ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SECONDA FACOLTÁ DI INGEGNERIA - SEDE DI CESENA
Corso di Laurea in Ingegneria Informatica

Architetture software per la coordinazione semantica: efficienza ed espressività

Tesi di Laurea in Sistemi Multi-Agente

Relatore:
Prof. Ing. Andrea Omicini

Presentata da:
Marco Savoia

Correlatore:
Ing. Davide Sottara

Sessione III
Anno Accademico 2010-2011

It is change, continuing change, inevitable change, that is the dominant factor in society today. No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be.

Isaac Asimov, 1978

Indice

1 Introduzione **1**
1.1 Motivazioni 2
1.2 Outline 2

2 Centri di tuple **5**
2.1 Linda e gli spazi di tuple 5
2.1.1 Comunicazione generativa 5
2.1.2 Caratteristiche e proprietà 7
2.2 Verso i centri di Tuple 10
2.3 TuCSoN & ReSpecT 11

3 Rappresentazione semantica **15**
3.1 Logica descrittiva 15
3.1.1 Descrizione dei formalismi 16
3.1.2 Modello 17
3.1.3 OWA vs CWA 22
3.1.4 Il linguaggio attributivo 23
3.1.5 Relazioni con altre logiche 25
3.1.6 Inferenza 27
3.1.7 Estensioni 29
3.2 Semantic Web 30
3.2.1 Architettura e strumenti 32
3.2.2 RDF 37
3.2.3 OWL 43

4 Centri di tuple semantici **53**
4.1 Verso i centri di tuple semantici 54
4.1.1 Le caratteristiche del sistema 55
4.2 Il modello 55
4.2.1 Ontologia e TBox 56
4.2.2 Tuple semantiche 58
4.2.3 Template semantici 59
4.2.4 Primitive semantiche 59
4.2.5 Reazioni semantiche 60
4.2.6 Alcuni altri aspetti 60
4.3 Progettazione e tecnologia 63
4.3.1 Il matching 64
4.3.2 SPARQL in TuCSoN 65

5 Fuzziness **67**
5.1 Introduzione alla fuzziness 67
5.2 Fuzzy DL 74
5.3 Fuzziness nei centri di tuple 75

6 Architettura **77**
6.1 Lo stato dell'arte: pregi e difetti 77
6.2 Architetture ibride 78
6.2.1 *Cohesion* e *coupling* 79
6.2.2 Modelli differenti 80
6.2.3 Architetture semantiche ibride 85
6.2.4 Sistemi esistenti 89
6.2.5 Verso la fuzziness 90
6.3 Verso una implementazione omogenea? 91
6.3.1 Tableau su un sistema a regole 91
6.3.2 Centri di tuple e ragionatori DL 92

7 Conclusioni **103**

Elenco delle figure

2.1 Comunicazione generativa. 6
2.2 Disaccoppiamento spaziale (distributed naming). 9
2.3 Schema semplificato dell'infrastruttura TuCSoN. 12

3.1 Architettura di una Description Logic. 18
3.2 Esempio di TBox. 19
3.3 Espansione parziale della TBox di figura 3.2. 20
3.4 Esempio di ABox 21
3.5 ABox mitologica $\mathcal{A}_{mit}$ 22
3.6 Semantic Web Cake 33
3.7 Grafo della proposizione. 39
3.8 Grafo delle proposizioni. 39
3.9 Grafico della proposizione. 40

4.1 Rappresentazione delle tuple in stile RDF 62
4.2 Architettura di TuCSoN 63
4.3 Architettura di un nodo TuCSoN 64

5.1 Unione e intersezione nei fuzzy set. 71
5.2 Esempio di di fuzzy set sovrapposti. 72
5.3 Esempio di di fuzzy set sovrapposti. 72

6.1 Moduli loosely coupled, con working memory disgiunte. . . 81
6.2 Moduli tightly coupled, con working memory condivisa. . . 83
6.3 Moduli coupled, con working memory separate. 84
6.4 Moduli coupled, con working memory condivisa. 85
6.5 Architettura di Pellet. 88

6.6 Architettura del sistema ibrido. 90
6.7 Architettura di Pellet (2). 92

Elenco delle tabelle

3.1 Costruttori elementari 17

6.1 Coupling metric. 80

Elenco dei sorgenti

4.1 SPARQL query 66
6.1 Interfaccia IOntology.java 93
6.2 Classe PelletOntology.java 94
6.3 Classe PelletReasonerUtils - metodo createIndividual 95
6.4 Classe PelletReasonerUtils - metodo incrementalCheckABox-Consistency 95
6.5 Classe SparqlQueryGeneratorImpl.java 96
6.6 Classe SPARQLStringBuildVisitor - metodo visit 97
6.7 Interfaccia Graph.java 98
6.8 Classe TripleAdder.java 99
6.9 Interfaccia Resource.java (stralci) 100
6.10 Classe Triple.java (campi e costruttore) 100
6.11 Classe Individual.java - metodo getRNeighborEdges 101

Capitolo 1

Introduzione

Sempre più nella progettazione dei sistemi software si rivela determinante il problema della coordinazione. Questo aspetto non banale è accentuato anche dalla maggiore *apertura* e *distribuzione* dei sistemi moderni; chiaramente la costruzione di un sistema il quale preveda di potersi interfacciare con una molteplicità di agenti esterni (eterogenei e dinamici) e al contempo essere composto di più componenti dislocati su nodi fisicamente disgiunti rappresenta una sfida complessa.

Partendo proprio dal problema centrale della coordinazione - e non dimenticando mai il contesto aperto e distribuito che vuole essere il nostro riferimento imprescindibile - non si può non considerare uno dei modelli che meglio sono riusciti ad affrontare e risolvere tale questione: gli *spazi di tuple*. Dal 1985 in avanti, a seguito dell'articolo [21] di David Gelernter su *Linda* si sono succeduti innumerevoli tentativi di estendere e migliorare quello strumento. In questo senso risulta piuttosto cruciale il passaggio dagli spazi di tuple ai *centri di tuple*, che hanno proprio nel nuovo modo di gestire gli eventi e la coordinazione il loro punto di forza.

Del tutto parallelamente a questo tipo di ricerche, nasce e si sviluppa un nuovo filone di studio che è quello del web semantico; questi lavori si pongono come obiettivo quello di trovare un'attuazione pratica a quanto si era studiato in merito alla *rappresentazione della conoscenza*, dalle logiche descrittive alle varie forme di strutturazione di una ontologia.

Unendo questi due ambiti di ricerca, nascono i *centri di tuple semantici*,

strumenti in grado di fondere le grandi capacità di coordinazione dei centri di tuple distribuiti con la gestione di una base di conoscenza e dei vari elementi che la compongono.

1.1 Motivazioni

A partire da quanto scritto fin qui si è ritenuto interessante prendere in esame TuCSoN. Dopo gli ottimi lavori[1] svolti per aggiungere a questo sistema una parte di riconoscimento semantico è opportuno fermarsi e riflettere sul lavoro svolto al fine di capire come poi proseguirlo al meglio. Così, questa tesi ha lo scopo di mettere in luce le caratteristiche di TuCSoN evidenziandone i pregi, tanto quanto gli eventuali difetti o carenze strutturali; poi, in relazione a queste ultime, si analizzeranno le possibili alternative e soluzioni con particolare interesse per l'assetto architetturale del modello TuCSoN, allo scopo di mantenere intatta (o possibilmente anche incrementare) l'espressività e potenziare l'efficienza.

1.2 Outline

La tesi si sviluppa come segue. Nel capitolo 2 si ripercorre il percorso che ha portato da gli spazi di tuple e dal modello Linda verso i centri di tuple, mettendone in evidenza le differenze e le caratteristiche. Nel terzo capitolo viene illustrato il contesto della rappresentazione semantica, dalle logiche descrittive al web semantico con le relative tecnologie. Nel capitolo 4 vengono introdotti i centri di tuple semantici e in particolare si fa riferimento a TuCSoN, mettendo in luce non solo il suo modello nel complesso, ma anche tutte quelle che sono state le scelte progettuali di rilievo. Nel capitolo 5 ci si addentra brevemente nell'ambito della *fuzziness*, provando a capire le basi matematiche su cui poggia questa branca complessa e cercando poi di orientare il discorso verso una possibile integrazione coi centri di tuple. Infine nel sesto capitolo, dopo una breve dissertazione

[1] Lavori documentati in [35, 44, 38].

sui possibili modelli architetturali e relativi pregi e difetti, viene presa in esame la possibilità di volgere verso una implementazione più omogenea dell'attuale TuCSoN, valutando da dove prendano origine gli attuali costi computazionali e le eventuali problematiche annesse.

Capitolo 2

Centri di tuple

2.1 Linda e gli spazi di tuple

Nel gennaio del 1985 David Gelernter in [21] illustra le ragioni e i vantaggi di Linda, un modello in grado di garantire disaccoppiamento tra i suoi vari componenti. In un'epoca in cui le reti di computer venivano definite *still purely experimental* egli fu in grado di capire l'importanza e le esigenze della computazione distribuita e di sviluppare in quest'ottica un sistema in grado di garantire nativamente coordinazione e comunicazione tra le proprie componenti.

2.1.1 Comunicazione generativa

Quando viene introdotta la comunicazione generativa in [21] (e poi in seguito in [22]), Gelernter specifica come questa sia sostanzialmente differente rispetto ai modelli di programmazione concorrente allora conosciuti, come i monitor, le operazioni in remoto e il passaggio di messaggi. In un periodo in cui si era ancora lontani dal concepire componenti hardware come i moderni multiprocessore o l'infrastruttura di rete attualmente disponibile, aveva certamente senso interrogarsi sull'utilità della comunicazione tra processi. Tuttavia gli spazi di tuple hanno saputo dare una risposta sia al problema della coordinazione sia a quello della distribuzione su più nodi delle varie unità di calcolo.

Gli *spazi di tuple* sono luoghi virtuali distribuiti che contengono tuple, ovvero frammenti ordinati e strutturati di informazione, come codice eseguibile o dati meramente passivi. Si suppone che più processi distinti agiscano concorrentemente sullo spazio di tuple e che vogliano comunicare tra loro: questo avviene mediante l'inserimento di una o più tuple. Il processo A inserisce una tupla e il processo B la ritira.

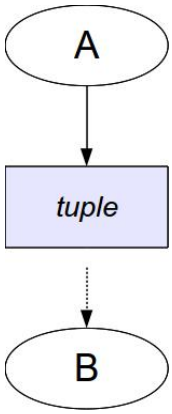


Figura 2.1: Comunicazione generativa.

In particolare questo sistema di comunicazione è detto *generativo* in quanto, finché non venga esplicitamente ritirata, la tupla generata da A persiste in maniera indipendente all'interno dello spazio di Tuple. Inoltre tale elemento è accessibile a tutti i processi che interagiscono con in *tuple space* e al contempo non è vincolato in alcun modo a nessuno di essi.

Primitive

Linda fornisce tre primitive di base: *out()*, *in()*, *read()*. Esse garantiscono rispettivamente di poter inserire, togliere e leggere tuple nel sistema. A ogni tupla viene associato un nome e una serie di variabili o costanti che ne costituiranno il corpo.

La **out()** assume la forma

```
out(N, P1, ...  ,Pj)
```

dove <P1 ... Pj>è una lista di parametri che possono essere attuali o formali (cioè indefiniti) e N è un parametro attuale che indica il nome. Da

notare che il processo che la sta eseguendo non si blocca, ma dopo questa chiamata procede oltre.

La **in()** appare come

```
in(N, P1, ...  ,Pj)
```

dove <P1 ... Pj>è una lista di parametri che possono essere attuali o formali e N è un parametro attuale che indica il nome. Nel caso in cui i parametri P siano variabili e nel TS[1] esista una tupla con quel nome, allora tale tupla viene ritirata dal TS e i suoi valori vengono assegnati alle variabili. La chiamata è bloccante, quindi nel caso in cui l'unificazione sopra descritta non avvenga, allora il processo non avanzerà oltre.

Infine la **read()**

```
read(N, P1, ...  ,Pj)
```

che agisce come la in() a parte per il fatto che la tupla, se individuata non viene rimossa dal TS.

2.1.2 Caratteristiche e proprietà

Distribuzione dei nomi

Sia `+t` a una tupla aggiunta al TS mediante una chiamata `out(+t)` e `-t` una lista di parametri di una chiamata `in(-t)` o `read(-t)`. Ebbene, il contenuto della tupla `-t` può anche non essere interamente composto da variabili, ma anche da valori. Dunque si potrà avere

```
read(P, i:integer, j:boolean)
```

ma anche

```
read(P, 7, j:boolean)
```

oppure

```
read(P, i:integer, FALSE).
```

[1]Tuple Space

Tutti i parametri espressi in `-t` devono corrispondere a termini presenti in `+t` affinché vi sia *unificazione* tra le due. Si può affermare che questo tipo di accesso associativo ricordi molto da vicino l'operazione SELECT nell'ambito dei database relazionali o altresì il *pattern-matching* caratteristico da alcuni linguaggi logici come il Prolog. Tuttavia una differenza sostanziale può essere individuata nel fatto che in Linda vi sia totale assenza di determinismo. In altre parole, se dovesse essere eseguita una operazione di `in(-t)` su uno Spazio di Tuple ove siano presenti più tuple in grado di unificare con `-t`, non si potrebbe identificare a priori quella che verrà rimossa. Analogamente, se molte `in(-t)` fossero in attesa della `+t` "unificante", qualora questa venga immessa in TS non si saprà quale processo si sarà sbloccato fino a che questo non sarà avvenuto. E in effetti questo modello di comportamento influenza grandemente proprio i processi che effettuano le chiamate su TS, con una attenzione particolare alla concorrenza tra essi: la concorrenza viene proprio gestita mediante la semantica sospensiva delle chiamate in() e read()[2].

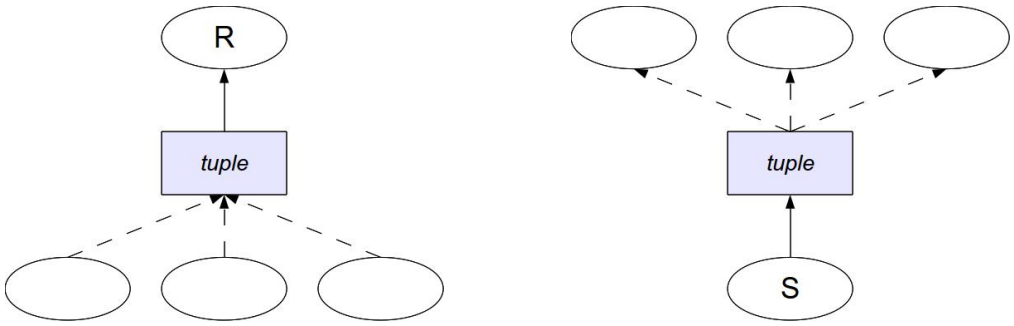
Ortogonalità della comunicazione

Mentre nella maggior parte dei modelli di comunicazione in ambito distribuito colui che riceve l'informazione non ha conoscenza in merito al mittente, al contrario il mittente deve citare esplicitamente (e quindi conoscere) il destinatario della comunicazione in questione. Tuttavia in Linda questo non avviene: a nessuno dei due (o più) attori in gioco viene richiesto di esplicitare l'altro (o gli altri). Il mittente si limita a lasciare la tupla nel TS e il ricevente (che a questo punto risulterebbe inesatto chiamare *destinatario*) si limita ad accedervi con le modalità messe in luce in precedenza. Da tutto ciò conseguono altre due proprietà, ovvero il disaccoppiamento spaziale e il disaccoppiamento temporale, che a loro volta danno origine alla cosiddetta *distributed sharing*.

