazione deve essere ripetuta e dal numero delle regole di inferenza da applicare, dipende

Parametri	Primo Test	Secondo test	Terzo test
Heap memory usage	6,3mb	8,5mb	9,4mb
Thread number	17	19	20
Classes	3,412	3,438	3,441
CPU	0,7%	14,9%	21,6%
Tempo di esecuzione	1sec-1,5sec	3-4sec	6-7sec

TABELLA 3. Risorse consumate nel secondo e terzo test

strettamente dall'esecuzione delle *query* SPAQRL all'interno dell'ontologia; applicando una regola di inferenza all'interno di un *RDF triple store* con molti dati, si potrebbero avere situazioni in cui la riposta ad una interrogazione del database impiegherebbe molto tempo ad arrivare, tempo dipendente dall'RDF *store* che si sta utilizzando come database: alcuni database potrebbero essere ottimizzati per eseguire certi tipi di operazioni, mentre con altre operazioni potrebbero avere un tempo di esecuzione molto alto. Ad esempio il *query optimizer* (il modulo che si occupa di ottimizzare le *query* secondo le politiche interne del *database mamangment system*) potrebbe scegliere di eseguire all'interno di questa query SPARQL

```
SELECT
?a ?b ?c ?d
WHERE {
?a rdf:type owl:TransitiveProperty .
?b ?a ?c .
?c ?a ?d
}
```

prima la ricerca delle informazioni che sono identificate da

```
?c ?a ?d
```

e successivamente eseguire

```
?a rdf:type owl:TransitiveProperty .
?b ?a ?c .
```

In questo modo nella prima parte della *query*, "?c ?a ?d" verranno restituite tutte le triple dell'ontologia e successivamente verranno confrontate con i risultati delle altre clausole del costrutto *where*: è naturale immaginare che eseguire un confronto con tutte le triple dell'ontologia costi molto di più rispetto all'esecuzione di un confronto con un numero ristretto di risultati ottenuto applicando come clausola inziale "?a rdf:type owl:TransitiveProperty " che effettua una scrematura delle triple presenti nell'ontologia. Non sapendo a priori quali regole verranno applicate, eseguire operazioni di *tuning* sulle query risulterebbe difficili se non impossibile per via dell'enormità delle possibili combinazione di *query* che possono essere create. Anche adottando una strategia di *tuning* globale potremmo incappare in situazioni in cui il tempo di esecuzione aumenterebbe invece di diminuire.

Tralasciando il problema relativo all'esecuzione delle *query* SPARQL che non dipendono strettamente da come è stato progettato il sistema e analizzando i valori mostrati nella tabella 3, si evince che il *reasoner* è in grado di consumare un numero limitato di risorse all'aumentare del carico delle operazioni che devono essere svolte all'interno dell'ontologia. Le regole di inferenza che sono state utilizzate durante i test rappresentano quelle regole che possono produrre, agendo direttamente sugli elementi del dominio, il maggior numero di nuove informazioni utili per aumentare il grado di relazione dei dati all'interno dell'ontologia. Complessivamente, quindi si è cercato di creare degli ambienti che fossero il più possibile vicini ad un utilizzo reale e concreto del sistema da parte dell'utente finale: soffermarsi sulla creazione di regole che favorissero il sovraccarico del sistema rispetto al reale contributo per l'esplicitazione delle informazioni all'interno dell'ontologia è sembrato un procedimento poco funzionale al fine di rappresentare l'utilizzo del sistema all'interno di un ambiente reale.

Parte III Integrazione con Semantic Turkey

Dopo avere implementato il sistema del *reasoner*, vederemo l'utilizzo di quest'ultimo in un ambiente reale. Il *reasoner* verrà integrato all'interno di un sistema per la gestione della conoscenza, sviluppato dal gruppo di intelligenza artificiale dell'università Tor Vergata di Roma, presente in commercio sotto il nome di *Semantic Turkey*.

Nel capitolo 6 sarà analizzata l'integrazione dell'architettura del *reasoner* con l'architettura di *Semantic Turkey*, l'implementazione delle strutture per la comunicazione e l'integrazione tra i due sistemi le tecnologie usate per raggiungere tale scopo. Infine, nel capitolo 7, saranno presentate le conclusioni della tesi.

CAPITOLO 6

Semantic Turkey, Editor di Ontologie

In questa parte ci occuperemo di come integrare il sistema del *reasoner* all'interno del sistema di Semantic Turkey.

6.1. L'ambiente operativo di Semantic Turkey

Semantic Turkey(ST) è una piattaforma per la gestione e l'acquisizione della conoscenza[**33**] sviluppata dal gruppo *Artificial Intelligence Research at Tor Vergata* (ART)[**33**]. La piattaforma, oltre ad offrire un *framework* per l'acquisizione, la gestione e lo scambio della conoscenza, mette a disposizione dell'utente un *tool* per l'editor di ontologie. Tramite questo editor è possibile operare sulle informazioni contenute all'interno delle ontologie attraverso delle funzionalità che offrono la creazione, modifica e gestione delle informazioni. Uno degli aspetti fondamentali della piattaforma di *Semantic Turkey* consiste nell'offrire allo sviluppatore un ambiente creato appositamente per inserire nuove funzionalità all'insieme di quelle preesistenti. L'obiettivo di questa parte consisterà nell'aggiungere ai servizi offerti da *Semantic Turkey*, in particolare nell'editor delle ontologie, la possibilità di eseguire l'operazione di *reasoning* sulle ontologie caricate all'interno del sistema.

L'architettura di *Semantic Turkey* è rappresentata in figura 6.1.1[**47**] ed è pensata per dare la possibilità di aggiungere nuove funzionalità attraverso il blocco riguardante le *service extension*; sarà proprio in questa parte che l'estensione del *reasoner* verrà posizionata. Come si vede dalla figura la sezione

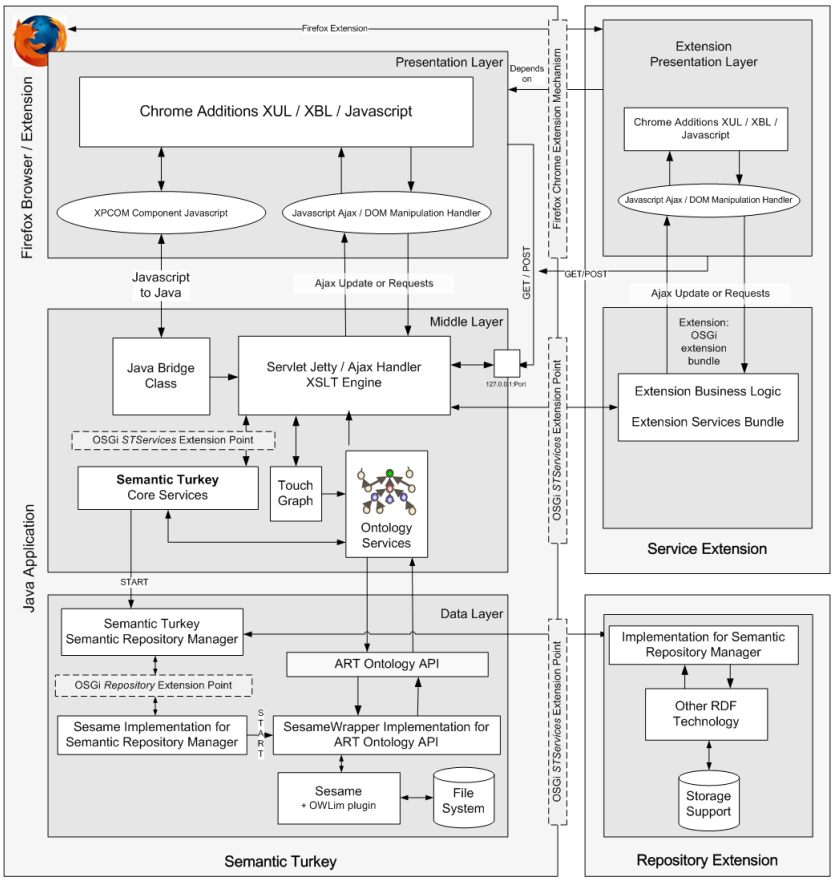


FIGURA 6.1.1. Architettura *Semantic Turkey*

LISTING 6.1. *Overlay* dei componenti della *GUI* di *firefox*

```
...
overlay
chrome://semantic-turkey/content/browser-overlay.xul
chrome://reasonerExtension/content/browser-overlay.xul
...
```

relativa alle estensione dei servizi è suddivisa in due parti: la prima, che estende il *presentation layer* di *Semantic Turkey* e la seconda che estende i servizi presenti nel sistema. In sostanza, i servizi che verranno creati nell'estensione saranno registrati all'interno dei servizi presenti in ST: quando l'utente richiederà di usufruire di un particolare servizio offerto dalla *reasoner extension*, ST controllerà l'esistenza di tale servizio e, in caso positivo, delegherà la richiesta al gestore inserito all'interno dell'estensione.