[2]In seguito, in sistemi analoghi a Linda sono state prodotte estensioni di questo modello comunicativo come le primitive di bulk(`in_all()` e `read_all()`) o le primitive predicative(`inp()` e `readp()`) che hanno ampliato il numero di possibili scenari nell'ambito della coordinazione

Disaccoppiamento spaziale

Lo *space uncoupling* o *distributed naming* si riferisce al fatto che in Linda una tupla può essere immessa nello Spazio di Tuple da processi allocati in punti (nodi) differenti. Tuttavia mentre molti sistemi distribuiti consentono di *ricevere* messaggi da chiunque, Linda consente anche di *inviare* a tutti coloro che accedono al TS. É in questo senso che questo modello si definisce "completamente distribuito nello spazio".



(a) La tupla può essere immessa da qualunque processo.

(b) La tupla può essere rimossa da qualunque processo.

Figura 2.2: Disaccoppiamento spaziale (distributed naming).

Disaccoppiamento temporale

Per rimuovere una tupla dallo Spazio di Tuple è necessario, come detto, che un processo invochi una `in(-t)` (laddove `-t` "fa match" col template `+t` della tupla in questione). In attesa che questo avvenga la tupla permane nel TS. Questo avviene potenzialmente per sempre. Così, mentre nei sistemi distribuiti tradizionali due processi per trasmettersi informazioni dovono possedere il non banale requisito di essere attivi nel momento dell'atto comunicativo, ora questa condizione non è più necessaria. Infatti in Linda è perfettamente naturale concepire comunicazione distribuita nel tempo, ovvero anche tra processi che sono attivi in finestre temporali prive di intersezione.

Distributed sharing

Gelrnter in [21] fa infine notare come le due proprietà sopra descritte diano vita a una terza proprietà, che è appunto la Distributed sharing. In sostanza il mantenimento delle variabili non avviene, a differenza di altri contesti, mediante l'istanza di un processo o un modulo delegato a questo compito; questo servizio viene fornito nativamente dal Tuple Space, il quale ha in questa peculiarità un notevole punto di forza.

2.2 Verso i centri di Tuple

Lavorare in sistemi *information driven* significa che gli agenti si coordinano, competono e cooperano per accedere, produrre e consumare informazione [42]. Gli Spazi di Tuple sono ambienti sono un ottimo esempio di semplicità ed espressività, tuttavia hanno il problema non consentire una distinzione efficace tra l'informazione introdotta dalle tuple e la rappresentazione di tale informazione in relazione agli agenti che vi accedono. Il noto esempio dei *dining philosophers* [13] pone alla luce la dicotomia tra la rappresentazione naturale del dominio e una rappresentazione meno coerente ma più efficace ai fini del funzionamento del sistema. Inoltre il medium di coordinazione mantiene sempre lo stesso comportamento senza possibilità di adattarsi a specifiche casistiche. Talvolta addirittura i meccanismi di basi forniti dal TS non sono sufficienti a garantire le necessarie politiche di coordinazione. Così il peso della coordinazione viene scaricato sugli agenti stessi, andando a determinare un contesto poco pulito dal punto di vista concettuale, con entità necessariamente consce del problema della coordinazione, e nella cui implementazione si andranno a fondere gli aspetti di sincronizzazione con quelli di mera programmazione.

Tornando al pranzo dei filosofi, una soluzione potrebbe essere quella di inserire nel TS altre tuple che abbiano la funzione esclusiva di coordinazione tra gli agenti. Questa soluzione però non è troppo elegante, perché incrementa il peso della comunicazione e poi perché fa ricadere ancora una

volta la coordinazione sui singoli processi. Allora una soluzione presentata in [42] è quella di favorire la creazione di un livello di coordinazione che sia trasparente agli agenti. In sostanza i singoli processi non vedranno mutare l'interfaccia di accesso al TS e non muteranno neanche le singole azioni elementari tuttavia sarà possibile arricchire il comportamento del modello in termini di transizioni di stato messe in atto in risposta a una serie di eventi determinati. Questa è l'idea alla base dei *tuple centre*: una maggiore customizzazione a fronte di una invariata semplicità di accesso. Così, dietro transizioni di stato che verranno percepite dagli agenti come "single-step" si nasconderanno semantiche tanto complesse quanto lo richiederà lo specifico contesto applicativo. Alla fine si può concludere che i Centri di Tuple rimangano tendenzialmente un medium di coordinazione *data-driven*, ma che tuttavia posseggano anche caratteristiche tipiche dei modelli *control-driven* come la totale osservabilità degli eventi comunicativi o l'abilità di reagire in maniera selettiva agli stessi.

2.3 TuCSoN & ReSpecT

La parola TuCSoN [43, 2, 46] è acronimo di *TUple Centres Spread Over the Network* ed è una infrastruttura sviluppata per la comunicazione e la coordinazione tra agenti distribuiti e più in generale tra MAS (*Multi-Agents Systems*). Ciò avviene per mezzo dei centri di tuple - illustrati nella sezione 2.2 - e pertanto TuCSoN è un sistema reattivo e distribuito su più nodi. Esso è un'estensione di Linda e come tale fornisce primitive del tutto analoghe, ma, come detto, riprende anche il concetto di Tuple Centre e pertanto possiede due caratteristiche importanti in aggiunta al modello di Gelernter: la molteplicità degli spazi e la programmabilità.

Molteplicità

Il primo aspetto prevede di mettere a disposizione degli agenti più di uno spazio di tuple al fine di evitare problemi di concorrenza e più in generale di esecuzione. L'idea è quella di assegnare ad ogni agente uno

spazio (o più) per limitare al massimo il carico computazionale su un dato nodo. Da ciò consegue che gli agenti in questione abbiano una percezione duplice del del luogo di interazione: una globale, costituita dall'unico medium di coordinazione collettivo, l'altra locale relativa allo spazio a lui assegnato su un determinato nodo.

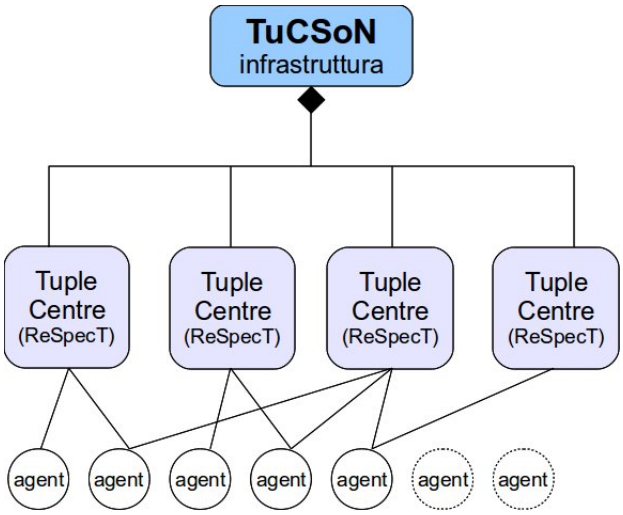


Figura 2.3: Schema semplificato dell'infrastruttura TuCSoN.

La sintassi di una operazione sul centro di tuple risulta pertanto la seguente:

```
TCName @ TCAddress ?  op(Tuple)
```

dove

`TCName` è il nome del centro di tuple;

`TCAddress` è l'indirizzo fisico del nodo su cui si trova il centro di tuple;

`op` rappresenta l'operazione da svolgere;

`Tuple` è il contenuto informativo coinvolto nell'operazione.

Programmabilità

Questo secondo aspetto di TuCSoN indica la possibilità di programmare il centro di tuple TuCSoN in maniera da poter ottenere uno specifico

comportamento in risposta all'esecuzione di primitive. Per fare ciò viene utilizzato ReSpecT (ovvero *REaction SPECification Tuples*) [1, 40, 41], che è un linguaggio logico di coordinazione atto a definire il comportamento di un centro di tuple mediante la gestione di (re)azioni a determinati eventi. Più nel dettaglio, a ReSpecT viene affidato un duplice ruolo:

Linguaggio di specifica ReSpecT filtra gli eventi di comunicazione e associa loro delle reazioni;

Linguaggio "reattivo" ReSpecT fornisce supporto per la creazione e la gestione di *reazioni* eseguite localmente all'interno del centro di tuple, laddove una reazione è un'insieme di operazioni elementari non bloccanti eseguite come fossero una unica operazione atomica.

Per capire un po' meglio questo ultimo aspetto vediamo come è strutturata la sintassi del linguaggio ReSpecT. Il componente base è la *reaction*, che esprime concettualmente quanto esposto qualche riga sopra ed è rappresentata come tupla logica nella forma:

```
reaction( Event, Guard, Body)
```

`Event` indica l'operazione a cui reagire;

`Guard` è una serie di predicati in merito alle proprietà dell'evento. Viene richiesto che la guardia sia verificata per attivare la reazione;

`Body` indica le primitive da svolgere quando viene attivata la reazione.

Capitolo 3

Rappresentazione semantica

3.1 Logica descrittiva

Lo scopo delle Logiche Descrittive è quello di rappresentare la conoscenza di uno specifico dominio applicativo, definendo da prima i concetti di rilievo di tale dominio (ovvero la completa gerarchia terminologica) e specificando in seguito le loro proprietà. Il tutto al fine di consentire ad applicativi ad hoc di ricavare conoscenza esplicita o implicita. A tale scopo, si potrebbe pensare, potrebbero essere utilizzate le logiche del primo ordine. In effetti queste ultime, sebbene si focalizzino più sulle frasi che non sulla descrizione dei termini, rappresentano spesso un formalismo sufficientemente espressivo. Tuttavia non sono di aiuto nel ragionare su categorie complesse e hanno un grosso limite nelle prestazioni: complessità e indecidibilità sono scogli di non poco conto! E dopo aver accantonato le logiche del primo ordine, si è costretti ad accantonare anche strumenti come le *semantic network* o *Frames* per l'assenza di semantiche formali di natura logica. E proprio quest'ultimo punto rappresenta la prima grande evoluzione delle Description Logics rispetto ai loro predecessori. Inoltre esse hanno come scopo centrale quello di favorire un ragionamento sulla conoscenza rappresentata. In pratica viene agevolato in processo di inferenza: mediante il supporto di specifici pattern, si possono inferire fatti e relazioni implicite a partire da quelle esplicite. Questo avviene anche grazie a meccanismi di descrizione della realtà e analogamente di inferenza

su di essa che ricalcano quelli della mente umana. In buona sostanza le Logiche Descrittive costituiscono un buon compromesso tra efficienza computazionale e potere espressivo, senza dimenticare che però spetta sempre al progettista il compito di scegliere una logica più espressiva, innalzando la complessità computazionale e rischiando di cadere nell'indecidibilità, piuttosto che non una logica più semplice, perdendo però la capacità rappresentare adeguatamente le realtà più complesse.

La prima rappresentazione della conoscenza elaborata mediante l'utilizzo di una Logica Descrittiva fu fatta nell'ambito del progetto KL-ONE del 1985 [9]. In questo sistema erano già presenti tre punti cardinali, che vennero poi mantenuti in tutti gli sviluppi successivi:

- la struttura sintattica basata su concetti, ruoli e individui. (Argomento approfondito nel prossimo capitolo)
- un insieme relativamente ristretto di costruttori per definire i domini più complessi.
- l'inferenza quasi automatica della conoscenza implicita.

Successivamente, negli anni Novanta la capacità di ragionamento di questi sistemi venne migliorata; l'introduzione di algoritmi a tableau e altre tecniche di ottimizzazione resero efficiente anche il calcolo in caso di complessità non polinomiale.

Attualmente le DL trovano impiego nei campi più svariati, dalle basi di dati alla bioinformatica, dall'ingegneria del software alla definizione di formalismi utili per le ontologie. In effetti lo stesso OWL può essere visto come una variante sintattica delle Description Logics.

3.1.1 Descrizione dei formalismi

Un linguaggio descrittivo si compone di tre semplici elementi: i concetti atomici (che possiamo vedere come classi o categorie), i ruoli (che rappresentano relazioni binarie) e gli individui o oggetti (che sono le singole istanze dei concetti). Si potrebbe dire che un linguaggio descrittivo

si compone di una terna $\langle \mathcal{C}, \mathcal{R}, \mathcal{O}b \rangle$ dove gli elementi di $\mathcal{C}$ sono indicati con le lettere maiuscole e rappresentano i concetti, gli elementi di $\mathcal{R}$ con le minuscole rappresentano i ruoli e con $\mathcal{O}b$ sono indicati i nomi dei ruoli. Partendo dalla realtà che ci circonda, l'esempio più classico di ruolo è "padre" o "madre" o viceversa "figlio"[1]. Esempi di concetti invece possono essere invece professore, studente.

I costruttori

I costruttori non sono altro che operandi, che, posti tra due concetti o due ruoli, hanno il compito di andare a formare concetti o ruoli più complessi. La maggior parte di essi deriva da costruttori di logiche del primo ordine, altri invece vengono definiti ad hoc. A seguire una lista dei più comunemente utilizzati.

Simbolo	Significato
$\top$	Tutti i concetti
$\bot$	Nessun concetto
$\sqcap$	Congiunzione
$\sqcup$	Unione
$\neg$	Negazione
$\forall$	Restrizione universale
$\exists$	Restrizione esistenziale
$\sqsubseteq$	Inclusione
$\equiv$	Equivalenza

Tabella 3.1: Costruttori elementari

3.1.2 Modello

I due componenti chiave di una Description Logic sono TBox e Abox. Tbox introduce la terminologia, ovvero il vocabolario del dominio applicativo, mentre Abox, rispettando il vocabolario, compie associazioni su

[1] E' da notare come questi ruoli possano generare ambiguità, a partire dal linguaggio naturale. Infatti *padre* può essere considerato anche un concetto, oltre che un ruolo; nell'accezione "A è padre di B" esso ricopre un ruolo, ma potrebbe essere visto semplicemente come l'insieme degli uomini che hanno almeno un figlio!

singole istanze. Si può asserire che la differenza tra i due è che TBox lavora su ruoli e concetti, mentre ABox lavora sugli individui[2].

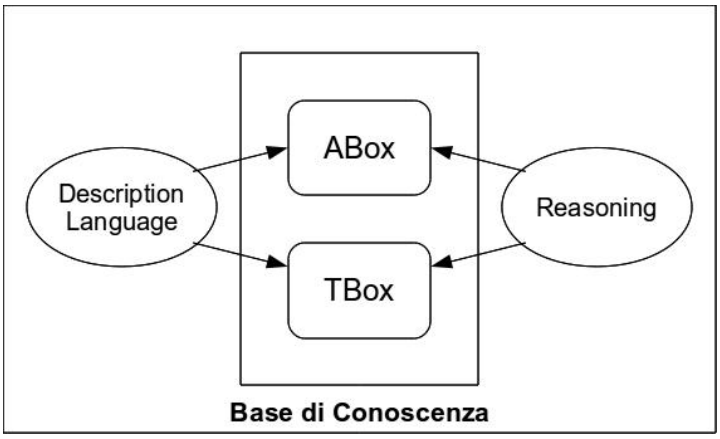


Figura 3.1: Architettura di una Description Logic.

TBox

La TBox è un insieme finito di frasi, assiomi o definizioni che descrivono i concetti. Una definizione è una uguaglianza tra un concetto atomico e un concetto arbitrario (non necessariamente esprimibile con un solo concetto). Ad esempio con la frase

$$\texttt{Mother} \equiv \texttt{Woman} \sqcap \exists \texttt{hasChild.Person}$$

si intende descrivere il concetto di "madre" come colei che è donna e ha almeno un figlio (ovvero esiste almeno una persona legata ha lei col ruolo di figlio). Continuando ad analizzare il caso molto semplice e intuitivo della struttura familiare si può provare a rappresentarla mediante definizioni di Logica Descrittiva elementare descritte nel capitolo secondo di [4].

[2]Nel web semantico questa differenza non esiste, in genere si ha una combinazione di entrmabi.

$$\texttt{Woman} \equiv \texttt{Person} \sqcap \texttt{Female}$$
$$\texttt{Man} \equiv \texttt{Person} \sqcap \neg\texttt{Woman}$$
$$\texttt{Mother} \equiv \texttt{Woman} \sqcap \exists\texttt{hasChild.Person}$$
$$\texttt{Father} \equiv \texttt{Man} \sqcap \exists\texttt{hasChild.Person}$$
$$\texttt{Parent} \equiv \texttt{Father} \sqcup \texttt{Mother}$$
$$\texttt{Grandmother} \equiv \texttt{Mother} \sqcap \exists\texttt{hasChild.Parent}$$
$$\texttt{Wife} \equiv \texttt{Woman} \sqcap \exists\texttt{hasHusband.Man}$$

Figura 3.2: Esempio di TBox.

Di norma in una TBox gli assiomi hanno due forme:

$$\texttt{C} \equiv \texttt{D} \text{ oppure } \texttt{C} \sqsubseteq \texttt{D}.$$

Nel primo caso si parla di *equivalenze*, nel secondo di *inclusioni*. Un'altra considerazione da fare è che non tutti le TBox possono essere considerate definitorie. Una Tbox infatti è tale solo se non presenta ciclicità. Per risultare aciclica una TBox deve soddisfare due condizioni:

- non definire lo stesso concetto due volte
- non definire concetti mediante frasi circolari (ovvero che hanno come parte del membro destro il concetto presente come membro sinistro.)

Una TBox ciclica, può generare problemi nella sua gestione e in genere è fortemente meno consigliata di una TBox aciclica, così come sono di relativa utilità le inclusioni generalizzate come

$$\texttt{GradFather} \sqsubseteq \texttt{Relative}$$

in quanto aiutano a chiarire un concetto, ma non rappresentano una frase definitoria in senso assoluto. E' bene tener presente che è in genere possibile passare da una TBox ciclica a una aciclica, sebbene nella maggior parte dei case tale operazione comporti dei costi computazionali abbastanza elevati; tanto che talvolta si preferisce lasciare le cose come stanno. In ogni

caso per effettuare un analisi sulla natura dei concetti sarà sempre necessario effettuare una *espansione* della TBox, cioè sostituire ricorsivamente ogni concetto con la sua definizione, in modo da ottenere come membro destro di ogni termine solo frasi contenenti concetti elementari. A seguire una parziale espansione della TBox in figura 3.2.

$$\texttt{Woman} \equiv \texttt{Person} \sqcap \texttt{Female}$$
$$\texttt{Man} \equiv \texttt{Person} \sqcap \neg(\texttt{Person} \sqcap \texttt{Female})$$
$$\texttt{Mother} \equiv (\texttt{Person} \sqcap \texttt{Female}) \sqcap \exists\texttt{hasChild.Person}$$
$$\texttt{Father} \equiv (\texttt{Person} \sqcap \neg(\texttt{Person} \sqcap \texttt{Female})) \sqcap \exists\texttt{hasChild.Person}$$

Figura 3.3: Espansione parziale della TBox di figura 3.2.

Proposizione 3.1.1 *Sia $\mathcal{T}$ una terminologia aciclica e sia $\mathcal{T}'$ la sua espansione. Allora:*

1. *$\mathcal{T}$ e $\mathcal{T}'$ hanno gli stessi simboli di base;*
2. *$\mathcal{T}$ e $\mathcal{T}'$ sono equivalenti;*
3. *$\mathcal{T}$ e $\mathcal{T}'$ sono entrambe definitorie.*

Fino a qui si è discusso sull'importanza di avere terminologie acicliche. Infatti solo grazie alle terminologie definitorie possono essere imbastire semantiche veramente *descrittive*. Tuttavia esistono anche situazioni in cui le definizioni cicliche sono intuitivamente indispensabili e non se ne può fare a meno. Ad esempio, si immagini di voler definire il concetto di "uomo che ha solo discendenza maschile", `MOMO` (Man who has Only Male Offspring). Tale uomo sarà per certo anche un `MOS` (Man who has Only Sons) laddove

$$\texttt{MOS} \equiv \texttt{Man} \sqcap \forall\texttt{hasChild.Man}.$$

Questo termine non è ciclico, ma non basta a descrivere `MOMO`. Per farlo serve necessariamente una descrizione ricorsiva, mettendo in rilievo il fatto

che anche tutta la discendenza del soggetto sarà composta da MOS; pertanto anche ili figlio del MOMO sarà un MOMO:

$$\texttt{MOMO} \equiv \texttt{Man} \sqcap \forall \texttt{hasChild.MOMO}.$$

ABox

Come detto Abox associa i ruoli e i concetti descritti in TBox a istanze specifiche. Risulta pertanto evidente che dipende fortemente da TBox. Come conseguenza si ha che questo insieme risulta molto più dinamico del primo. Ma non solo: facendo riferimento a ruoli espressi in TBox, ABox rischia costantemente di presentare inconsistenze. Infatti la conoscenza rappresentata in ABox deve sempre essere coerente con il modello definito nella terminologia, che potrebbe essere modificato in corsa.

```
Person(Paolo),    Person(Anna)
¬Female(Paolo),   Female(Anna)
Person(Marco),    ¬Female(Marco),
hasHusband(Anna,Paolo),
hasChild(Anna,Marco),
hasChild(Paolo,Marco)
```

Figura 3.4: Esempio di ABox

L'esempio di ABox qui sopra fa riferimento alla TBox di figura 3.2. Ad esempio è evidente che se nella TBox venisse aggiornata la definizione di Person, la ABox di figura 3.3 rischierebbe di dover essere stravolta.

Come detto, il compito della ABox è quello di presentare gli individui, dare loro un nome e asserire le loro proprietà. Come si evince anche da figura 3.4, questo viene fatto sia mediante l'utilizzo di ruoli R, sia mediante concetti C. Le asserzioni assumono quindi questa forma:

C(a) oppure R(b,c).

Nel primo caso si parla di *concept assertion*, che definisce un'appartenenza o una interpretazione, mentre nel secondo caso si parla di *role assertion*, la quale indica che `b` è legato a `c` tramite una relazione di tipo `R`. Questa dicotomia tra relazioni e appartenenza fa assomigliare la ABox a una sorta di DB relazionale, con legami di tipo unario o binario. L'unica differenza risiede nel fatto che per quel tipo di struttura vale una semantica "CWA", mentre la semantica delle ABox è "OWA". Da ultimo, è importante specificare l'importanza di una mappatura nella quale a distinti nomi di individuo, corrispondano differenti istanze; la *unique name assumption* garantisce una disambiguazione tutt'altro che scontata e assolutamente necessaria.

3.1.3 OWA vs CWA

Al contrario di altri formalismi, le logiche descrittive sono basate su Open World Assumption, che significa che se non possiamo inferire una frase, allora il suo valore di verità è sconosciuto. Al contrario, nel contesto Closed World Assumption ciò che non può essere inferito viene considerato falso; è il caso, ad esempio, del Prolog, dove tutta la verità deriva da delle operazioni di "match" con fatti presenti nella base di conoscenza. Il fatto che le DL siano OWA apre qualche possibilità in più, sebbene vengano richieste anche maggiori capacità di analisi e di ragionamento.

Prendiamo in considerazione la seguente ABox, presa a prestito dalla mitologia greca ma soprattutto, ancora una volta, da [4].

```
hasChild(IOKASTE, OEDIPUS),
hasChild(IOKASTE, POLYNEIKES),
hasChild(OEDIPUS, POLYNEIKES),
hasChild(POLYNEIKES, THERSANDROS),
Patricide(OEDIPUS),
¬Patricide(THERSANDROS)
```

Figura 3.5: ABox mitologica $\mathcal{A}_{mit}$

Come noto dalla mitologia, Edipo uccide il padre e sposa la madre Giocasta. Da lei ha un figlio, Polinice, il quale ha sua volta ha un discendente

(da Egia) di nome Tersandro. Nella nostra base di conoscenza Edipo risulta giustamente patricida, mentre è un fatto che Tersandro non lo sia. Supponiamo ora di voler sapere *se Giocasta abbia un figlio patricida che a sua volta abbia un figlio non patricida*. Formalmente

$$\mathcal{A}_{mit} \models (\exists \texttt{hasChild}.(\texttt{Patricide} \sqcap \exists \texttt{hasChild}.\neg\texttt{Patricide}))(\texttt{IOKASTE}) \ ?.$$

Giocasta ha due figli. Uno è Edipo, patricida, che a sua volta ha un figlio, Polinice. Tuttavia su Polinice non possiamo asserire nulla, non sappiamo se è patricida o meno. Così, Edipo sembra non essere il figlio che stiamo cercando. L'altro figlio di Giocasta è Polinice, su quale, come detto, non possiamo affermare nulla. Quindi anche Polinice non sembra essere l'obiettivo della nostra ricerca. In questo modo saremmo portati ad affermare che la risposta alla domanda che ci siamo posti sia NO. Ma non è così. Il ragionamento da fare è un altro.

Possiamo assumere due differenti modelli di $\mathcal{A}_{mit}$. Uno in cui Polinice è un patricida e un altro in cui non lo è. Nel primo modello Polinice è un figlio patricida di Giocasta, che a sua volta ha un figlio non patricida. Nel secondo modello Edipo è il figlio patricida e Polinice è il suo erede non patricida. Quello che conta è che in tutti i modelli (dal momento che non ve ne sono altri di nostro interesse) Giocasta è madre di un patricida che ha un figlio non patricida, sebbene non si possa identificare chi lo sia tra Edipo e Polinice.

Portare a termine questo ragionamento comporta un certo costo computazionale e in ogni caso ciò è reso possibile dalla flessibilità del OWA. Nel contesto CWA la nostra ricerca si sarebbe conclusa a basso costo computazionale, ma con la risposta sbagliata

3.1.4 Il linguaggio attributivo

Un breve passo indietro. Il linguaggio attributivo (*attibutive language* $\mathcal{AL}$) viene introdotto nel 1991 in [49] e costituisce la base, nonché il para-

gone, per tutti gli altri linguaggi che implementano le logiche descrittive. Le basi sintattiche di $\mathcal{ALC}$ sono:

- i concetti atomici;
- due strumenti per indicare "tutti i concetti" e "nessun concetto" ($\top$ e $\bot$);
- la negazione atomica (la negazione dei concetti complessi appare solo con una estensione del linguaggio);
- l'intersezione ($\sqcap$);
- le restrizioni universali ed esistenziali ($\forall$ e $\exists$).

Semantica e interpretazione

Sia $\mathcal{L} = \langle \mathcal{C}, \mathcal{R}, \mathcal{O}b \rangle$ un linguaggio descrittivo. Una *interpretazione* di esso è una coppia $\mathcal{I} = (\triangle^{\mathcal{I}}, f_{\mathcal{I}})$ dove $\triangle^{\mathcal{I}}$ è un insieme non vuoto (dominio) e $f_{\mathcal{I}}$ è la funzione di interpretazione che assegna ad ogni concetto atomico A un insieme $A^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}}$ e ad ogni ruolo atomico R una relazione binaria $R^{\mathcal{I}} \subseteq \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}}$.

La funzione di interpretazione si estende ai concetti come segue:

$$
\begin{aligned}
&\top^{\mathcal{I}} = \triangle^{\mathcal{I}} \\
&\bot^{\mathcal{I}} = \emptyset \\
&(\neg A)^{\mathcal{I}} = \triangle^{\mathcal{I}} \setminus A^{\mathcal{I}} \\
&(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}} \\
&(\forall R.C)^{\mathcal{I}} = \{a \in \triangle^{\mathcal{I}} \mid \forall b, (a,b) \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\} \\
&(\exists R.\top)^{\mathcal{I}} = \{a \in \triangle^{\mathcal{I}} \mid \exists b, (a,b) \in R^{\mathcal{I}}\}.
\end{aligned}
$$

Se una data interpretazione $\mathcal{I}$ soddisfa tutte le asserzioni di una ABox, si può dire che essa è un *modello* di quella Abox. Parimenti, se una interpretazione soddisfa tutte le definizioni di una TBox essa è un modello di quella Tbox. Una interpretazione $\mathcal{I}$ che sia modello sia della TBox che della Abox di una base di conoscenza, allora sarà anche un *modello di tale base di conoscenza*.

Estensioni di AL

L'aggiunta di nuovi costruttori alla grammatica di una logica, tra le altre cose, rappresenta un fattore importantissimo l'incremento della sua capacità espressiva.

Complementarietà Indicato con la lettera $\mathcal{C}$. Rappresenta la negazione di un concetto arbitrario o complesso.

$$(\neg C)^{\mathcal{I}} = \triangle^{\mathcal{I}} \backslash C^{\mathcal{I}}.$$

Unione Indicato con la lettera $\mathcal{U}$, si interpreta

$$(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}.$$

Restrizione numerica Indicato con la lettera $\mathcal{N}$ e si può scrivere $\geq nR$ (almeno n) oppure $\leq nR$ (al più n).

3.1.5 Relazioni con altre logiche

Le logiche descrittive più elementari (o con meno costruttori) sono la base di partenza per l'implementazione di logiche più articolate. L'aggiunta o la riduzione di potere espressivo e l'adattabilità a specifici domini applicativi sono i caratteri più importanti di una logica.

Logica del primo ordine

Il potenziale espressivo delle logiche descrittive è molto superiore a quello delle logiche del primo ordine. Questo talvolta comporta una impossibilità a trasporre una DL in una FOL e quando anche questo fosse possibile si rischierebbe di ottenere una logica dalle prestazioni davvero poco competitive.

Per passare da un linguaggio descrittivo a uno del primo ordine si deve ottenere:

- un predicato unario da ogni concetto atomico.

- una relazione binaria da ogni ruolo

- una costante da ogni nome di oggetto

In seguito si dovrà tradurre in FOL anche i concetti arbitrari, TBox e ABox.

Concetti Va individuata una formula F tale che, dato un concetto C e una variabile del primo ordine x, sia F(C,x). Alcuni esempi possono essere:

$$F(\top, x) = \top,$$
$$F(C \sqcap D, x) = F(C, x) \wedge F(D, x),$$
$$F(\forall R.C, x) = \forall y(T_r(x, y) \rightarrow F(C, y)).$$

TBox Data una TBox T, la sua traduzione F(T) si ottiene facendo la congiunzione di tutte le traduzioni delle equivalenze e delle inclusioni. $A = C$ si traduce con $\forall x(F(A, x) \leftrightarrow F(C, x))$, mentre $A \sqsubseteq B$ si traduce con $\forall x(F(A, x) \rightarrow F(C, x))$.

ABox Analogamente si procede con la congiunzione di tutte le traduzioni di ABox.

Logiche descrittive fuzzy

Le logiche fuzzy (ovvero logiche sfumate) sono logiche in cui le frasi non sono semplicemente vere o false, ma al contrario ad esse può essere associato un "grado di verità", esprimibile tipicamente con un valore compreso tra 0 (affermazione falsa) e 1 (affermazione vera). Questa logica nasce dalla teoria degli insiemi sfumati per andare a definire la realtà come somma di verità parziali. A partire da queste considerazioni si capisce come le logiche fuzzy abbiano reso le logiche descrittive più adatte a definire concetti vaghi, indefiniti e dai contorni imprecisi.