Semantic Turkey, come il *reasoner*, lavora con oggetti di dominio appartenenti alla libreria OWL-ART, libreria sviluppata da ART. Come è stato accennato durante la spiegazione dello sviluppo del *reasoner,* questa libreria fornisce uno strato di astrazione sulle differenti tecnologie utilizzare per scrivere le ontologie. Di conseguenza, l'output prodotto dal *reasoner* non necessiterà di alcuna conversione per essere utilizzato all'interno di ST, viceversa, l'acquisizione dell'ontologia da parte del *reasoner* non avrà bisogno di essere adattata ai modelli utilizzati nel *reasoner* stesso. Sviluppare il *reasoner* attraverso il livello di astrazione fornito dalle OLW-ART ci permette di applicare l'operazione di *reasoning* in modo del tutto indipendente dalle tecnologie utilizzate per descrivere le informazioni presenti nelle ontologie: ST, infatti, ha la possibilità di operare con gli *standards* W3C per la rappresentazione della conoscenza che operano attraverso le famiglie RDF come RDFS, OWL, SKOS and SKOS-XL. Questa proprietà rende raggiungibile l'obiettivo che era stato posto nella sezione 1.2 ossia di costruire un *reasoner* che fosse utilizzabile indipendentemente dal tipo di ontologia utilizzata.

6.2. Reasoner presentation layer

In questa sezione verrà descritta la parte implementativa relativa allo sviluppo del *presentation layer* dell'estensione del *reasoner*. Questa componente si occupa di raccogliere le richieste dell'utente, che avvengono tramite l'interfaccia grafica, e di inviarle alla parte server di ST che si occuperà di gestire tali richieste. Il *presentation layer* dell'estensione del *reasoner* è pensato come un estensione del *presentation layer* del sistema di *Semantic Turkey*, pertanto deve rispettare i vincoli imposti dall'implementazione usata nella parte di *presentation* di ST. Il *presentation layer* di ST è stata pensata come un *add-on* del *browser web Firefox*[**36**] scritto usando il linguaggio *XUL*. La creazione di un estensione di ST è vincolata alla creazione di un estensione di *firefox* e deve essere scritta con l'intento di creare un *add-on*: in pratica, il *presentation layer* dell'estensione deve essere, a sua volta, un estensione dell'*add-on* di ST. Le tecnologie utilizzate per implementare l'estensione di *firefox* sono: il linguaggio XUL per rappresentare gli oggetti da visualizzare nella GUI come le finestre o le *textarea*, il linguaggio *Javascript* per gestire l'interazione con la GUI e il file *.manifest* per gestire la struttura, con i relativi percorsi ai file, dell'estensione[1].

Nel file *.manifest*, oltre a definire i *path* per i file contenuti all'interno l'estensione, è possibile aggiungere alcune proprietà per eseguire un *overlay* dei componenti esistenti nel *browser web*. Nel nostro caso, vogliamo aggiungere una voce nel menù dell'estensione *firefox* di ST andando ad eseguire un *overlay* su un componente visivo scritto in *XUL*(in questo caso l'azione di *overlay* consiste solamente nell'aggiungere un nuovo item ad un menù). L'istruzione che permette di ottenere questo comportamento è mostrata nel listato 6.1: mentre il file *.xul* è espresso nel listato 6.2 che specifica l'esistenza di un *item* all'interno del menù con id *menu_ToolsPopup2* (menù di Semantic Turkey) e come funzione associata al click sull'item la funzione *Javascript showReasonerConfiguration*. I componenti che non devono sovrascrivere alcun oggetto presente nel *browser web* ma che godono di un indipendenza propria sono:

- una finestra per gestire i parametri di configurazione del *reasoner*
- una finestra per la visualizzazione dell'output.

Da questi componenti sarà possibile usufruire dei servizi messi a disposizione dal *reasoner*. In figura 6.2.1 è mostrata l'interfaccia per la gestione dei parametri di configurazione del *reasoner*. L'utente, tramite l'ausilio della finestra, può configurare i parametri in accordo con i requisiti espressi nella sezione 3.1:

- Quali regole di inferenza eseguire

[1]Sono state elencate solamente le tecnologie più importanti ai fini dell'implementazione dell'estensione. Un estensione per il browser web Firefox consta di una struttura che al suo interno utilizza varie tecnologie.

LISTING 6.2. Definizione del menù del *reasoner*

```
....
<menupopup id="menu_ToolsPopup2">
  <menu id="menuReasoner" label="reasoner">
    <menupopup>
      <menuitem id="ReasonerMenu"
                label="StartReasoner"
                oncommand="showReasonerConfiguration();"/>
    </menupopup>
  </menu>
</menupopup>
...
```

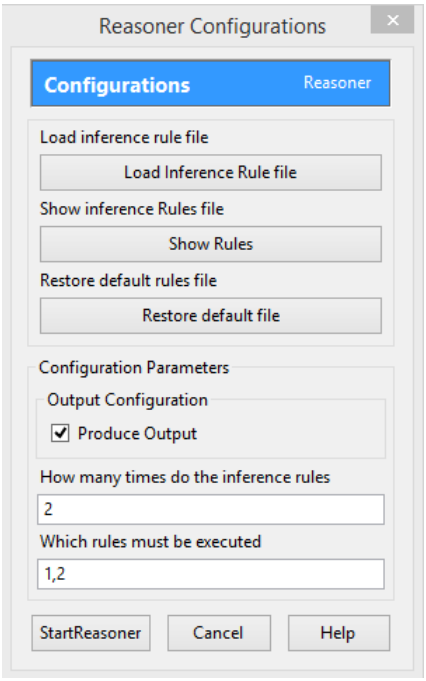


FIGURA 6.2.1. Finestra per la gestione della configurazione del *reasoner*

- Quante volte eseguire l'operazione di *reasoning*
- Scegliere di produrre l'output

Oltre ai parametri elencati l'utente potrà eseguire l'*upload* di un proprio file delle regole di inferenza, modificarlo *online* o usarne uno di *default precaricato* all'interno dell'estensione. Sarà sempre possibile, prima di dare il via all'operazione di *reasoning*, ripristinare il file di *default* delle regole di inferenza qualora, quello caricato dall'utente o quello modificato, non sia più utile ai fini del *reasoning*. Una volta scelti i parametri di configurazione e caricato il proprio file delle regole di inferenza (o scelto quello di *default*), l'utente potrà avviare l'operazione di *reasoning*. La richieste dei servizi, come la configurazione dei parametri o l'inizio dell'operazione di *reasoning*, vengono gestiti tramite funzioni *Javascript* contenute all'interno dell'estensione di *firefox*. Queste funzioni, provvedono ad utilizzare le funzioni HTTP Get e Post messe a disposizione da *Semantic Turkey* per inviare le richieste di utilizzo dei servizi al server di ST. Il modulo di ST che si occupa di eseguire il *forward* delle richieste, una volta ricevuta una *request*, provvederà a comunicare alla parte dell'estensione che estende i servizi di ST di rispondere alla richiesta e di inviare la risposta al *presentation layer* dell'estensione. Un esempio di funzione *Javascript* per la chiamata del servizio di *reasoning* è mostrata nel listato 6.3, che provvede ad inserire i parametri di configurazione del *reasoner* all'interno della richiesta da inviare al *server* di ST. La riposta, invece, viene gestita nel metodo espresso nel listato 6.4: nella prima parte viene controllato che la richiesta non abbia prodotto errori e, in caso positivo, si procede a recuperare le informazioni contenute all'interno della

LISTING 6.3. Funzione *javascript* per l'esecuzione dell'operazione di *reasoning*

```
function startReasoning(params){
var howManyTimesApllyInferenceRule =
"howManyTimesApllyInferenceRule="+params.out.cycleNumber;
var whicruleApply = "whicRulesApply="+params.out.whicruleApply;
var produceOutput = "produceOutput="+params.out.outputValue;
return HttpMgr.GET(serviceName,service.startReasoner,
howManyTimesApllyInferenceRule,whicruleApply,produceOutput);
}
```

LISTING 6.4. Gestione della risposta ricevuta dal server di ST in merito alla richiesta di esecuzione dell'operazione di *reasoning*

```
...
//Call the reasoning service from SemantickTurkey server.
var response = reasoner.Requests.ReasoningService.startReasoning(params);
...
//show the output of reasoning operation
if (response.getElementsByTagName("reply")[0].getAttribute("status") == "ok") {
//Get the information from response
//Number of new triples discovered
var triple_discovered =
response.getElementsByTagName("startReasoning")[0].
              getAttribute("numberOfTriple");
//Boolean to represent if output has been produced
var output =
response.getElementsByTagName("startReasoning")[0].
              getAttribute("produceOutput");
//The text output of reasoning
var print = response.getElementsByTagName("startReasoning")[0].
               getAttribute("printOutput");
//Create an array of parameters with the information taken
//from the server response
var parameters =
{inn:{newTriple:triple_discovered,
   produceOutput:output,printOutput:print}, out:null};
//If the output has not produced, show the dialog box
//to save the new triples discovered, else show the window output
if (output == 'false') {
window.openDialog("chrome://reasonerExtension/content/saveNewTriple.xul",
 "Output window", "chrome,centerscreen,modal,resizable=yes").focus();
        }
else{
         window.openDialog("chrome://reasonerExtension/content/showOutput.xul",
      "Output window", "chrome,resizable=yes",parameters);
                                    }
                            }
```

risposta per visualizzarle all'interno della GUI di Firefox.

Al termine dell'operazione di *reasoning*, se l'utente ha scelto di non produrre alcun output e se non si sono verificati degli errori, sarà mostrato un messaggio in cui verrà richiesto all'utente di salvare le nuove triple inferite all'interno dell'ontologia. Invece, se l'utente ha scelto di produrre l'output, verrà visualizzata una finestra in cui sarà possibile visionare le nuove informazioni in due modalità: la lista delle nuove triple e il grafo dell'operazione di reasoning. Un esempio è mostrato in figura 6.2.2. Nella visualizzazione testuale sarà possibile ricercare la storia di una singola tripla attraverso una *search box*, mentre, nella visualizzazione del grafo, questa funzionalità avverrà attraverso il click del mouse su un nodo della struttura. All'atto della chiusura della visualizzazione dell'output sarà possibile salvare le nuove triple inferite.

Nel sistema originale del *reasoner,* la parte relativa alla visualizzazione del grafo (descritta nella sezione 4.5) era stata implementata attraverso una finestra creata con l'ausilio delle librerie di *Java swing* e con quelle messe a disposizione dalla libreria per la creazione di grafi, *Jung Graph.* In pratica, la visualizzazione, veniva creata con una tecnologia che in *Java* porta il nome di *Java applet.* Purtroppo questa

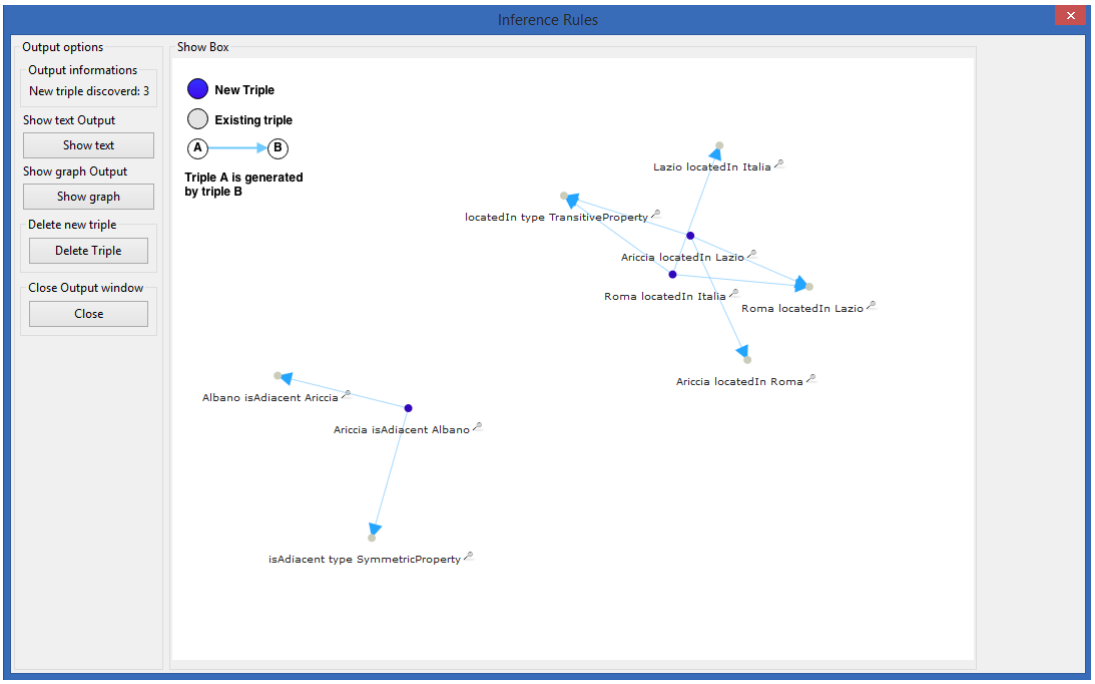


FIGURA 6.2.2. Esempio di visualizzazione dell'output

implementazione, ha portato, in fase di sviluppo dell'estensione del *reasoner*, seri problemi di compatibilità con l'ambiente di *Semantic Turkey*, con l'ambiente di *firefox* e con i più recenti aggiornamenti della *Java virtual machine* all'interno del *browser web.* Questi problemi sono dovuti principalmente a questioni di sicurezza che obbligano lo sviluppatore a fornire una firma digitale del pacchetto software contenente l'*applet,* firma con la quale la *jvm* autorizza ad utilizzare l'*applet* all'interno di una web *application* come l'estensione del *reasoner* [**37**]. Poiché questo passaggio, che deve essere ripetuto ogni volta che l'*applet* viene modificata, aumenta le operazioni di manutenibilità del codice, si è scelto di abbondare l'idea della visualizzazione tramite *applet.* Inoltre, per rendere indipendente la visualizzazione del grafo dalla tecnologia usata per rappresentarlo visivamente, si è scelto di lasciare il compito di visualizzazione del grafo all'applicazione che userà il *reasoner*: il *reasoner* restituirà solamente l'oggetto contenente le informazioni sul grafo.

All'interno dell'estensione del *reasoner*, per visualizzare il grafo è stata usata la libreria *JavaScript InfoVis Toolkit* (http://philogb.github.io/jit/) che opera su rappresentazioni di grafi espressi mediante la tecnologia *Json* [**38**]. Nel nostro caso, la parte dell'estensione del *reasoner* che si occupa di gestire i servizi, una volta che ricevuto in input il grafo costruito durante l'operazione di *reasoning*, lo convertirà in un formato *Json* da restituire alle API di *InfoVis*: il risultato di questa operazione è mostrato in figura 6.2.2. Un esempio di grafo dell'operazione di *reasoning* tradotto in *Json* è mostrato nel listato 6.5.

6.3. Reasoner extension services

La parte che si occupa di rispondere alle richieste dei servizi che arrivano dal gestore dei servizi di ST è l'*extension services* del *reasoner.* Come è stato detto precedentemente, questa parte si occupa di aggiungere i servizi, implementati nell'estensione, alla lista di quelli disponibili all'interno di *semantic tukery.* La definizione dei servizi avviene tramite l'utilizzo di appositi *bean* definiti all'interno dell'ambiente di *Spring*[**39**] utilizzato in ST, mentre la registrazione dei servizi presso *semantic tukery* è ottenuta mediante l'utilizzo di un *framework* (che a sua volta utilizza *Spring*) che implementa la logica *OSGi*[**40**]. La logica OSGi è stata utilizzata in ST per dare la possibilità di abilitare, all'interno del sistema, l'assemblaggio modulare dei software implementati con tecnologie *Java*[**41**], in altre parole, consiste nel progettare un architettura che esprime la proprietà di supportare l'inserimento di nuovi moduli che estendono quelli esistenti nell'architettura. Un esempio di definizione di un servizio dell'estensione del *reasoner* con relativa registrazione all'interno del *framework* OSGi è mostrata nel listato 6.6: nella prima parte viene definito il *bean* relativo al nuovo servizio disponibile nell'estensione mentre nella seconda parte il servizio viene registrato come estensione dei servizi OSGi.