Logiche modali

Le logiche modali esprimono il "modo" in cui una preposizione è vera: questo avviene principalmente attraverso i due operatori di *possibilità* ($\Diamond$) e di *necessità* ($\Box$). Le logiche descrittive possono essere collegate anche alle logiche modali. In effetti alcune DL sono varianti di logiche modali.

Logiche temporali

Le logiche descrittive temporali sono la fusione delle logiche descrittive e di quelle temporali. Nascono dalla necessità di fondere la capacità di inferenza e la potenza descrittiva delle DL con le concezioni di sequenzialità o parallelismo temporale delle frasi il TL.

3.1.6 Inferenza

Lo scopo di una logica descrittiva non è solo quello di catalogare informazioni o di descrivere la realtà, ma anche quello di consentire ragionamenti sulla conoscenza acquisita. Tali ragionamenti hanno anche lo scopo di rendere esplicita la conoscenza implicita racchiusa nelle informazioni in TBox e ABox; ciò avviene mediante meccanismi di *inferenza*. Inferire un fatto B significa concludere che tale fatto sia vero, a partire da un altro fatto A assunto come vero: in sostanza, la verità è già "contenuta" in A e il processo di inferenza la rende esplicita. In questa sezione si discuterà dei problemi legati alle inferenze dei concetti della TBox e della ABox.

Inferenze sui concetti

La prima questione, di basilare importanza, nel momento della costruzione di una terminologia $\mathcal{T}$ è che i concetti creati abbiano un senso e non siano contraddittori tra loro. In particolare, un concetto ha senso se esiste almeno una interpretazione che soddisfa gli assiomi di $\mathcal{T}$ e tale che per essa quel concetto non sia denotato dall'insieme vuoto. Se così fosse, tale concetto sarebbe **soddisfacibile** rispetto a $\mathcal{T}$. Più rigorosamente

un concetto C risulta *soddisfacibile* se esiste un modello $\mathcal{I}$ di $\mathcal{T}$ tale percui $C^{\mathcal{I}} \neq \emptyset$.

Un altro problema è quello della **sussunzione**. Esso risulta strategico per ottimizzare la formulazione dei concetti e delle interrogazioni. Un concetto C è *sussunto* da un concetto D quando l'insieme indicato da C è un sottoinsieme di D. Oppure, più formalmente si può scrivere che un concetto C è sussunto da un concetto D rispetto a una terminologia $\mathcal{T}$ se $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ per ogni modello $\mathcal{I}$ di $\mathcal{T}$.

Altri due problemi sono quelli della **equivalenza** e della **disgiunzione**. Si può scrivere che due concetti C e D sono equivalenti rispetto a $\mathcal{T}$ se $C^{\mathcal{I}} = D^{\mathcal{I}}$ per ogni modello $\mathcal{I}$ di $\mathcal{T}$ e che due concetti C e D sono disgiunti rispetto a $\mathcal{T}$ se $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ per ogni modello $\mathcal{I}$ di $\mathcal{T}$.

Talvolta si può pensare di ridurre problemi di un tipo a problemi di sussunzione o di insoddisfacibilità.

Proposizione 3.1.2 *Dati due concetti C e D, volendo ridurre a sussunzione:*

- *C non è soddisfacibile $\Leftrightarrow$ C è sussunto da $\bot$;*
- *C e D sono equivalenti $\Leftrightarrow$ C è sussunto da D e D è sussunto da C;*
- *C e D sono disgiunti $\Leftrightarrow$ $C \sqcap D$ è sussunto da $\bot$.*

Proposizione 3.1.3 *Dati due concetti C e D, volendo ridurre a insoddisfacibilità:*

- *C è sussunto da D $\Leftrightarrow$ $C \sqcap \neg D$ non è soddisfacibile;*
- *C e D sono equivalenti $\Leftrightarrow$ sia $(C \sqcap \neg D)$ che $(\neg C \sqcap D)$ sono non soddisfacibili;*
- *C e D sono disgiunti $\Leftrightarrow$ $C \sqcap D$ non è soddisfacibile.*

Considerazioni sulla TBox

Non è difficile dimostrare che se $\mathcal{T}$ è una terminologia aciclica si possono equiparare i ragionamenti su $\mathcal{T}$ ai ragionamenti su una TBox vuota. Come si affermava nella proposizione 3.1, una terminologia e la sua espansione sono equivalenti. Dal momento che l'espansione C' di un concetto

C sostituisce i concetti arbitrari con una descrizione composta da concetti elementari, interpretati identicamente in ogni modello di $\mathcal{T}$, allora: *C è soddisfacibile rispetto a $\mathcal{T}$ se e solo se C' è soddisfacibile*. Analogamente *C e D sono disgiunti rispetto a $\mathcal{T}$ se e solo se C' e D' sono disgiunti*.

Risulta subito evidente quali siano i vantaggi di fare i conti con una TBox virtualmente vuota. Ciò detto, l'espansione dei concetti è tanto importante quanto onerosa. Tale costo, che nel caso peggione lievita in maniera esponenziale, nella maggior parte dei casi non può essere evitato.

Consistenza della ABox

Il problema della consistenza è un grosso problema, relativamente alla ABox: essa consiste nel verificare che una stessa istanza non sia correlata a concetti disgiunti. In sostanza è auspicabile che non si verifichi una cosa del tipo:

```
Mother(ANNA), Father(ANNA).
```

Una ABox $\mathcal{A}$ è consistente rispetto a una TBox $\mathcal{T}$ se esiste almeno una interpretazione che è modello sia di $\mathcal{A}$ che di $\mathcal{T}$. Ovviamente se tutto può succedere in relazione a una TBox vuota, non è così rispetto a una TBox non vuota, ad esempio quella espressa in figura 3.2. Così si può ridurre il problema della verifica della consistenza di una ABox dispetto a una TBox alla verifica su una ABox espansa rispetto a quella TBox. Da qui si possono dedurre tre considerazioni:

1. *$\mathcal{A}$ è consistente rispetto a $\mathcal{T}$ se e solo se la sua espansione $\mathcal{A}'$ è consistente*;
2. *$\mathcal{A} \models C(a)$ se e solo se $\mathcal{A} \cup \{\neg C(a)\}$ è inconsistente*;
3. *C è soddisfacibile se e solo se $\{C(a)\}$ è consistente*;

3.1.7 Estensioni

In molti contesti il potere espressivo dei linguaggi $\mathcal{ALCN}$ non è sufficiente, motivo per cui è necessario utilizzare delle estensioni. A seguire

ne verranno introdotte due: una per estendere i costruttori per i ruoli e un'altra per gestire restrizioni numeriche.

Costruttori per i ruoli

Essendo i ruoli delle relazioni binarie, risulta naturale impiegare operazioni binarie su di essi. Dati i ruoli R e S:

- **composizione**:
  $(R \circ S)^{\mathcal{I}} = \{(a,b) \in \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}} \mid \exists b, (a,b) \in R^{\mathcal{I}} \wedge (b,c) \in S^{\mathcal{I}}\}$
- **chiusura transitiva**:
  $(R^{+})^{\mathcal{I}} = \bigcup_{i \geq 1} (R^{\mathcal{I}})^{i}$
- **inverso**:
  $(R^{-})^{\mathcal{I}} = \{(b,a) \in \triangle^{\mathcal{I}} \times \triangle^{\mathcal{I}} \mid (a,b) \in R^{\mathcal{I}}\}$

Restrizioni di numero

Si accennerà a due tipologie di restrizioni sui numeri. La prima è una la cosiddetta *qualified number restriction* (identificata dalla lettera $\mathcal{Q}$). Esse sono restrizioni su ruoli legati a specifici concetti. O meglio, se prima con l'estensione $\mathcal{N}$ si poteva aggiungere una cardinalità a un ruolo ed esprimere una frase del tipo ≥ 2 `hasChild` intendendo "ha almeno due figli", ora si può scrivere ≥ 2 `hasChild.Male` col significato "ha almeno due figli maschi". Infine si può gestire anche il concetto di *variabile*; ad esempio si può scrivere `Person` $\sqcap \geq \alpha$ `hasDaughter` $\sqcap \leq \alpha$ `hasSon` per intendere "ha tanti figli maschi quanti femmina".

3.2 Semantic Web

In principio fu il Web 1.0. Nato nel 1991 e reso accessibile alle masse due anni dopo, il web ottiene presto un successo planetario grazie ad alcuni fattori determinanti, come:

- contenuti espressi in *linguaggio naturale*

- eventuali elementi multimediali (compatibilmente con le capacità infrastrutturali dell'epoca)

- possibilità di customizzare l'aspetto grafico al fine di rendere più accessibile (sperabilmente) il contenuto

- *Hyperlinks*

In particolare questo ultimo punto rivoluzionò il modo di cercare informazioni, rendendo fruibili dati sparsi per la rete in una maniera che ricalca il funzionamento del cervello umano, risultando pertanto intuitivo e immediato. Tuttavia risultò evidente sin da subito che il web così concepito si focalizzava interamente sul modo di rappresentare l'informazione, piuttosto che non sulla analisi della sua natura. *I know how, but I don't know what.* Gli stessi collegamenti erano privi di indicazioni o classificazioni in merito alla loro natura o alla natura degli elementi a loro connessi.

Lo sviluppo del web 2.0 non dette risposta a tali questioni. Nonostante la nascita di piattaforme sociali - dallo scambio di conoscenza fino al commercio elettronico - e dei Web Services avessero reso la rete più dinamica e interattiva, la gestione delle risorse avveniva sempre alla stessa maniera, relazionandosi cioè alla forma e non alla sostanza o essenza informativa di un dato specifico.

> *I have a dream for the Web [in which computers] become capable of analyzing all the data on the Web – the content, links, and transactions between people and computers. A 'Semantic Web', which should make this possible, has yet to emerge, but when it does, the day-to-day mechanisms of trade, bureaucracy and our daily lives will be handled by machines talking to machines. The 'intelligent agents' people have touted for ages will finally materialize.*

Queste parole vennero pronunciate da Tim Berners-Lee nel 1999. In sostanza l'inventore del web si immaginava uno scenario in cui i nodi della rete - mediante una non meglio specificata forma di intelligenza -

riuscissero a utilizzare i dati presenti, per estrapolare conoscenza in maniera autonoma. L'idea era quindi quella di dare importanza al contenuto informativo di un insieme di dati, portando il ragionamento sul piano semantico. Per fare questo si pensa di sfruttare la potenza di calcolo di agenti distribuiti che che dovranno necessariamente lavorare per mezzo di formati comuni per lo scambio e l'integrazione delle informazioni. Per la prima volta non si parla più di mero scambio di dati, ma si tenta di capire la relazione tra la conoscenza in esso contenuta e il mondo reale. Al fine di raggiungere questo scopo bisogna analizzare la struttura del web e comprenderne gli imprescindibili vincoli strutturali. Ad esempio, il fatto che i documenti siano distribuiti significa che vi è anche la possibilità che siano duplicati, parziali, incongruenti tra loro (sia sul piano del contenuto, che sul piano del collegamento fisico) o incompleti piuttosto che non del tutto falsi. In conclusione, per edificare e apprezzare una intelaiatura semantica del web le informazioni devono prima essere classificate e strutturate; in seguito sarà necessario un meccanismo di inferenza per orientarsi nell'enorme bacino della conoscenza acquisita o addirittura di inferirne di nuova.

3.2.1 Architettura e strumenti

L'intera architettura del web semantico, come è naturale che sia, poggia sui *dati*. Più nello specifico sulla rappresentazione che di essi viene data in rete. Il primo passo verso la costituzione di una semantica è quello di definire degli insiemi, delle classi mediante le quali descrivere i dati. Questa sorta di mappatura tra dati e concetti viene compiuta mediante dei *metadati*. Infine occorre tracciare vincoli e *relazioni* tra i vari domini informativi creati in precedenza.

Naturalmente per fare questo occorre definire degli strumenti di lavoro adeguati. In particolare parleremo di linguaggi, framework, ontologie e logiche.

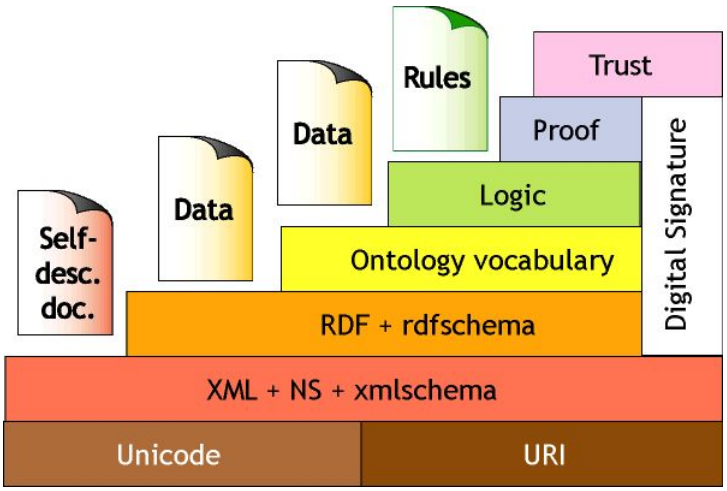


Figura 3.6: Semantic Web Cake

URI

Uno Uniform Resource Identifier si compone di una stringa di caratteri che identifica in maniera univoca una risorsa. Questo strumento è alla base del web attuale, ma costituirà anche la base del web semantico, in quanto costituisce un buon modo per rappresentare una specifica risorsa. In particolare gli URI hanno alcune peculiarità interessanti:

- Ciascun URI è *unico*. Non esistono due URI composti della medesima stringa di caratteri
- Ad ogni URI corrisponde una e una sola risorsa (o concetto, come lo chiameremo in seguito)
- Più URI differenti possono puntare alla stessa risorsa.
- Non necessariamente a un URI deve corrispondere un documento fisico. (E', ad esempio, il caso degli indirizzi mail)

Il penultimo punto rappresenta sia un aspetto positivo, poiché aggiunge notevole flessibilità e robustezza, ma anche una nota dolente in quanto potenzialmente introduce inconsistenze.

All'ultimo punto della lista qui sopra, viene esplicitata la dicotomia tra l'oggetto in se è la sua rappresentazione. A tal proposito va chiarito che

mentre un URI è un identificatore generico, un tipologia più particolare di URI è l'URL (Uniform Resourse Locator), il quale, oltre a identificare la risorsa, esplicita anche il meccanismo per accedervi. In definitiva un URI non rappresenta un complesso di istruzioni per raggiungere un file specifico nel Web (sebbene lo faccia anche), ma è un nome per una particolare risorsa, sperabilmente accessibile attraverso Internet.

Cenni su RDF

```
<rdf:RDF xmlns:rdf="http://www.w3.org/2000/01/rdf-syntax-ns#"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:foaf="http://xmlns.com/0.1/foaf/" >
  <rdf:Description rdf:about="www.marcosavoia.it">
    <dc:creator rdf:parseType="Resource">
      <foaf:name>Marco Savoia</foaf:name>
    </dc:creator>
    <dc:title>Semantic Web</dc:title>
  </rdf:Description>
</rdf:RDF>
```

RDF (acronimo di Resource Description Framework) è uno strumento basato su XML e proposto da W3c per la codifica e lo scambio di metadati strutturati. Le informazioni in questione sono relative a risorse identificate in maniera univoca attraverso degli URI e hanno a che fare, in particolare, con le *relazioni* tra le risorse, delle quali si possono esplicitare proprietà identificate da un nome e relativi valori. Come detto, RDF è figlio di XML e come tale ne eredita diverse caratteristiche, come l'immediatezza, l'estendibilità, la flessibilità e la struttura gerarchica. Tuttavia presenta anche delle differenze che lo rendono adatto al "contesto semantico": RDF è pensato espressamente per gestire metadati e riesce a veicolare legami di parentela tra concetti in maniera molto più diretta e subitanea di quanto non faccia XML, che al contrario ha un rapporto più stretto coi dati stessi, piuttosto che non con la loro descrizione. In sostanza la componente

semantica e quella sintattica all'interno di un RDF rimangono eterogenee, mentre estrapolarle da un file XML risulterebbe più complesso.