LISTING 6.5. Grafo dell'output rappresentato in *json*

```
[ {"id":"Ariccia␣locatedIn␣Italia",
  "name":"Ariccia␣locatedIn␣Italia",
  "adjacencies":[{
          "nodeTo":"locatedIn␣type␣TransitiveProperty",
              "nodeFrom":"Ariccia␣locatedIn␣Italia",
                  "data":{"$type":"arrow"}
    },{
              "nodeTo":"Ariccia␣locatedIn␣Roma",
                  "nodeFrom":"Ariccia␣locatedIn␣Italia",
                  "data":{"$type":"arrow"}
          },{
          "nodeTo":"Roma␣locatedIn␣Italia",
                  "nodeFrom":"Ariccia␣locatedIn␣Italia",
                  "data":{"$type":"arrow"}
            }
    ]
         },{
    "id":"Roma␣locatedIn␣Lazio",
         "name":"Roma␣locatedIn␣Lazio",
         "adjacencies":[]},{
         "id":"locatedIn␣type␣TransitiveProperty",
         "name":"locatedIn␣type␣TransitiveProperty",
         "adjacencies":[]},{
         "id":"Ariccia␣locatedIn␣Roma",
         "name":"Ariccia␣locatedIn␣Roma",
         "adjacencies":[]},{      "id":"Lazio␣locatedIn␣Italia",
         "name":"Lazio␣locatedIn␣Italia",
    "adjacencies":[]},
         ....
    ....
]
```

LISTING 6.6. Definizione dei servizi per la gestione del *reasoner*

```
<bean id="reasoningService"
   class="it.uniroma2.reasoner.services.ReasoningService">
        ...
</bean>
....
<osgi:service ref="reasoningService"
interface="it.uniroma2.art.semanticturkey.plugin.extpts.ServiceInterface" />
```

L'interazione dei servizi con lo strato di *presentation* dell'estensione utilizza il *pattern* architetturale *Model View Controller*[**42**]. Un esempio di *controller* relativo alle richieste per i servizi di *reasoning* è illustrato nel listato 6.7: il metodo *getPreCheckedResponse*() si occupa di ascoltare le richieste per l'utilizzo dei servizi: quando viene inoltrata una richiesta da parte di ST, viene verificato che il *controller* sia in grado di gestire la specifica richiesta e, in caso positivo, chiama il metodo *startReasoning*() che si occuperà di avviare l'operazione di *reasoning* e di restituirne il risultato. Il metodo *startReasoning* è definito nel listato 6.8: nella prima parte vengono recuperati i parametri relativi alla configurazione del *reasoner* salvandoli all'interno del gestore delle configurazioni, successivamente viene preso il progetto dall'ambiente di ST contenete l'ontologia e su di esso viene applicata l'operazione di *reasoning*: i risultati dell'operazione vengono salvati, attraverso una struttura XML, all'interno della riposta e inviati allo strato di *presentation* dell'estensione.

In questo passaggio vengono mostrati, ancora una volta, i punti di forza dell'architettura e dell'implementazione del reasoner. Utilizzando delle classi che gestiscono le operazioni logiche per eseguire le

LISTING 6.7. *Controller* per i servizi di *reasoning*

```
Response getPreCheckedResponse(String request) {
//Create new response
Response response = null;

if(request.equals(startReasoning)){
        String cycleReasoning = setHttpPar(howManyTimesApllyInferenceRule);
        String idsRule = setHttpPar(whicRulesApply);
        String output = setHttpPar(produceOutput);
        return response = startReasoning(cycleReasoning,idsRule,output);
}
```

LISTING 6.8. Metodo *javascript* per l'esecuzione dell'operazione di *reasoning*

```
Response startReasoning(String cycleReasoning,String idsRule,String output) {
XMLResponseREPLY response;
//Retrieve inference rules file
File inferenceRuleFile = reasonerServiceConfiguration.getInferenceRuleFile();
Project<? extends RDFModel> project= null;

//Set parameters of reasoning operation
reasonerFacade.setParameter(ConfigurationParameter.NUMBER_OF_EXECUTION, cycleReasoning);
reasonerFacade.setParameter(ConfigurationParameter.PRODUCE_OUTPUT, output);
reasonerFacade.setParameter(ConfigurationParameter.WHICHRULEEXECUTE, idsRule);
//Get current project to apply resoning operation
project = ProjectManager.getCurrentProject();
....
baseRDFTripleModel = project.getOntModel();
//Apply reasoning on model
List<ARTStatement> new_triple =
        reasonerFacade.startReasoner(baseRDFTripleModel,inferenceRuleFile);
....
response = createReplyResponse(RepliesStatus.ok);
Element dataElement = response.getDataElement();
Element reasonerElem = XMLHelp.newElement(dataElement, "startReasoning");
//Populate response
provaElem.setAttribute("produceOutput",output);
provaElem.setAttribute("numberOfTriple", Integer.toString(new_triple.size()));
provaElem.setAttribute("printOutput", reasonerFacade.getOutputList().printOuput());
//return response
return response;
}
```

funzionalità del sistema permettono ai sistemi esterni che usano il *reasoner* di non occuparsi di come il sistema è stato implementato, ma dovranno solamente dipendere dall'interfaccia dei moduli che il *resoner* espone all'esterno: in questo modo una eventuale modifica all'interno *reasoner* non andrà ad impattare sull'estensione dei servizi di ST.

CAPITOLO 7

Conclusioni

Il *reasoner* descritto nei precedenti capitoli deve essere inteso come uno strumento che dia la possibilità, all'utente finale, di modellare il funzionamento di tale oggetto in base alle proprie esigenze. Questo strumento non è stato pensato per fornire un processo di *reasoning* il cui unico obiettivo consiste nel trovare il maggior numero di informazioni implicite all'interno di un'ontologia, ma per permettere all'utente di modificare ogni processo dell'operazione di *reasoning,* in modo tale da trovare solamente le informazioni di cui necessita. Per questo motivo, durante la progettazione del *software*, si è posta l'attenzione su come fornire un modello per la rappresentazione del maggior numero possibile di regole di inferenza: sarà poi compito dell'utente specificare le proprie regole di inferenza da usare all'interno dell'ontologia. Questo comportamento differisce totalmente dalla maggior parte dei *reasoner* che sono disponibili in commercio nei quali non solo l'utente è totalmente all'oscuro dei meccanismi messi in atto dal *reasoner* per scovare le informazioni implicite ma è anche impossibilitato nel decidere in quale modo il *reasoner* deve agire. Se da una parte i *reasoner* più comuni forniscono un servizio adito a trovare il maggior numero di informazioni nascoste all'interno dell'ontologia, dall'altra parte mancano totalmente di configurabilità e di personalizzazione. Naturalmente tutto ruota intorno alle esigenze dell'utente finale: un utente potrebbe essere interessato solamente a trovare tutte le relazioni dei dati all'interno dell'ontologia, mentre un altro è interessato a trovare solamente un particolare tipo di relazioni. Nel primo caso, l'utilizzo dei più comuni *reasoner* è più che sufficiente, nel secondo caso, invece, l'utente deve avere a disposizione un *reasoner* che dia la possibilità di essere configurato per raggiungere gli obiettivi imposti dall'utilizzatore di questo strumento. In conclusione, questo *reasoner* è stato pensato per quel bacino di utenti che hanno la necessità di creare delle proprie regole di inferenza e di visualizzare come è stato attuato il processo di *reasoning*. Per raggiungere questi obiettivi sono stati creati degli strumenti di personalizzazione che fossero il più *user-friendly* possibile in modo tale da rendere l'utilizzo del *reasoner* di facile intuizione: l'unica conoscenza richiesta è come devono essere scritte le regole di inferenza. Fornendo questi strumenti viene data la possibilità all'utente di controllare ogni singolo aspetto personalizzabile dell'operazione di *reasoning*. Questo *reasoner*, ad esempio, potrebbe essere usato come uno strumento didattico per gli studenti che si approcciano, per la prima volta, al mondo del *semantic web*. Poiché vengono mostrati tutti i procedimenti per l'esplicitazione delle nuove informazioni, gli utenti sono in grado di avere una maggiore comprensione del lavoro svolto dai *reasoners*. Infine, un altro aspetto che ha portato allo sviluppo del *reasoner* è stato quello di fornire all'editor di ontologie della piattaforma di *Semantic Turkey* l'opzione di attuare processi di *reasoning* all'interno delle ontologie caricate in quest'ultima. Combinando le potenzialità offerte da *Semantic Turkey* con i servizi messi a disposizione dal *reasoner* è possibile consegnare nelle mani dell'utente un *tool* per l'editor di ontologie che sia in grado di operare indipendentemente dal tipo di ontologia utilizzata e sul quale vi è l'opportunità di eseguire l'operazione di *reasoning* con le modalità descritte precedentemente.

Come è stato sottolineato nella sezione 3.2 la creazione di un architettura del *reasoner* che fosse il più modulare possibile, ha permesso di creare un terreno, su cui è costruito il *reasoner,* che sia adattabile a nuove modifiche. Ad esempio, se ci fosse la possibilità di trovare un modo più efficiente di eseguire le *query* di ricerca all'interno dell'ontologia (si ricordi il problema dell'esecuzione delle *query* descritto nella sezione 5.2), sarà possibile inserire questa nuova operazione all'interno del *reasoner* senza stravolgere l'intera architettura. Inoltre, grazie a questo tipo di architettura, si è assicurata la più totale versatilità del sistema potendolo integrare in futuri sistemi diversi. La vita del *reasoner* è strettamente legata ai futuri sviluppi della rappresentazione dei dati all'interno del *semantic web*. Poiché il *reasoner* abbraccia le tecnologie usate per la definizione delle ontologie definite dal *W3C*, necessariamente dipenderà dal loro futuro: qualora dovessero cambiare tali tecnologie il processo di *reasoning* dovrà essere riveduto e modificato opportunamente. Oltre ad essere dipendente dalle rappresentazioni delle ontologie, il *reasoner* dipende dalla logica dei processi di inferenza che vengono utilizzati per applicare le regole di inferenza. Anche in questo caso, eventuali scoperte di procedimenti logici che portano alla scoperta di informazioni in modo più rapido all'interno delle ontologie, costringeranno a rivedere e correggere l'intero processo di

applicazione delle regole di inferenza. In ogni caso, come è stato detto precedentemente, l'architettura del *reasoner* è stata concepita per accogliere tutte le eventuali modifiche che dovranno essere effettuate in prossimo futuro.

Appendici

Appendice A - Installazione del sistema

Il progetto generale consiste di due parti: una parte è relativa al *reasoner* e l'altra all'estensione di *Semantic Turkey.* Mentre la prima parte è pensata per funzionare come una applicazione *stand-alone*, la seconda necessita dell'installazione della piattaforma di *Semantic Turkey* per potere funzionare correttamente. Verranno prima illustrate le modalità con le quali è possibile eseguire il *reasoner* come un applicazione *stand-alone* e successivamente verrà analizzata l'integrazione dell'estensione del *reasoner* all'interno di *Semantic Turkey.*

Sorgenti *reasoner*

I sorgenti del *reasoner* si possono reperire al seguente link: sorgenti Reasoner. Una volta scaricata la cartella contenente il progetto è necessario eseguire una *build* attraverso l'utilizzo di *maven* (Maven build) posizionandosi nella *directory* in cui si è scaricato il progetto e dove è presente il file *pom.xml.* Successivamente aprire una console a riga di comando (*shell* in ambienti operativi *unix* o un *command prompt* in ambienti operativi *windows*). Una volta aperta la console si deve eseguire il comando:

mvn install /path/to/src/folder

dove */path/to/src/folder* indica il percorso dove è contenuto il file *pom.xml.* Una volta eseguito il comando, i sorgenti del progetto saranno compilati e sarà possibile importare il progetto all'interno dei vari *IDE* di sviluppo. Se non si è interessati ai sorgenti del progetto si può passare al paragrafo che illustra le modalità per avviare un operazione di *reasoning.*

Avvio del *reasoner*

Per eseguire il *reasoner* è necessario scaricare un file *.jar* disponibile al seguente link: Reasoner Jar. Una volta scaricato il link è necessario aprire una console a riga di comando (*shell* in ambienti operativi *unix* o un *command prompt* in ambienti operativi *windows*) e posizionarsi all'interno della *directory* in cui si è scaricato il file *.jar.* A questo punto, digitare nel terminale il seguente comando:

java -jar nome_file.jar

dove *nome_file.jar* è il nome si è dato al file scaricato. A questo punto si aprirà l'interfaccia grafica illustrata in figura 7.0.1 dalla quale è possibile configurare il *reasoner* per l'operazione di *reasoning.* È necessario, per far partire l'operazione di *reasoning,* specificare attraverso il menù *"Ontology"* (mostrato in figura 7.0.2) un'ontologia su cui eseguire l'operazione di *reasoning* (all'interno della cartella *inputFolder,* contenuta all'interno del progetto sono disponibili alcune ontologia di *default*) e una cartella di supporto per lo *storage* di informazioni che verranno raccolte durante l'operazione. All'interno del sistema è caricato un file di *default* delle regole di inferenza dal quale si possono scegliere quali regole devono essere eseguite. Nel caso in cui si vuole inserire un proprio file bisognerà utilizzare il *button "load inference rules".* Inoltre, all'interno della finestra in cui vengono visualizzate le regole di inferenza caricate, (raggiungibile con il *button "show rules"*) sarà possibile modificare tali regole ricordandosi di salvarle al termine della modifica. Le regole scelte dovranno essere inserite all'interno dell'apposita *text area* facendo attenzione ad inserire solamente l'ID (identificativo univoco) della regola di inferenza. Nel caso in cui devono essere inserite più regole, esse devono essere separate dal carattere ",". Se non viene inserita alcuna regola allora verranno applicate tutte le regole di inferenza specificate all'interno del file. Una volta scelti i parametri di configurazione sarà possibile eseguire l'operazione di *reasoning.* Terminata l'operazione verrà visualizzato l'output prodotto (se è stato scelto di produrlo) attraverso la visione testuale e/o attraverso la visione del grafo dell'output.

Nella visione testuale è possibile filtrare i risultati ottenuti in base alle regole di inferenza da cui sono stati generati: funzione che può essere eseguita attraverso l'apposito riquadro situato in alto a destra. Invece, nel riquadro in basso a destra, è possibile cercare come è stata generata una singola informazione. Supponiamo che l'informazione che si vuole cercare sia quella espressa nel listato 7.1. L'informazione dovrà essere inserita, all'interno delle *search box,* nella maniera mostrata in figura 7.0.3 Nella visione del