Le componenti di un RDF sono due:

- *RDF Model*: definisce un modello e descrive una sintassi per rappresentare i metadati
- *RDF Schema*: definisce schemi e vocabolari per i metadati

Similitudini col modello E/R

Sebbene RDF sia più espressivo, risulta facile cogliere le similitudini con il modello E/R, tanto che si potrebbe definire RDF come una sorta di modello E/R per il web. In questo tipo di visione i record possono essere visti come nodi RDF, il nome di una colonna di tabella potrebbe essere concepito come nome di proprietà, mentre il valore del campo corrisponderebbe al valore della suddetta proprietà.

Ontologie

Che cos'è una ontologia? Una ontologia è una rappresentazione condivisa, formale e non ambigua della conoscenza all'interno di uno specifico dominio di interesse. L'ontologia costituisce uno strumento indispensabile per organizzare - e quindi rendere fruibile - le informazione. Ora, nella descrizione appena data l'ontologia potrebbe apparire niente più che una sorta di "vocabolario", nel quale vengono illustrati tutti i concetti facenti parte di un determinato dominio. Tuttavia non è così: l'ontologia ha lo scopo di creare una sorta di base dati mediante la quale sia esplicitata una gerarchia dei concetti, organizzati non solo grazie a una relazione di sussunzione, ma anche mediante una relazione di tipo semantico.

Gli elementi di cui si servono le ontologie sono di diversi tipi, semplici o più articolati. Si possono avere *classi* (categorie), *individui* (istanze), *attributi*, *relazioni*, *vincoli*, *regole* (espresse in forma causale), *assiomi* (insiemi di più regole espressi in una data forma logica).

Come detto sin dall'inizio della nostra analisi, noi siamo propensi a considerare l'ontologia, come uno strumento per analizzare la realtà solo di uno o più specifici domini di interesse. Sarebbe infatti impensabile definire una singola ontologia per l'interno universo, per almeno quattro differenti ordini di ragioni:

- **Tempo**: Indicizzare e gerarchizzare ogni cosa richiederebbe uno sforzo incalcolabile in termini di tempo e di risorse messe in gioco.
- **Utilità**: Sarebbe davvero utile indicizzare tutto e subito? Probabilmente è meglio concentrarsi solo sui domini locali di interesse ed evitare di perdere tempo su domini che nessuno guarderà mai.
- **Praticità**: Ovviamente una ontologia così vasta richiederebbe troppo spazio per essere salvata e troppo tempo per accedervi. Per non parlare dell'eventualità in cui fosse richiesta una modifica di una certa parte del modello.
- **Coerenza**: Questo è il punto più interessante. Infatti per descrivere l'universo, l'intera realtà nel suo complesso, ci si troverebbe dinnanzi a domini tra i più disparati, ambiti concettuali talmente distanti tra loro da richiedere necessariamente modelli descrittivi disgiunti. Tale diversità potrebbe dare vita a inconsistenze tra le varie parti della nostra base di conoscenza. Controllare questo aspetto può risultare molto complesso.

Queste sono le ragioni per le quali ci si troverà sempre ad avere a che fare con ontologie specifiche, dette appunto *ontologie di dominio*. Al contrario, le *ontologie fondazionali* (ovvero *upper ontologies*), analizzano più di un dominio e forniscono una sorta di glossario di base di tutti i termini comuni di rilievo (e annesse relazioni, vincoli, attributi, etc.) dei domini d'interesse. Ciò detto, in generale quello della coerenza rimane un problema notevole. E' facile evincere come differenti background culturali possano portare a differenti rappresentazioni di identici domini. Così talvolta si rende necessario effettuare la fusione di modelli differenti tra loro; tali modelli possono riguardare lo stesso dominio o domini differenti

- magari con certe zone di sovrapposizione. Effettuare questo mapping tra due sistemi descrittivi differenti può risultare estremamente ostico. Di conseguenza sono oggetto di interesse - quindi di studio - tecniche per il merging di ontologie differenti.

In conclusione perché le ontologie sono uno strumento così importante? Perché permettono di definire con chiarezza l'intero dominio applicativo senza ambiguità, consentendo un'analisi e una comprensione molto profonda dello stesso. Infine aiutano a mettere in luce gli aspetti meno chiari e più peculiari. In sostanza, sebbene siano molto dispendioso in termini di tempo, portano un vantaggio notevole sia nell'ambito della progettazione in azienda, sia nel contesto accademico.

Altri strumenti

- **GRDDL**: acronimo di Gleaning Resource Descriptions from Dialects of Languages. E' un meccanismo per ottenere dati RDF da documenti XML e in particolare da pagine XHTML.
- **POWDER**: è un meccanismo utile a individuare e descrivere nel web. Può essere utile ad aiutare l'utente a "scovare" risorse d'interesse.
- **RIF**: definisce uno standard per lo scambio di regole tra differenti sistemi.
- **SPARQL**: Può essere utile a esprimere query attraverso differenti sorgenti di dati

3.2.2 RDF

Come accennato nella sezione 3.2.1, RDF è un linguaggio creato per rappresentare meta-informazioni riguardo le risorse presenti nel web. Queste possono essere tese ad aggiungere contenuto informativo alla risorsa - come nel caso di autore, data di creazione, titolo, etc. - o a identificare un o specifico oggetto in rete, anche quando questo non possa essere direttamente reperito. Inoltre è importante tenere presente che RDF ha lo

scopo primario di favorire lo scambio e l'elaborazione di informazioni tra applicativi, piuttosto che non tra utenti.

Dalle asserzioni al modello RDF

Come espresso in [60] RDF come un modo per comporre frasi sulle risorse della rete. Si prenda ad esempio la seguente frase:

> *http://www.esempio.org/index.html ha un autore il cui valore è Marco Savoia*.

Questa proposizione identifica alcuni elementi:

- ciò che viene descritto dalla frase (la pagina web, identificata dal suo URI);
- una proprietà ("essere autore di");
- il valore di tale proprietà.

Più formalmente, la parte che identifica la cosa di cui si sta parlando viene chiamata soggetto, la parte che identifica la proprietà è il predicato e il valore del predicato è l'oggetto della frase. Ovviamente tutte queste componenti sono identificate in maniera univoca:

- **soggetto**: http://www.esempio.org/index.html;
- **predicato**: http://purl.org/dc/elements/author;
- **oggetto**: http://www.esempio.org/staffid/561.

A questo punto, RDF modella la frase - o le frasi, come vedremo - sotto forma di grafo. Un nodo per il soggetto, un nodo per l'oggetto e un arco direzionato per il predicato.

Infine si può pensare anche di graficare un insieme di frasi, andando a ottenere un aggregato di informazioni maggiore. Pensiamo, ad esempio, di avere anche le frasi

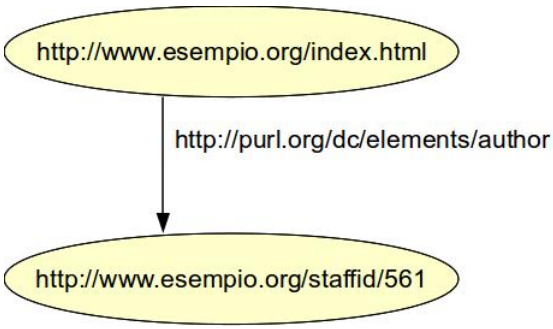


Figura 3.7: Grafo della proposizione.

http://www.esempio.org/index.html ha una data di creazione, il cui valore è 11 Luglio 2011.

e

http://www.esempio.org/index.html ha una lingua il cui valore è English.

quello che otterremmo sarebbe:

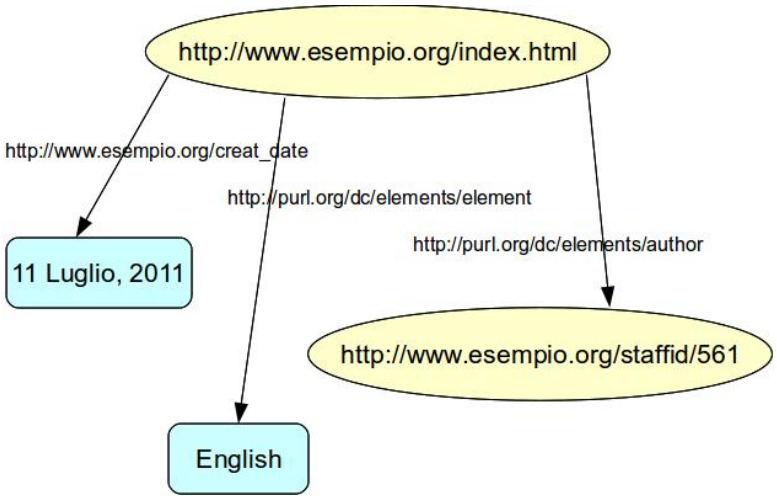


Figura 3.8: Grafo delle proposizioni.

Sintassi

Naturalmente RDF fornisce una sintassi basata su XML per esprimere i grafi in linguaggio scritto. Come esempio prendiamo il grafo sottostante.

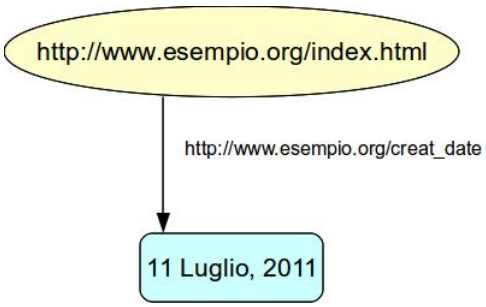


Figura 3.9: Grafico della proposizione.

La sua traduzione in RDF è la seguente:

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf=
  "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:exterms="http://www.esempio.org/terms/">

<rdf:Description
  rdf:about="http://www.esempio.org/index.html">
<exterms:creation-date>
  11 Luglio, 2011
</exterms:creation-date>
</rdf:Description>

</rdf:RDF>
```

In questo breve stralcio di codice sono esplicitate diverse informazioni. Nel tag iniziale rdf:RDF vengono dichiarati i namespaces, sia per l'RDF che per l'XML, individuati rispettivamente negli URI

http://www.w3.org/1999/02/22-rdf-syntax-ns#

e

http://www.esempio.org/terms/

Poi inizia la descrizione della risorsa, identificata con il tag `rdf:about`. Di questa viene specificato `exterms:creation-date` che ha come valore il

contenuto del tag. Nel caso in cui siano richieste più descrizioni per la stessa risorsa queste possono essere accorpate in una sola. Ad esempio, con riferimento al grafo di figura 3.8 si scriverà

```
...
<rdf:Description
  rdf:about="http://www.esempio.org/index.html">
  <exterms:creation-date>
    11 Luglio, 2011
  </exterms:creation-date>
  <dc:language>
    en
  </dc:language>
  <dc:author rdf:resource="http://www.esempio.org/staffid/561"/>
</rdf:Description>
...
```

Containers & Collections

Talvolta vi è la necessità di descrivere *gruppi* di cose: ad esempio si può voler indicare tutti i tecnici del suono che hanno contribuito a un film, gli autori di un libro o gli studenti di un corso. A questo scopo esistono i *Container*, che sono di diversi tipi:

Bag è una lista non ordinata di risorse o costanti. Viene utilizzato per dichiarare che una proprietà ha valori multipli. Per esempio i componenti di un convegno.

Sequence differisce da Bag per il fatto che l'ordine delle risorse è significativo. Per esempio si vuole mantenere l'ordine alfabetico di un insieme di nomi, gli autori di un sito.

Alternative una lista di risorse che definiscono un'alternativa per il valore singolo di una proprietà. Per esempio per fornire titoli alternativi in varie lingue.

Quindi un container stabilisce quali siano i suoi membri, tuttavia non dice quali *non* lo sono e non dice nemmeno che da qualche altra parte non ne possano essere nominati di nuovi. Per fare questo RDF utilizza le *Collection*, che sono una sorta di insiemi "chiusi" senza possibilità di nuove aggiunte.

RDF Schema

RDF Schema [61] permette di definire dei vocabolari e conseguentemente l'insieme delle proprietà semantiche di uno specifico gruppo di risorse. Inoltre consente di stabilire il significato, le caratteristiche, i vincoli e le relazioni di un insieme di proprietà. Inoltre, implementando il concetto di classe e sottoclasse, consente di definire uno schema a classi di tipo gerarchico. La *classe* in RDF è molto simile a quello di alcuni linguaggi di programmazione a oggetti, a partire dal Java. In sostanza le classi sono delle tipologie di oggetti, che possono anche essere in relazione tra loro. Nell'esempio sottostante vengono classificati diversi automezzi.

```
<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/rdf-schema#"
  xml:base="http://example.org/schemas/vehicles">

<rdf:Description rdf:ID="MotorVehicle">
  <rdf:type rdf:resource=
  "http://www.w3.org/rdf-schema#Class"/>
</rdf:Description>

<rdf:Description
    rdf:ID="PassengerVehicle">
  <rdf:type rdf:resource=
  "http://www.w3.org/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
```

```
</rdf:Description>

<rdf:Description rdf:ID="Truck">
  <rdf:type rdf:resource=
  "http://www.w3.org/rdf-schema#Class"/>
  <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>
</rdf:RDF>
```

Oltre a descrivere specifiche classi a cui le risorse appartengono, è necessario anche descrivere specifiche proprietà che caratterizzano i membri di tali classi. A questo scopo viene introdotto il tag `rdf:Property` e i tre `rdfs:domain`, `rdfs:range` e `rdfs:subPropertyOf`. L'ultimo tra questi indica una gerarchia tra le proprietà specificate, mentre `rdfs:range` indica che il valore di una determinata proprietà è una istanza di una specifica classe. Infine `rdfs:domain` viene utilizzato per indicare che una particolare proprietà si applica a una determinata classe. Ad esempio