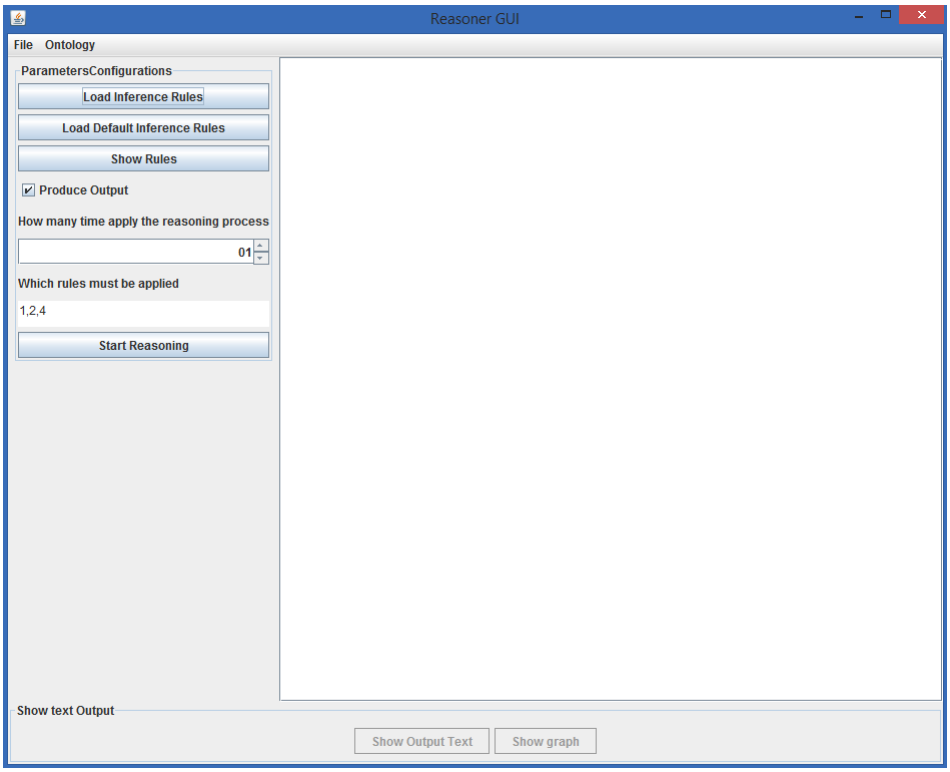


FIGURA 7.0.1. *Reasoner GUI*

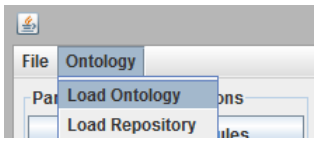


FIGURA 7.0.2. Caricamento di un ontologia e di una cartella di *repository*

LISTING 7.1. Esempio di una tripla esplicitata

```
############ Triple ###############
Rule type: new rule
Rule name: RegolaTransitiva Rule
ID: 1
Result:
http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#Roma,
http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#locatedIn ,
http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#Italia )
....
```

grafo è possibile ottenere una spiegazione delle funzionalità messe a disposizione di tale visualizzazione cliccando sull'apposito *help button.*

Sorgenti estensione *reasoner*

I sorgenti dell'estensione sono disponibili al seguente link sorgenti estensione. Analogamente per come è stato fatto con i sorgenti del progetto del *reasoner,* sarà necessario eseguire una *build* con *maven.* L'estensione di *Semantic Turkey* del *reasoner* necessita dei sorgenti compilati del progetto del reasoner, per questo motivo, prima di eseguire la *build* dell'estensione, bisognerà eseguire una *build* del progetto del *reasoner* (come indicato nella sezione 7). Una volta eseguita la *build* del progetto e la *build* dell'estensione

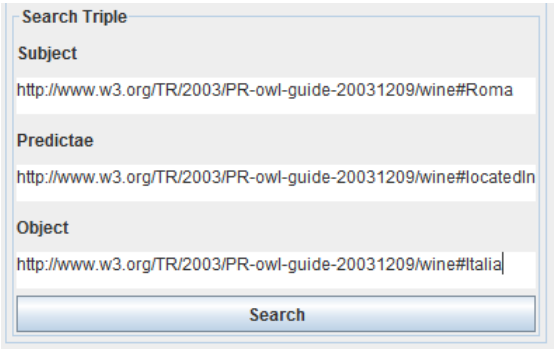


FIGURA 7.0.3. Esempio di ricerca di come è stata esplicitata un informazione

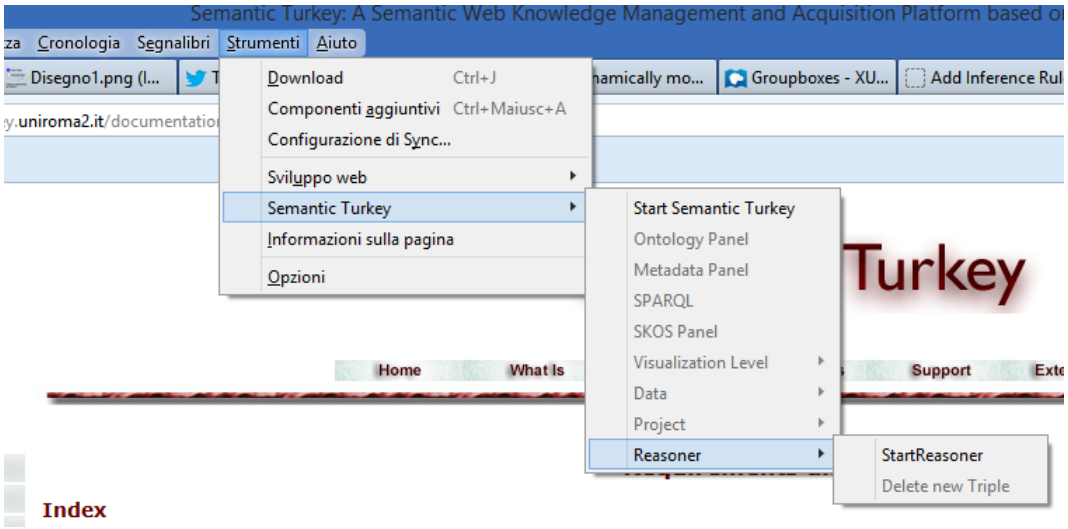


FIGURA 7.0.4. Menù contestuale dell'estensione del *reasoner*

sarà possibile integrare i progetti all'interno di un *IDE* di sviluppo e eseguire l'estensione all'interno di *Semantic Turkey.*

Avvio estensione all'interno di *Semantic Turkey*

Come abbiamo detto all'inizio dell'appendice, per potere utilizzare all'interno della piattaforma di *ST* l'estensione del *reasoner,* è necessario avviare la piattaforma di ST all'interno del proprio sistema. Per eseguire questa operazione bisognerà dapprima scaricare l'estensione di ST per *firefox* e successivamente avviare il server di ST con all'interno l'estensione del *reasoner.* Per una guida completa all'installazione di ST e per l'avvio della piattaforma fare riferimento alla seguente guida: ST installazione. Una volta installato ST bisognerà inserire l'estensione del reasoner all'interno del server di ST: per fare ciò è necessario copiare il file *reasoner-extension.jar*, che si trova nella cartella */target* situata nella cartella in cui si è eseguita la *build* del *reasoner,* all'interno della cartella */extensions/service* che si trova nella cartella in cui avete scaricato il server di ST. A questo punto rimane da installare l'estensione per *firefox* (nome del file *reasoner-extension.xpi*)d el *reasoner* che è contenuta nella stessa cartella in cui era contenuto il file *.jar* precedente. Una volta che il server e l'estensione di ST sono avviati è possibile iniziare ad operare con l'estensione del *reasoner* cliccando sul menù contestuale come descritto in figura 7.0.4. Cliccando sulla voce "*start reasoner*" si aprirà l'interfaccia grafica come mostrato in figura 7.0.5 dalla quale sarà possibile configurare i parametri per l'operazione di *reasoning.* Per avviare l'operazione di *reasoning* è necessario che l'utente abbia scelto almeno un progetto all'interno dell'ambiente di *Semantic Turkey* altrimenti verrà restituito un messaggio di errore. All'interno del sistema è caricato un file di *default* delle regole di inferenza dal quale si possono scegliere quali regole devono essere eseguite. Nel caso in cui si vuole inserire un proprio file sarà possibile farlo attraverso l'apposito *button "load inference rules".* Inoltre, all'interno della finestra in cui vengono visualizzate le regole di inferenza caricate (raggiungibile con il *button "show rules"*) è possibile modificare le regole di inferenza e aggiungerne di nuove attraverso