```
<rdf:Property rdf:ID="registeredTo">
  <rdfs:domain rdf:resource="#MotorVehicle"/>
  <rdfs:range rdf:resource="#Person"/>
</rdf:Property>
```

significa che la proprietà `ex:registeredTo` si applica a `ex:MotorVehicle` e fa riferimento a una istanza della classe `ex:Person`.

3.2.3 OWL

> *XML is only the first step to ensuring that computers can communicate freely. XML is an alphabet for computers and as everyone who travels in Europe knows, knowing the alphabet doesn't mean you can speak Italian or French.*
>
> Business Week, 18 Marzo 2002

Come detto qui sopra, XML non può bastare. XML è l'alfabeto di diverse lingue: una tra queste è OWL [58] (ovvero Web Ontology Language). Esso è un linguaggio di markup che si basa ed estende RDF allo scopo di dare una rappresentazione esplicita della semantica e delle relazioni che intercorrono tra le varie entità di un dominio applicativo.Abbiamo visto come RDF offra una classificazione gerarchica, la definizione di proprietà (che possono essere anch'esse gerarchiche) e tutta una serie di restrizioni di dominio e di range. Tutto questo non è sufficiente? Perché usare OWL e come OWL estende RDF?

In effetti RDF riesce ad esprimere solo relazioni di tipo binario e non riesce a esprimere bene certe caratteristiche di alcune proprietà (come la simmetria o la transitività). Inoltre in RDF può risultare molto difficile descrivere concetti complessi o classi disgiunte. Infine ha delle lacune per quanto riguarda restrizioni locali o restrizioni riguardo la cardinalità. Per sopperire a tutto questo W3C ha fornito OWL, uno strumento che aggiunge espressività ad efficacia, senza intaccare la semplicità (è anch'esso basato su XML) o incrementare la complessità rispetto a RDF.

Breve storia ed evoluzione

Dagli anni '90 si è cercato di esplorare l'idea di rappresentazione della conoscenza. Sin da subito si è capita l'enorme potenzialità di linguaggi di markup come l'XML e si è lavorato per espanderli: da XML a RDF, da RDF fino a OWL. La W3C fondò nel 2001 un gruppo di lavoro per fare ricerca su ontologia e web e l'anno seguente uscì la prima pubblicazione su OWL. Il Web Ontology Language divenne uno standard nel 2004, per poi continuare a essere oggetto di ricerca e di modifiche; modifiche che hanno portato prima a OWL 1.1 a fine 2006 e poi OWL 2.0 a ottobre 2009.

Tre sottolinguaggi

OWL offre tre diverse varianti, che garantiscono differenti livelli di espressività. Gli utenti saranno liberi di scegliere quella che maggiormente si adatta alle loro esigenze, al loro contesto.

- **OWL Lite**: E' il più semplice dei tre. Particolarmente indicato per chi necessiti di una classificazione gerarchica e di vicoli basilari. E' di facile implementazione, ma con basso potere espressivo e consente una facile migrazione verso sistemi semplici che includano thesauri o differenti tassonomie. Tuttavia, sebbene OWL Lite abbia una minore complessità formale rispetto a OWL DL (tanto che diversi costrutti del DL possono essere forgiati mediante combinazioni complesse di elementi del Lite), l'implementazione di tool per il Lite si è dimostrata tanto complessa quanto quella di tool per il DL. Questo, di fatto, è uno dei motivi del suo scarso impiego.

- **OWL DL**: DL sta per *Description Logics*, per la forte corrispondenza con le logiche descrittive (legame che è alla base dell'intero OWL). Viene utilizzato da coloro i quali vogliono la massima espressività, pur vedendo garantiti completezza (tutto può essere calcolato) e decidibilità (il calcolo avviene sempre in un tempo finito). Comprende tutti i costrutti del linguaggio OWL, ma possono essere utilizzati solo sotto certi vincoli.

- **OWL Full**: E' adatto a chi ricerca massima espressività anche a costo di minori garanzie sulla decidibilità. Definito su semantiche differenti rispetto al Lite e al DL, consente di estendere il dizionario esistente o di trattare una classe sia come individuo che come insieme di individui.

Terminologia

OWL si fonda su quattro concetti basilari. Come si noterà, la correlazione con le logiche descrittive è molto forte.

- **Istanze**: Corrispondono agli *individui* nelle logiche descrittive.

- **Classi**: Una classe definisce un insieme di oggetti ed è assimilabile a quello che nelle logiche descrittive si chiamava *concetto*. Ogni classe può avere una o più istanze, che a loro volta possono appartenere anche a più di una classe. Sempre di derivazione DL è il concetto di

sottoclasse ($\sqsubseteq$) e il modo per indicare la *classe root* ($\top$) e la *classe vuota* ($\bot$).

- **Proprietà**: Sono relazioni binarie dotate di verso che intendono specificare una certa caratteristica. C'è forte corrispondenza con i *ruoli* delle logiche descrittive.

- **Operatori**: Sono ammesse varie operazioni sulle classi, come l'intersezione, la disgiunzione, restrizioni numeriche, l'unione, il complemento.

Sintassi

OWL supporta una molteplicità di sintassi, che possono essere suddivise in due tipologie: sintassi di alto livello e sintassi di scambio. Del primo tipo sono la *OWL abstract syntax* e la *OWL 2 functional syntax*, con lo scopo di specificare le varie semantiche, ma anche di mapparsi con le altre sintassi. Tra le sintassi di scambio ve ne sono una per la mappatura su XML, alcune per la mappatura su RDF e la sintassi Manchester, con una buona leggibilità e uno stile vicino a quello dei linguaggi Frame.

In particolare, un brevissimo scorcio sulla sintassi astratta. L'esempio presentato in seguito è abbastanza generico da poter essere ritenuto valido sia nel contesto di OWL Lite sia di OWL DL. Questa sintassi astratta viene presentata per mezzo di una grammatica EBNF dove i simboli terminali sono virgolettati e i non-terminali sono in grassetto, le alternative vengono presentate separate col simbolo |. Le parentesi quadre indicano che il componente può comparire al più una volta e le graffe che potrebbe comparire più volte.

Una ontologia OWL, nella sintassi astratta contiene una sequenza di *annotazioni*, *fatti* e *assiomi*. Le annotazioni hanno lo scopo di registrare le informazioni riguardo l'ontologia, come il nome dell'autore, il nome dell'ontologia e i riferimenti alla sua importazione. Tuttavia il vero contenuto informativo, quello di interesse, è veicolato mediante i fatti e gli assiomi.

ontology ::= 'Ontology(' [ontologyID] directive ')'
directive ::= 'Annotation(' **ontologyPropertyID ontologyID** ')'
| 'Annotation(' **annotationPropertyID URIreference** ')'
| 'Annotation(' **annotationPropertyID dataLiteral** ')'
| 'Annotation(' **annotationPropertyID individual** ')'
| axiom
| fact

Allo stesso modo le annotazioni possono anche trasportare contenuto informativo su nomi di classi, nomi di proprietà. Invece i fatti si riferiscono ai singoli individui in termini di valori e proprietà di quell'individuo e del numero di classi a cui esso appartiene. Talvolta al singolo individuo può essere associato un identificatore per rendere univoci i riferimenti a esso.

fact ::= **individual**
individual ::= 'Individual(' [**individualID**] **annotation** 'type('**type**')' **value** ')'
value ::= 'value(' **individualvaluedPropertyID individualID** ')'
| 'value(' **individualvaluedPropertyID individual** ')'
| 'value(' **datavaluedPropertyID dataLiteral** ')'

Infine gli assiomi. Vengono utilizzati per associare gli di classe e di proprietà con specifiche, sia parziali che complete delle loro caratteristiche. Talvolta gli assiomi sono anche detti definizioni.

axiom ::= 'Class(' **classID** ['Deprecated']
modality annotation description ')'
modality ::= 'complete' — 'partial'

laddove

description ::= **classID**
|**restriction**

|'unionOf(' **description** ')'
|'intersectionOf(' **description** ')'
|'complementOf(' **description** ')'
|'oneOf(' **individualID** ')'

Semantica

In contrasto con altri linguaggi, come Prolog e SQL, OWL è open world assumption. Questo significa che ciò che non viene esplicitamente espresso non viene valutato come falso, ma viene lasciata aperta la possibilità che sia vero. Questo naturalmente comporta una notevole flessibilità in più e consente ai ragionatori (non a meno di un costo computazionale più elevato) di poter inferire anche conoscenza aggiuntiva, oltre a quella esplicita. Come detto, OWL Lite e OWL DL discendono dalle logiche descrittive e ne ereditano buona parte della struttura e anche la caratteristica di favorire il ragionamento sulla conoscenza rappresentata. In particolare OWL 1.0, proprio come le DL, era un linguaggio $\mathcal{SHOIN}$, mentre OWL 2.0 è di tipo $\mathcal{SROIQ}$.

Più nel dettaglio: gli elementi di OWL

Sicuramente la potenza di una ontologia dipende dalla sua capacità di saper ragionare sugli individui. Ma per fare questo in maniera efficiente c'è bisogno di meccanismi per descrivere le classi a cui tali individui appartengono e le proprietà che questi ereditano in quanto membri di una specifica classe. E' evidente come poi si possa sempre aggiungere proprietà a singoli individui, ma buona parte dell'efficienza di una ontologia deriva dal ragionamento sulla struttura a classi.

Ogni individuo, nel mondo OWL è membro della classe `owl:Thing`. Ne consegue che tutto ne è sottoclasse. Al contrario la classe vuota è `owl:Nothing`.

Nell'esempio seguente sono specificate tre classi di domini differenti[3].

[3]Questo e gli esempi seguenti sono tratti dal sito `www.w3c.org`

```
<owl:Class rdf:ID="Winery"/>
<owl:Class rdf:ID="Region"/>
<owl:Class rdf:ID="ConsumableThing"/>
```

Al momento non sappiamo praticamente nulla su queste tre classi, a parte il loro nome. Non sappiamo nemmeno se esistono individui che appartengono a loro. Così è necessario introdurre qualche altro elemento; un costruttore fondamentale per le classi è `rdfs:subClassesOf`, che mette in relazione una classe più specifica, con una più generale.

Una definizione di classe si compone di due parti: la prima introduce il nome, la seconda è una lista di restrizioni. Queste ultime vanno a restringere il numero di individui che appartengono a quella classe, dal momento che le istanze di quella classe dovranno appartenere all'intersezione di tutte le restrizioni poste.

```
<owl:Class rdf:ID="Wine">
  <rdfs:subClassOf
    rdf:resource="&food;PotableLiquid"/>
  <rdfs:label xml:lang="en">wine</rdfs:label>
  <rdfs:label xml:lang="fr">vin</rdfs:label>
  ...
</owl:Class>

<owl:Class rdf:ID="Pasta">
  <rdfs:subClassOf
    rdf:resource="#EdibleThing" />
  ...
</owl:Class>>
```

Ovviamente il parametro `rdfs:label` fornisce un nome di classe più leggibile per l'utente, fornendo eventualmente (come nel caso sopra) una traduzione in più lingue. E come conseguenza dell'esempio sopra, è di grande interesse anche la definizione di tutti quegli individui che fanno parte di una specifica classe.

```
<Opera rdf:ID="Tosca">
  <hasComposer
    rdf:resource="#Giacomo_Puccini"/>
  <hasLibrettist
    rdf:resource="#Victorien_Sardou"/>
  <hasLibrettist
    rdf:resource="#Giuseppe_Giacosa"/>
  <hasLibrettist rdf:resource="#Luigi_Illica"/>
  <premiereDate rdf:datatype="&xsd;date">
    1900-01-14</premiereDate>
  <premierePlace rdf:resource="#Roma"/>
  <numberOfActs rdf:datatype="&xsd;
    positiveInteger">3</numberOfActs>
</Opera>
```

Da questo esempio si evince che "Tosca" è una istanza della classe "Opera" e su di essa vengono asseriti diversi fatti.

Ovviamente nel contesto del web non è possibile basarsi sul principio UNA[4] secondo il quale a ogni nome viene associato uno specifico individuo. Così anche OWL non se ne avvale. Tuttavia offre qualche strumento per specificare se due URI fanno riferimento alla stessa istanza o meno:

- `owl:sameAs`: Asserisce che due URI fanno riferimento allo stesso individuo;
- `owl:differentFrom`: Asserisce che due URI fanno riferimento a due individui distinti;
- `owl:AllDifferent`: Usato per asserire che una gli individui che compongono una data lista sono tutti differenti.

Infine le proprietà. Questo discorso è piuttosto vasto perché le proprietà sono una componente indispensabile per OWL al fine di andare a definire qualcosa in più di semplici tassonomie. Le proprietà asseriscono

[4]Unique Name Assumption

fatti sulle classi (e quindi su tutti i loro membri) o sui singoli individui. Esse sono di due tipi:

- Object properties: mettono in relazione due istanze
- Datatype proprieties mettono in relazione una istanza con un valore

```
<owl:ObjectProperty rdf:ID="madeFromGrape">
  <rdfs:domain rdf:resource="#Wine"/>
  <rdfs:range rdf:resource="#WineGrape"/>
</owl:ObjectProperty>
```

Come si evince dall'esempio sopra di una proprietà possono essere specificati i legami con determinate classi con l'utilizzo di campi come `rdfs:domain` e `rdfs:range`. Altri campi per definire maggiormente una proprietà possono essere `owl:equivalentProperty`, `rdfs:subPropertyOf`, `owl:inverseOf`, `owl:FunctionalProperty`, `owl:TransitiveProperty` e `owl:SymmetricProperty`.

OWL 2

Cosa cambia tra OWL 2 [62] e OWL 1? Non cambia il rapporto con RDF e XML. Non cambia il rapporto con le sintassi presentate prima. Inoltre c'è totale retro-compatibilità con OWL 1. Tuttavia OWL 2 ha arricchito i datatype e le proprietà, ha aggiunto tre nuovi profili[5] (o sottolinguaggi) e ha introdotto una nuova sintassi.

[5] EL, QL, RL

Capitolo 4

Centri di tuple semantici

Nei capitolo 2 si sono analizzate le caratteristiche dei centri di tuple ed è stato messo in luce il modello TuCSoN. Nel capitolo 3 sono state illustrate le possibili vie per realizzare un componente semantico e per descrivere uno specifico dominio ontologico. In questo capitolo si tenterà di sfruttare quanto appreso fin qui per aggiungere un componente semantico ai centri di tuple. Cerchiamo di capire perché questa operazione può essere interessante.

Nell'ultimo decennio internet è stata l'innegabile protagonista di una profonda e importante rivoluzione sociale. Una miriade di nuove applicazioni ha dettato e continua a dettare il passo di un cambiamento radicale e costante nel nostro modo di vivere il web. La nostra idea di "rete" si evolve senza sosta e a sua volta genera nuove esigenze (o tendenze) che trovano risposta in nuovi applicativi, creando così una spirale evolutiva i cui contorni non sono sempre facili da tratteggiare e nella quale il rapporto di causa-effetto tra necessità e innovazione è piuttosto sfocato. É in questo scenario che emergono distintamente due caratteristiche dominanti e determinanti: la distribuzione e l'apertura. Da una parte i componenti dei vari sistemi vengono distribuiti in più nodi della rete, dall'altra i sistemi in gioco accettano l'idea di divenire strumento di interazione per agenti non noti a priori. Analogamente sarà altrettanto normale per un agente cercare di divenire parte di un sistema sconosciuto, in uno scenario assolutamente dinamico. Ora, non sarà difficile comprendere come i numerosi vantaggi

di un universo applicativo così mutevole e flessibile siano accompagnati anche da nuove problematiche tanto complesse quanto ineludibili. La prima è certamente la sicurezza, una questione che in un simile contesto può assumere una rilevanza e una difficoltà non banali. Poi esiste il problema della fiducia tra i vari agenti e sistemi, ma bisogna anche occuparsi di mantenere sempre completa interoperabilità e coerenza tra i componenti. Infine c'è il problema di rendere accessibile il contenuto informativo racchiuso nei dati messi a disposizione alle varie componenti dei sistemi. Tuttavia anche questo compito è tutt'altro che semplice da assolvere, in quanto per definizione ad un modello aperto possono partecipare agenti estremamente eterogenei tra loro. Come garantire dunque interpretazioni coerenti? In definitiva come garantire la comprensione dell'informazione? La soluzione è quella di attribuire un significato ai dati ovvero costruire una interpretazione univoca degli stessi. Ecco dunque perché si è optato per la costituzione di blocco per il riconoscimento e la gestione di una semantica formale nel sistema che andremo ad analizzare: trovare un accordo sul significato comune da attribuire alle informazioni.

4.1 Verso i centri di tuple semantici

Illustrato il valore potenziale di una semantica formale nel contesto dei sistemi aperti, resta ora da spiegare perché il modello scelto debba essere proprio quello dei centri di tuple semantici. Per fare ciò è importante avere a mente quanto espresso alla sezione 3.2, ove si è discusso sulle potenzialità del Web Semantico. Quest'ultimo, in questi anni, sta contribuendo a rivedere il modo in cui si concepisce (o si concepirà) lo scambio di informazioni in rete; in quest'ottica un particolare riferimento lo meritano i Web Service [57]. Questo sistema software è stato pensato per garantire interoperabilità tra vari punti di una stessa rete, senza però porsi la questione della comprensione di tali dati scambiati. A tale scopo sono stati sviluppati in seguito i Semantic Web Service.

I Web Service lavorano in un contesto che replica decisamente il classico modello Client-Server, senza tuttavia possederne una importante carat-

teristica: i dati in questione non vengono messi a disposizione dal server in maniera che il client vi possa accedere "attivamente", ma al contrario il meccanismo è quello dello scambio di messaggi. Tuttavia questo paradigma stride parecchio con il modello classico di Internet. Per far sì che la comunicazione sia più in linea con gli standard del Web è ideale un modello come quello degli spazi di tuple dove le informazioni rimangano in maniera persistente lato Server, fino a che non ne venga invocata una esplicita rimozione [39]. Inoltre va considerato che prendendo in considerazione la naturale evoluzione degli spazi di tuple, ovvero i centri di tuple così come descritti al capitolo 2, allora potremmo anche godere degli enormi vantaggi in merito alla coordinazione che tali modelli offrono nativamente.

4.1.1 Le caratteristiche del sistema

Da quanto discusso fino ad ora si evince che un centro di tuple semantico deve possedere tre fondamentali caratteristiche:

Coordinazione Fornita nativamente dal sistema a partire da un modello comunicativo basato sulle tuple. Presente negli spazi di tuple. Rafforzata nei centri di tuple.

Programmabilità Prerogativa dei centri di tuple, potenzia le capacità di coordinazione del sistema, rendendolo più modellabile sulle specifiche esigenze contestuali e svincolando gli attori da ogni *coordination awarness* potenzialmente richiesta negli spazi di tuple.

Supporto Semantico Con un linguaggio per definire le ontologie si intende creare un mezzo per tracciare una rappresentazione semantica della conoscenza racchiusa nel centro di tuple.

4.2 Il modello

Abbiamo osservato che affinché il centro di tuple semantico si possa definire tale, deve possedere le caratteristiche esposte qui sopra, alla se-

zione 4.1.1. Più nel dettaglio, per implementare questi aspetti si rendono necessari alcuni ingredienti specifici [37], come:

- **Una ontologia** - A ogni centro di tuple viene associata una ontologia che fa riferimento a uno specifico dominio. Sulla base di questa viene edificata una TBox.
- **Tuple semantiche** - Le tuple semantiche rappresentano i singoli individui in riferimento all'ontologia associata al centro di tuple.
- **Un template per le tuple semantiche** - Sono dei template sulla base dei quali vengono recuperate le tuple associate a una specifica ontologia.
- **Primitive semantiche** - Operazioni base che il sistema mette a disposizione degli utenti per "consumare" le tuple semantiche nel sistema. Come anche nei modelli descritti in precedenza, esse fanno riferimento all'inserimento, alla rimozione e alla lettura della tupla nel sistema.
- **Reazioni semantiche** - Insieme di azioni complesse composte di azioni elementari. Esse si attivano in reazione a determinati eventi e sono definite mediante un *reaction specification language*.

4.2.1 Ontologia e TBox

Quanto alla sezione 3.1 si è parlato di logica descrittiva, sono stati introdotti i concetti di ABox e TBox. In riferimento a quanto esposto in quel frangente, un centro di tuple semantico necessita di entrambi questi luoghi concettuali. La ABox ce la si può immaginare come la collezione di tutte le tuple nel sistema che, come detto prima, rappresentano gli individui. Al contrario, la TBox descrive i concetti, le loro proprietà, i ruoli che li legano a partire da una specifica ontologia. Prendendo a riferimento le logiche descrittive di tipo $\mathcal{SHOIN}(\mathcal{D})$ possiamo considerare validi tutti i costruttori logici illustrati nella tabella 3.1 - unione, intersezione, negazione, restrizione universale ed esistenziale, inclusione ed equivalenza - e

sulla base di questi fornire un breve esempio immediato a partire da un ipotetico *dominio dei libri*:

`Writer` $\sqsubseteq \top$
`Book` $\sqsubseteq$ `(>=1 hasWriter)`
`(>=1 hasWriter)` $\sqsubseteq$ `Book`
$\top \sqsubseteq \forall$`hasWriter.Writer`
`FantasyBook` $\sqsubseteq$ `Book`

Qui vengono illustrati il concetto `Book` e il concetto `Writer` e viene espresso il vincolo che per ogni libro deve esistere almeno uno scrittore ad esso legato dal ruolo `hasWriter`. Infine viene affermato che `FantasyBook` è un sottoinsieme di `Book`.

Naturalmente per poter lavorare correttamente nell'ambito di un centro di tuple semantico, un agente deve essere conscio dell'ontologia a cui il CTS[1] fa riferimento. L'unico modo per far sì che vengano da lui effettuate operazioni coerenti all'interno del sistema è dargli l'opportunità di acquisire la descrizione formale del dominio (tipicamente espressa in forma testuale e locata su ogni nodo o punto d'accesso per il CTS). Ciò detto, ora è bene spendere due parole anche sull'evoluzione del sistema. Un centro di tuple semantico è fortemente caratterizzato dal dominio a cui fa riferimento la sua semantica e si potrebbe dire che questo aspetto sia più caratterizzante dello spazio fisico (o logico) sul quale esso si trovi; questo perché l'ontologia ad esso associata descrivendo le tuple semantiche in esso contenute di fatto pone dei vincoli sulla loro costituzione e sulla loro permanenza nel sistema. Alla luce di tutto ciò, "sfilare" l'ontologia a run-time per andarla a sostituire con un'altra completamente diversa minerebbe la consistenza e quindi la validità di tutto il modello. Tuttavia è possibile ipotizzare che, sebbene meno dinamica della ABox, anche la TBox possa evolversi nel tempo, anche se questo significa necessariamente andare a verificare (e potenzialmente eliminare) eventuali incoerenze manifestate dalla presenza di tuple che sono improvvisamente divenute semanticamente obsolete. É da notare come questo procedimento pos-

[1]Centro di Tuple Semantico.

sa rivelarsi tanto oneroso sul piano computazionale quanto invasivo sul piano del risultato.

4.2.2 Tuple semantiche

Come detto, una tupla semantica è un frammento di ABox, ovvero modo di rappresentare determinate informazioni coerentemente con quanto espresso nella TBox. Quindi quando parliamo di tuple semantiche scendiamo dall'ambito astratto dei concetti, per passare all'ambito concreto degli individui. Una di queste tuple può incorporare il seguente contenuto informativo:

- il nome dell'individuo;
- il nome del concetto dell'individuo;
- il nome delle relazioni in cui è coinvolto l'individuo, che a loro volta devono formalizzare:
  - il nome della relazioni;
  - il nome degli altri individui ad esso legati attraverso queste relazioni.

In altre parole possiamo tradurre in tuple ciò che in logica descrittiva avremmo scritto come `C(a)` e `R(a,b)`, cioè l'appartenenza dell'individuo `a` al concetto `C` e il legame degli individui `a` e `b` mediante la relazione `R`. Un esempio, in accordo con quello della sezione precedente può essere:

```
Writer(welsh)
Writer(tolkien)
Book(glue)     hasWriter(glue, welsh)
FantasyBook(the_hobbit)     hasWriter(the_hobbit, tolkien)
```

Se da una parte le tuple possono essere un buon modo per rappresentare il maniera persistente il contenuto di una ABox, dall'altra bisogna però che il meta-modello semantico sul quale esse si appoggiano si basa su specifiche tecnologie. Come puntualizzato egregiamente in [44], quando si opera

in un contesto semantico OWL e RDF Schema sono due strumenti tra i più usati e anche molti altri progetti oltre TuCSoN, come si vedrà nel capitolo 6, si rifanno al Triple Space Computing [17]. Per quanto riguarda invece la logica descrittiva di riferimento si è ritenuto $\mathcal{SHOIN}(\mathcal{D})$ il miglior compromesso tra efficienza ed espressività.

4.2.3 Template semantici

Mentre quando viene effettuata una operazione di matching sintattico il confronto tra due istanze avviene solo sul piano della forma, durante un matching semantico è tutto più complesso, perché il raffronto deve avvenire coerentemente con quanto scritto nella TBox. L'indagine più banale che è possibile effettuare nell'ambito delle logiche descrittive è l'*instance checking*, ovvero la verifica di appartenenza di una istanza a un determinato concetto. Analogamente in un centro di tuple si può avere la necessità di accertare se una data tupla appartenga o meno a un concetto del dominio ontologico; per fare questo si utilizzano i template semantici, cioé dei formalismi atti a svolgere interrogazioni semantiche sul centro di tuple. Riprendendo ancora una volta l'esempio dei libri che abbiamo utilizzato fin qui, possiamo ad esempio volerci chiedere se in ABox esistano libri scritti da Cesare Pavese: questa frase sarebbe trascritta

$\texttt{Book} \sqcap \exists \texttt{hasWriter.pavese}$

4.2.4 Primitive semantiche

Il un centro di tuple sintattico le tre operazioni basilari sono *in*, *out* e *rd*, rispettivamente per cavare, mettere e leggere una tupla nel sistema. Tipicamente queste chiamate sono bloccanti, ma al di là di questo aspetto esse possono avere successo o fallire a seconda della presenza o meno di almeno una tupla nel sistema la quale risponda ai requisiti sintattici espressi nel corpo di queste interrogazioni (*in* e *rd*) o nel caso di errori formali nella composizione dell'invocazione. Nel caso dei CTS invece è tutto molto meno immediato. Si pensi al caso in cui si voglia banalmente immettere un in-

dividuo nel CTS : `the_silmarillion:'FantasyBook'(hasWriter.tolkien).` A questo punto è necessario operare delle verifiche di consistenza per accertarsi che il concetto `FantasyBook` esista realmente nella mia TBox e che `hasWriter` sia una delle possibili relazioni. Se così non fosse l'operazione in questione dovrebbe inevitabilmente fallire, dimostrando in questo modo che in un centro di tuple semantico anche l'operazione meno problematica, la *out* può fallire. Inoltre va ricordato che per il principio UNA[2] può esistere un solo individuo con un certo nome e per garantire questo sarà necessario un ulteriore controllo oltre quelli già effettuati sulla TBox.

4.2.5 Reazioni semantiche

Un linguaggio di specifica per le reazioni deve definire *i*) quali siano le reazioni per un determinato modello e *ii*) gli eventi ai quali esse devono rispondere. Anche nel contesto semantico,le reazioni possono contenere primitive coordinamento per accedere e modificare la conoscenza semantica locata nel CTS, attraverso le tuple semantiche e i template semantici. Così, in questo caso l'evento intercettato non sarà più legato alla forma, ma al contenuto informativo. Naturalmente, anche in questo caso, per trasporre il ragionamento dal mero match sintattico al più complesso piano ontologico è necessario un costo computazionale maggiore.

4.2.6 Alcuni altri aspetti

Match sintattico

Quando il progetto TuCSoN semantico era ancora in fase di analisi si è pensato se fosse giusto ipotizzare un modello nel quale tutte le tuple fossero semantiche. Se si fa un po' di ricerca si potrà facilmente scoprire che in letteratura la maggior parte dei lavori prodotti presenta questa caratteristica, la quale tuttavia, a dispetto del largo utilizzo che ne viene fatto, presenta alcuni svantaggi:

[2]Unique Name Assumption

- tutti gli elementi inseriti nel CTS devono avere un riscontro con l'ontologia. In altre parole questo significa che è necessario riuscire a modellare qualsiasi cosa vi sia necessità di inserire e che se ciò non è possibile (o se la tupla esula dal dominio in questione) l'inserimento di tale oggetto sarà impossibile;

- Come conseguenza del punto precedente si ha che sarà impossibilitato l'inserimento di eventuali elementi a mero scopo comunicativo o coordinativo. Questo genere di tupla avrebbe chiaramente una rilevanza solo ed esclusivamente per un match di tipo sintattico;

- Una computazione sintattica è decisamente più snella dal punto di vista del carico computazionale e non sembra sciocco appoggiarsi alla parte semantica solo in certi casi.

Principalmente per questi tre ordini di ragioni si è deciso di preservare in TuCSoN la possibilità di continuare a gestire anche tuple sintattiche.

OWA e CWA in TuCSoN

Cosa si intenda per Open World Assumption o per Closed World Assumption lo si è già discusso alla sezione 3.1.3 a pagina 22, fornendo anche l'esempio di Edipo proposto per la prima volta da Baader e Nutt in [4]. Naturalmente la questione è stata affrontata anche nel contesto di TuCSoN semantico per decidere come il sistema avrebbe dovuto reagire all'inserimento o alla lettura di determinate tuple. É emerso che probabilmente la soluzione migliore consiste nel mantenere un atteggiamento differente tra ABox e TBox, la prima con OWA e la seconda con CWA. Vediamo brevemente il perché.

La **ABox** contiene gli individui presenti nel sistema, dopo che essi sono stati inseriti sotto forma di tuple a che la consistenza di tale inserimento sia stata verificata compatibilmente anche con la TBox e con gli altri elementi della ABox in modo da non presentare conflitti.

A partire dal contesto dei libri discusso in precedenza si supponga ora di voler inserire due tuple, una riferita ad un autore, l'altra riferita a un libro. Consideriamo i seguenti gruppi di asserzioni:

```
Writer(terry_brooks)
hasBirthDate(terry_brooks, 08_01_1944)
wroteBook(terry_brooks, the_sword_of_shannara)
```

laddove `hasBirthDate` e `wroteBook` siano naturalmente proprietà del concetto `Writer` e