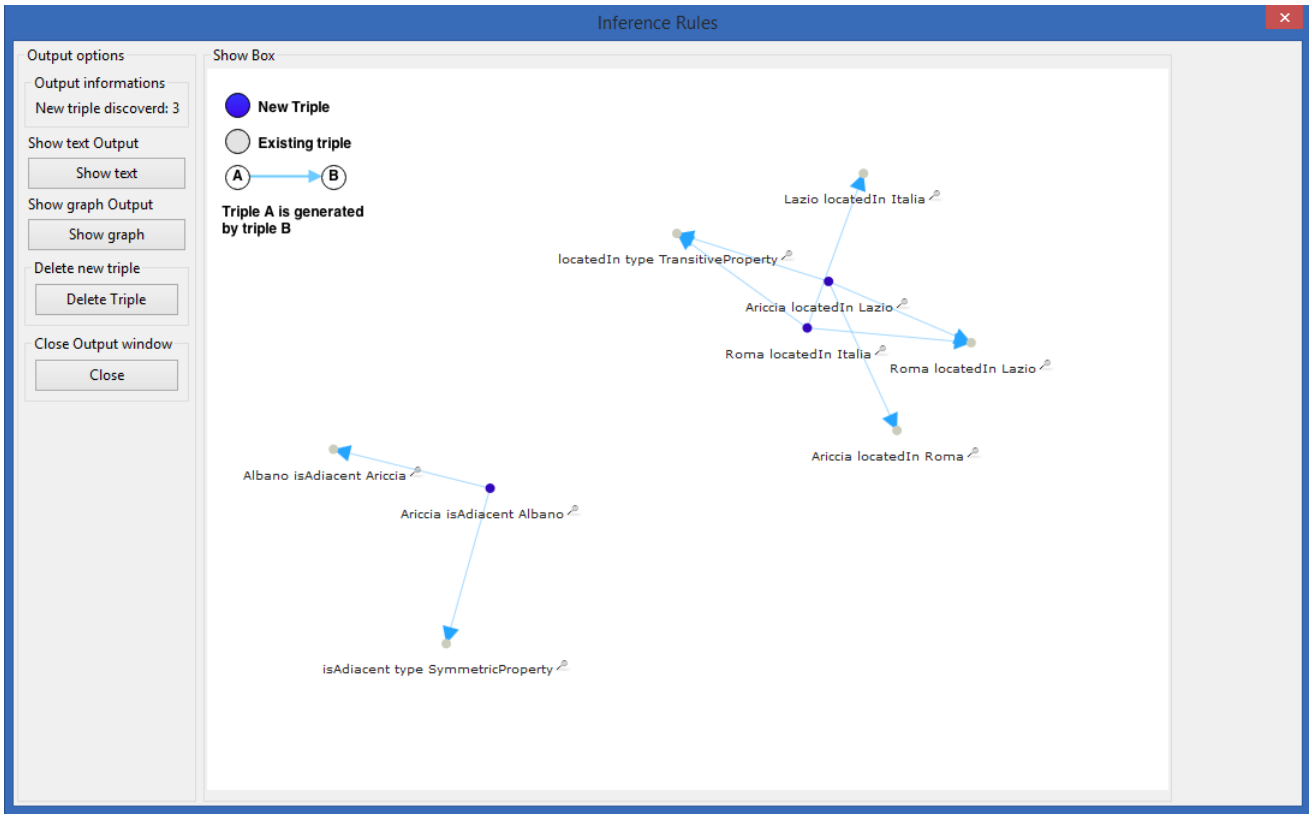


FIGURA 7.0.5. ST *GUI reasoner*

il modulo raggiungibile dall*'add rule button.* Al termine delle operazione di modifica è necessario cliccare sul *button upload* per salvare le modifiche nel sistema. Le regole scelte dovranno essere inserite all'interno dell'apposita *text area* facendo attenzione ad inserire solamente l'ID (identificativo univoco) della regola di inferenza. Nel caso in cui devono essere inserite più regole, esse devono essere separate dal carattere ",". Se non viene inserita alcuna regola allora verranno applicate tutte le regole di inferenza specificate all'interno del file. Una volta scelti i parametri di configurazione sarà possibile eseguire l'operazione di *reasoning.* Terminata l'operazione verrà visualizzato l'output prodotto (se è stato scelto di produrlo) attraverso la visione testuale e/o attraverso la visione del grafo dell'output.

Nella visione testuale è possibile filtrare i risultati ottenuti in base alle regole di inferenza da cui sono stati generate: funzione raggiungibile attraverso l'apposito riquadro situato in alto a destra. Invece, nel riquadro in basso a destra, è possibile cercare come è stata generata una singola informazione con gli stessi procedimenti descritti nella sezione 7.

Nella visione del grafo dell'output è possibile cercare come è stato generato un nodo cliccando sull'icona a forma di lente di ingrandimento. Invece, cliccando sul nome del nodo, verranno evidenziati gli archi che sono collegati al nodo selezionato.

Una volta che il processo di *reasoning* è stato completato si può scegliere se eliminare le nuove informazioni dal progetto corrente. Questa operazione è possibile sia attraverso l'uso dell'apposito menù contestuale "*delete new triple*" espresso in figura 7.0.4 (potrà essere utilizzato solamente dopo che l'operazione di *reasoning* è stata completata) sia attraverso il *button* "*delete triple*" posizionato nell'interfaccia di output del processo di *reasoning* (se è stato scelto di produrre l'output). Se viene scelto di non rimuovere le nuove triple, queste verranno salvate in modo permanete all'interno dell'ontologia caricata in *Semantic Turkey.*

Appendice B - Definizioni per la scrittura delle regole di inferenza

Le regole di inferenza che possono essere date in input al *reasoner* devono avere, in accordo con la definizione data nel capitolo 2, i seguenti elementi:

- Insieme di informazioni aggiuntive
- Una o più premesse
- Una o più conclusioni

Informazioni aggiuntive

Una regola di inferenza necessita di alcune informazioni aggiuntive che servono per identificare e catalogare tale regola durante il processo di *reasoning* e nella visualizzazione dell'output. Queste informazioni sono:

- Tipo di regola
- Nome della regola
- Identificativo della regola

Le regole di inferenza si suddividono in base a due tipi: "*new rule*" e "new inconsistency rule". Con il primo tipo si identificano tutte quelle regole di inferenza che hanno l'obiettivo di produrre una nuova informazione, mentre nel secondo tipo si identificano le regole che servono per scovare le inconsistenze all'interno dell'ontologia.

Ogni regola deve avere un nome che esplicita il comportamento della regola e deve avere un identificativo univoco: se esistono due o più regole con lo stesso identificativo, l'operazione di *reasoning* restituirà un messaggio di errore. Un esempio di queste informazioni aggiuntive è il seguente:

```
type : new rule
name: RegolaTransitiva
id: 1
```

dove ogni informazione aggiuntiva è preceduta da una *keyword* che deve essere necessariamente specificata per ogni regola di inferenza.

Premesse e conclusioni

Dopo la specifica delle informazioni aggiuntive è possibile definire l'insieme delle premesse e delle conclusioni. Sia le premesse che le conclusioni condividono la stessa struttura formata da un *subject,predicate* e *object.* Questi tre componenti possono assumere i seguenti valori:

- variabile
- *URI*
- *blank node*
- numero

Una variabile è formata da un "?" seguita dal nome della variabile (senza spazi), ad esempio:

```
?variabile
```

Un URI invece identifica una risorsa all'interno di un ontologia o all'interno di *RDFSchema* o più in generale all'interno dei *vocabolary* usati nei linguaggi per descrivere le ontologie, come *RDF* e OWL. Alcuni esempi di *URI* sono:

```
http://www.w3.org/1999/02/22-rdf-syntax-ns#type
http://www.w3.org/2002/07/owl#TransitiveProperty
http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#locatedIn
```

dove il primo e il secondo *URI* identificano delle risorse definite all'interno dei vocabolari *RDFS* e *OWL,* mentre il terzo *URI* identifica la risorsa definita all'interno dell'ontologia che ha come *namespace* "*http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine*". Le *URI* che vengono specificate all'interno delle regole di inferenza devono essere preceduta dal simbolo "<" e terminate con il simbolo ">". Le *URI* descritte precedentemente verranno quindi riscritte nel seguente modo:

<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2002/07/owl#TransitiveProperty>
<http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#locatedIn>

Un *blank node* rappresenta una risorsa contenuta all'interno di un'ontologia al quale non è associato nessun *URI* o un identificativo in generale. Un *blank node* è composto dall'elemento "_:" seguito da un nome senza spazi. Un esempio di *blank node* è:

_:blanknode

Infine, è possibile specificare come valore di un *subject* e di un *object* un numero che rispetta il seguente *pattern*:

"value"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>

dove *value* è il numero che si vuole esprimere. Un esempio è:

"10"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>

premise: subject: value predicate: value object:value

dove la *keyword premise* identifica l'inizio della definizione di una nuova premessa e dove *value* rappresenta il valore che viene dato ai componenti *subject,predicate* e *object.* Un esempio di regola di inferenza con tre premesse è:

```
type : new rule
name: RegolaTransitiva
id: 1
premise: subject: ?p predicate: <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
object: <http://www.w3.org/2002/07/owl#TransitiveProperty>
premise: subject: ?a predicate: ?p object: ?b
premise: subject: ?b predicate: ?p object: ?c
```

Una volta specificate le premesse e se ci troviamo in presenza di una regola di inferenza che è di tipo "*new rule*" è possibile specificare una o più conclusioni. Il *pattern* di specifica delle conclusioni è analogo a quello delle premesse cambia solamente per la *keyword* iniziale che consiste in "*conclusion:".* La regola di inferenza definita prima con l'aggiunta di una conclusione diventa

```
type : new rule
name: RegolaTransitiva
id: 1
premise: subject: ?p predicate: <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
object: <http://www.w3.org/2002/07/owl#TransitiveProperty>
premise: subject: ?a predicate: ?p object: ?b
premise: subject: ?b predicate: ?p object: ?c
conclusion: subject: ?a predicate: ?p object: ?c
```

Se ci troviamo in presenza di una regola che appartiene al tipo "*new inconsistency rule"* non è permesso specificare alcuna conclusione. Al termine della specifica delle premesse dovrà essere solamente aggiunta la seguente clausola:

conclusion: false

Un esempio di una regola di tipo "*new inconsistency rule"* è:

```
type : new inconsistency rule
name: ClassInconsistency
id: 12
premise: subject: ?x predicate: <http://www.w3.org/2002/07/owl#maxCardinality>
object: "0"^^<http://www.w3.org/2001/XMLSchema#nonNegativeInteger>
premise: subject: ?x predicate: <http://www.w3.org/2002/07/owl#onProperty> object: ?p
premise: subject: ?u predicate: <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
object: ?x
premise: subject: ?u predicate: ?p object: ?y
conclusion: false
```

Un file di esempio di regole di inferenze può essere visualizzato al seguente link Inference Rules file

Opzione *filter*

Dopo la specifica delle premesse è possibile applicare un filtro ai risultati ottenuti dall'applicazione delle premesse all'interno dell'ontologia. Il filtro deve essere scritto rispettando il seguente *pattern*:

```
filter: expression
```

dove la *keyword filter* identifica l'inizio della specifica di un nuovo filtro e *expression* identifica un espressione booleana. L'espressione è composta da almeno due predicati che possono essere le variabili specificate nella regola di inferenza legati da una condizione logica. Un esempio di espressione è:

```
filter: ?c1 != ?c2
```

inoltre è possibile creare espressioni *booleane* più complesse facendo attenzione al posizionamento delle parentesi, ad esempio:

```
filter: (?c1 != ?c2) && ((?a1 = ?b2) || (?a3 && ?b3))
```

Un esempio di regola di inferenza con una condizione di filtro è:

```
type : new rule
name: FilterExampleRule
id: 10
premise: subject: ?p predicate: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#locatedIn>
object: ?o
premise: subject: ?a predicate: <http://www.w3.org/TR/2003/PR-owl-guide-20031209/wine#isAdiacent>
object: ?b
filter: ?p != ?b c
onclusion: subject: ?a predicate: ?p object: ?o
```

Bibliografia

[1] http://www.alexa.com/siteinfo/google.com - Global Rank
[2] http://www.google.com/intl/en/insidesearch/howsearchworks/crawling-indexing.html
[3] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schutze, Introduction to Information Retrieval. *Introduction to Information Retrieval.* Cambridge University Press, New York, July 2008, p 5.
[4] Grigoris Antonius, Frank van Harmelen. *A Semantic Web Primer*. The MIT press, 2nd edition, USA, 2008, p 1.
[5] http://oald8.oxfordlearnersdictionaries.com/dictionary/inference
[6] Dean Allemang, Jim Hendler. *Semantic web for the working ontologist - Effective modeling in RDFS and Owl.* Morgan Kaufmann pubblications, Burlington, MA, 2008, p 11.
[7] http://www.w3.org/2001/sw/
[8] Hans Kamp, Uwe Reyle. *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory.* Kluwer Academic Publisher, Netherlands, 1993, pp 14-15.
[9] Ling Liu, M. Tamer Özsu (Eds.). *Encyclopedia of Database Systems.* Springer-Verlag, 2009, also http://tomgruber.org/writing/ontology-definition-2007.htm
[10] Grigoris Antonius, Frank van Harmelen. *A Semantic Web Primer*. The MIT press, 2nd edition, USA, 2008, p 1.
[11] https://jaxb.Java.net/
[12] A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman. *Compiler:principles,Techinques and Tool.* Pearson Education , Italy, 2008, 2nd edition.
[13] J.E.Hopcroft, R.Motwani, J.D.Ullman. *Introduction to Automata theory,Languages and Computation.* Pearson Education, Italy, 2008, third edition
[14] A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman. *Compiler:principles,Techinques and Tool.* Pearson Education , Italy, 2008, 2nd edition, p 56.
[15] http://www.w3.org/TR/rdf-sparql-query/
[16] http://jena.apache.org/documentation/inference/
[17] http://owlapi.sourceforge.net/reasoners.html
[18] Patrick Suppes. *Introduction to Logic.* Litton Educational pubblishing, London, 1957, pp 20-22.
[19] T. R. Gruber. *Toward principles for the design of ontologies used for knowledge sharing.* Presented at the Padua workshop on Formal Ontology, March 1993, later published in International Journal of Human-Computer Studies, Vol. 43, Issues 4-5, November 1995, pp. 907-928 also http://www-ksl.stanford.edu/kst/what-is-an-ontology.html.
[20] http://semanticweb.org/wiki/Ontology_language
[21] Grigoris Antonius, Frank van Harmelen. *A Semantic Web Primer*. The MIT press, 2nd edition, USA, 2008, p 67.
[22] http://www.w3.org/RDF/
[23] David Barnes, Micheal Kolling. *Object First with Java.* Pearson Education, USA, 2009, pp 205.
[24] J. Douglas Orton and Karl, WeickReviewed. *Loosely Coupled Systems: A Reconceptualization* from *The Academy of Management Review*, Vol. 15, No. 2 (Apr., 1990), Academy of Management, pp. 203-223
[25] Ian Sommerville. *Ingegneria del software.*Pearson Education, Italia, 2007, 8ava edizione, p 11.
[26] Luciano Baresi, Luigi Lavazza, Massimiliano Pianciamore. *Dall'idea al codice com UML 2.* Pearson Education, Italia, 2006, p 21.
[27] Luciano Baresi, Luigi Lavazza, Massimiliano Pianciamore. *Dall'idea al codice com UML 2.* Pearson Education, Italia, 2006, p 18.
[28] http://www.oracle.com/us/technologies/Java/overview/index.html
[29] http://docs.oracle.com/Javase/tutorial/essential/environment/properties.html
[30] The "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company, USA, 1998, p 144
[31] http://docs.oracle.com/Javase/6/docs/api/Java/util/ArrayList.html
[32] http://docs.oracle.com/Javase/7/docs/technotes/guides/swing/
[33] http://art.uniroma2.it/
[34] http://semanticturkey.uniroma2.it/
[35] http://art.uniroma2.it/owlart/
[36] https://developer.mozilla.org/it/docs/XUL
[37] http://www.Java.com/en/download/help/appsecuritydialogs.xml
[38] http://www.json.org/
[39] http://spring.io/
[40] http://www.osgi.org/Main/HomePage
[41] http://www.osgi.org/About/HomePage
[42] http://docs.spring.io/spring/docs/2.0.8/reference/mvc.html
[43] http://www.w3.org/TR/owl-guide/
[44] http://docs.oracle.com/Javase/1.5.0/docs/guide/management/jconsole.html

[45] http://www.oracle.com/technetwork/Java/Javase/tech/Javamanagement-140525.html
[46] Dean Allemang, Jim Hendler. *Semantic web for the working ontologist - Effective modeling in RDFS and Owl.* Morgan Kaufmann pubblications, Burlington, MA, 2008, p 66
[47] http://semanticturkey.uniroma2.it/documentation/developers.jsf
[48] J.E.Hopcroft, R.Motwani, J.D.Ullman. *Introduction to Automata theory,Languages and Computation.* Pearson Education, Italy, 2008, third edition, pp 104.
[49] A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman. *Compiler:principles,Techinques and Tool.* Pearson Education , Italy, 2008, 2nd edition, pp 5.
[50] A.V.Aho, M.S.Lam, R.Sethi, J.D.Ullman. *Compiler:principles,Techinques and Tool.* Pearson Education , Italy, 2008, 2nd edition, pp 9.