```
Book(the_sword_of_shannara)
hasPublicationYear(the_sword_of_shannara, 1977)
hasAuthor(the_sword_of_shannara, terry_brooks)
```

con `hasPublicationYear` e `hasAuthor` proprietà del concetto `Book`. La rappresentazione delle tuple in RDF potrebbe essere la seguente. Que-

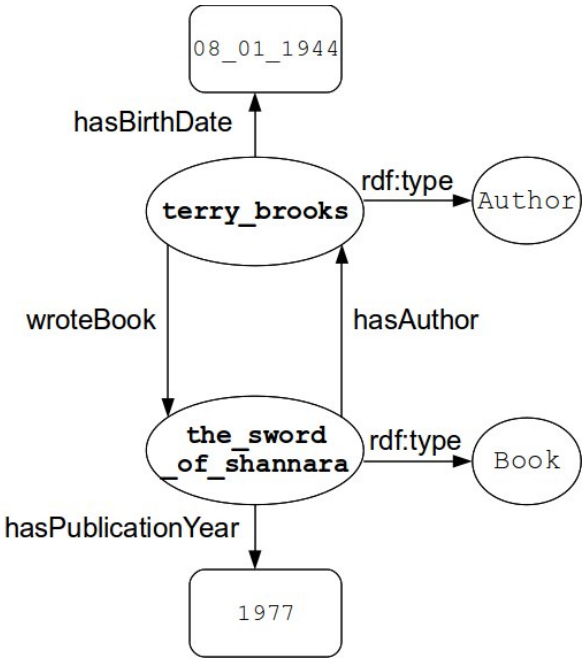


Figura 4.1: Rappresentazione delle tuple in stile RDF

ste due tuple distinte hanno riferimenti incrociati, nei ruoli `wroteBook` e `hasAuthor`. Se ipotizzassimo di trovarci in un contesto CWA, allora sorgerebbe un grosso problema: indipendentemente dall'ordine di inserimento,

dopo l'ingresso della prima tupla nel sistema nascerebbe un problema di consistenza. Immettendo prima lo scrittore la ABox, sarebbe incoerente per la mancanza del libro, e viceversa immettendo prima il libro la ABox sarebbe incoerente per l'assenza dello scrittore. L'unico modo per ovviare a questo inconveniente è quello di assumere un universo "aperto".

Per quanto riguarda la **TBox** assumere OWA potrebbe essere rischioso. Ciò significherebbe che sono ammessi inserimenti o letture di concetti non esistenti in ontologia, nella speranza che questi vengano inseriti in seguito. Tuttavia questa pratica, se rivolta ad operazioni bloccanti, può ipoteticamente lasciare in attesa dei processi in maniera indefinita, nel caso in cui l'inserimento dei concetti coinvolti venga disatteso. Per questo motivo sulla TBox si applica CWA.

4.3 Progettazione e tecnologia

In questa sezione vediamo ora come l'introduzione della semantica modifica l'architettura del nostro sistema.

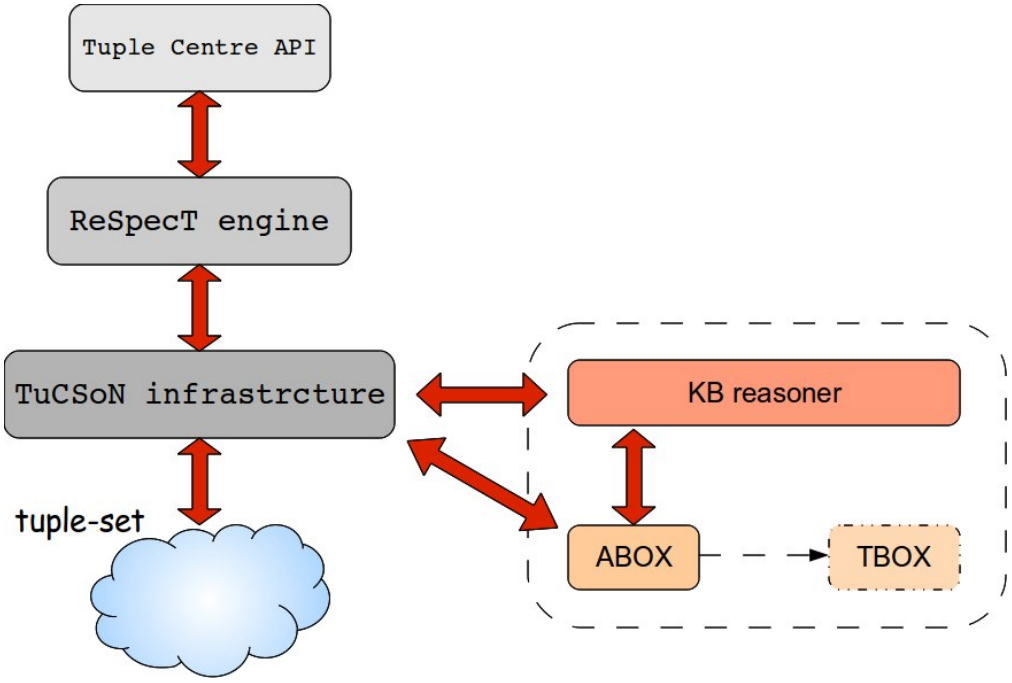


Figura 4.2: Architettura di TuCSoN

In figura 4.2 viene messa in luce la presenza di un blocco (quello tratteggiato sulla destra) che contiene un *ragionatore*, una *ABox* e una *TBox*. La ABox contiene gli individui inseriti attraverso le tuple, che quindi ora non saranno più solo all'interno del tuple-set. La TBox contiene le notazioni sui concetti e sui ruoli utilizzabili. Infine il reasoner offre la possibilità di effettuare ragionamento sulla base di conoscenza mediante le possibilità offerte dalla logica descrittiva.

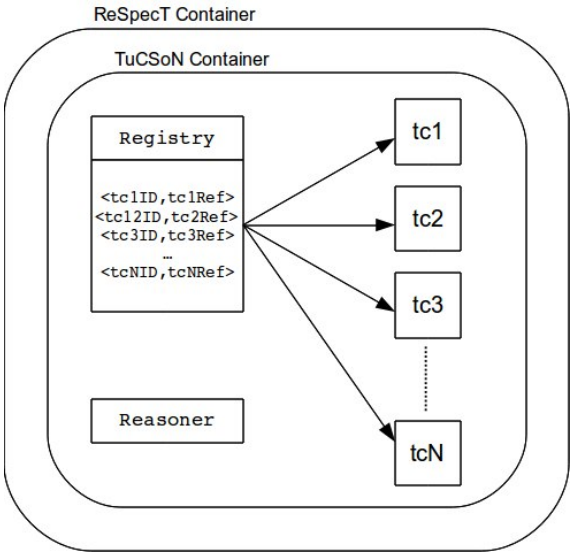


Figura 4.3: Architettura di un nodo TuCSoN

Nella concezione distribuita sulla quale nasce TuCSoN ci si immagina il *nodo* come un luogo nel quale si possano trovare più centri di tuple. Ad ogni nodo viene assegnato un TuCSoN Container che a sua volta contiene un ReSpecT Container. In quest'ottica l'unica cosa che cambia con l'introduzione della semantica è l'aggiunta del ragionatore, posto che le singole istanze all'interno del sistema sono costituite come detto in precedenza e come rappresentato in figura 4.2.

4.3.1 Il matching

In precedenza si è parlato di operazioni sul centro di tuple e di query. Ma cosa succede esattamente quando viene invocata una query? Una quesy, in sostanza, ha lo scopo di verificare se una data tupla semantica sia o

meno compatibile con un certo template. Per fare ciò è possibile seguire due metodologie:

- **instance checking**
  questa procedura ha lo scopo di verificare se un certo individuo appartiene a un concetto;

- **retrieval**
  questa procedura ha lo scopo di trovare tutti gli individui che appartengono a un dato concetto.

Per trovare una tupla compatibile con un certo template di norma l'instance checking va reiterato fino a quando non viene individuato un risultato positivo, con un caso peggiore pari al numero di tuple presenti nell' intero sistema. Al contrario, il retrieval non necessita di tale reiterazione, con l'effetto di risultare complessivamente meno pesante[3].

Ciò detto, nel modello di matching sintattico di TuCSoN la procedura seguita era molto più simile all'instance checking così come esposto qui sopra. Per questo motivo, in definitiva per coerenza, si decise di evitare il retrieval nonostante questo sia potenzialmente il migliore dei due.

4.3.2 SPARQL in TuCSoN

Come si è visto, i concetti vengono definiti dall'ontologia. Tuttavia potrebbe rendersi necessario l'utilizzo di concetti che non siano attualmente definiti, senza però voler pagare il costo di inserimento degli stessi nella TBox. SPARQL [59] si presta molto bene a questo, poiché può essere in grado di formalizzare concetti "al volo", ovvero in via temporanea. Infatti Pellet, che supporta pienamente SPARQL, è in grado di eseguire operazioni (come ad esempio il *retrieval*) attivando query atte a rappresentare specifici concetti. Ad esempio si supponga di avere un template come

```
'Book' (exists hasWriter :  'Writer')
```

[3]Va tuttavia fatto notare come, presa singolarmente, sia di norma più leggera l'operazione di instance checking. Tuttavia in questa dissertazione si è valutato le due operazione contestualizzandole nel modello in esame.

che in logica descrittiva si traduce con

Book ⊓ (∃hasWriter.Writer).

Questo template può essere tradotto in una query SPARQL come illustrato qui di seguito in figura 4.1.

```
SELECT *
WHERE
{
  { ? X rdf : type : Book . }
  {
    {
      ? X rdf : type _ : b0 .
      _ : b0 rdf : type owl : Restriction .
      _ : b0 owl : onProperty : hasWriter .
      _ : b0 owl : someValuesFrom : Writer .
    }
  }
}
```

Sorgente 4.1: SPARQL query

Capitolo 5

Fuzziness

La parola *fuzziness* può essere tradotta nella nostra lingua con le espressioni "vaghezza" o "incertezza". Conseguentemente è *fuzzy* tutto ciò che risulta indistinto, non ben definito e in definitiva difficilmente catalogabile all'interno di sistema di classificazione rigido. Zadeh in [65] rileva come nel mondo reale difficilmente si incontrino classi di oggetti che si prestino ad essere facilmente categorizzati. La stessa classificazione delle specie animali, per esempio, è una rappresentazione schematica che l'uomo da di uno specifico dominio per poter gestire e sfruttare in maniera più rapida e intelligente la conoscenza acquisita; il migliore pregio di questo modo di schematizzare la realtà è, in maniera duale, anche il suo più grande difetto, ovvero la semplicità. In sostanza, sebbene spesso non vi sia una attinenza stretta in senso matematico, questo modo di astrarre e rappresentare l'universo agevola il pensiero e la comunicazione.

Partendo da queste osservazioni si può affermare che i sistemi standard di classificazione possono risultare inadatti o limitanti; così si è iniziato a studiare un modo per rappresentare più fedelmente la vaghezza e la naturale "imperfezione" del mondo reale.

5.1 Introduzione alla fuzziness

Lukasiewicz e Straccia in [29] affermano che rientrano nel concetto di fuzziness tutti quei contesti in cui le affermazioni sono vere con una certa

sfumatura, con un certo grado. Ne consegue che una interpretazione assegna a una frase un certo "grado di verità", almeno fino a quando non è possibile stabilire se essa sia interamente vero o falsa. Sempre nello stesso articolo viene riportata a titolo d'esempio la frase "il pomodoro è maturo". Mentre il concetto di *pomodoro* è facilmente definibile, il concetto di *maturo* è molto più difficile da descrivere in maniera rigorosa e univoca; in sostanza è proprio l'essere "maturo" che risulta essere una categorizzazione imprecisa ed è da questa che poi si determina la vaghezza di tutta la frase. In effetti un pomodoro può essere più o meno maturo o, in altre parole, appartenere all'insieme dei frutti maturi solo in certa misura. Ma come è possibile definire la classe dei frutti maturi? Per fare questo sfruttiamo la notazione di *fuzzy set* che ha dato Zadeh in [65, 66] e che è stata anche ripresa in altri lavori tra cui [14].

Fuzzy Set

Sia X un insieme di oggetti che possiamo chiamare *universo*. Gli elementi generici appartenenti a tale universo sono detti x. Tipicamente siamo abituati a vedere una "classificazione" come l'appartenenza dei vari oggetti ad A, sottoinsieme di X, e a definirla come una funzione μ_A che varia tra i due valori 0 e 1. Ovvero:

$$\mu_A(x) = \begin{cases} 1, & \text{iff } x \in A, \\ 0, & \text{iff } x \notin A. \end{cases} \tag{5.1}$$

Tuttavia se ipotizziamo di espandere il codominio della funzione di appartenenza dal ristretto insieme digitale a tutti i valori reali compresi in $[0, 1]$, allora possiamo iniziare a parlare di "grado di appartenenza". Più un valore di μ_A si avvicina a 1, maggiore sarà il suo grado di appartenenza all'insieme A. Naturalmente A cessa di avere dei margini definiti nitidamente per assumere invece contorni più sfumati.

Così A è caratterizzato dalla coppia

$$A = \{(x, \mu_A(x)), x \in X\}. \tag{5.2}$$

Nel caso in cui X sia un insieme finito $\{x_1, x_2, ..., x_n\}$, allora un sottoinsieme su X si può definire

$$A = \mu_A(x_1)/x_1 + \mu_A(x_2)/x_2 + ... + \mu_A(x_n)/x_n = \sum_{i=1}^{n} \mu_A(x_i)/x_i. \quad (5.3)$$

Al contrario, quando X non è finito

$$A = \int_X \mu_A(x)/x. \quad (5.4)$$

Definizioni

- Tutti gli elementi di X appartengono a X con grado 1, cioè $\forall x \in X, \mu_X(x) = 1$
- Un fuzzy set si dice vuoto quando la sua funzione di appartenenza vale 0 su tutto X: $\forall x \in X, \mu_\varnothing(x) = 0$
- si definisce *supporto* (o *sostegno*) di un fuzzy set A la chiusure dell'insieme dei punti del dominio dove la funzione non si annulla: $suppA = \{x \in X, \mu_A(x) > 0\}$
- I *punti di crossover* sono quegli elementi del dominio per i quali vale $\mu_A(x) = \frac{1}{2}$
- L'*altezza* di A corrisponde al massimo valore del codominio raggiunto nel sottoinsieme A: $hgt(A) = sup_{x \in X} \mu_A(x)$

Operazioni sugli insiemi

A seguire alcune operazioni sui fuzzy set.

Uguaglianza

Due fuzzy set A e B sono uguali (e allora scriveremo $A = B$) se e solo se $\forall x \in X, \mu_A(x) = \mu_B(x)$;

Complemento

Il complemento di un fuzzy set A è $\bar{A}$ ed è definito dalla

seguente funzione di appartenenza

$$\forall x \in X, \quad \mu_{\bar{A}}(x) = 1 - \mu_A; \tag{5.5}$$

Unione

L'unione di due fuzzy set A e B (con le loro funzioni di appartenenza $\mu_A(x)$ e $\mu_B(x)$) è un nuovo fuzzy set C, scritto $C = A \cup B$ e la cui funzione di appartenenza si lega a quelle di A e B come segue:

$$\forall x \in X, \quad \mu_{A \cup B}(x) = Max(\mu_A(x), \mu_B(x)). \tag{5.6}$$

L'unione di A e B risulta essere il più piccolo insieme che li contiene entrambi, pertanto risulta che

$$Max(\mu_A(x), \mu_B(x)) \geq \mu_A(x)$$

e

$$Max(\mu_A(x), \mu_B(x)) \geq \mu_B(x).$$

Come nel caso degli insiemi tradizionali questa operazione sia associativa che commutativa.

Intersezione

L'intersezione di due fuzzy set A e B (con le loro funzioni di appartenenza $\mu_A(x)$ e $\mu_B(x)$) è un nuovo fuzzy set C, scritto $C = A \cap B$ e la cui funzione di appartenenza si lega a quelle di A e B come segue:

$$\forall x \in X, \quad \mu_{A \cap B}(x) = Min(\mu_A(x), \mu_B(x)). \tag{5.7}$$

Naturalmente se A e B sono disgiunti, la loro intersezione risulta vuota.

Nell'illustrazione 5.1 a seguire, tratta da [65], si vede un esempio di intersezione e di unione tra due funzioni di

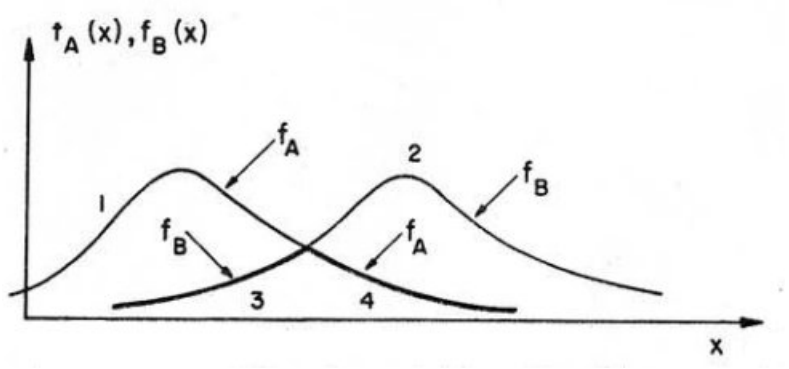


Figura 5.1: Unione e intersezione nei fuzzy set.

appartenenza (che Zadeh ha denominato $fA(x)$ e $fB(x)$). L'unione è rappresentata dalla somma dei due segmenti 1 e 2, mentre l'intersezione dalla somma dei segmenti 3 e 4.

Sottoinsieme

A è contenuto in B (quindi A è sottoinsieme di B) se e solo se $\mu_A(x) \leq \mu_B(x)$ e scriveremo

$$A \subseteq B \Leftrightarrow \mu_A(x) \leq \mu_B(x). \tag{5.8}$$

Qualche osservazione

Nell'ambito della cosiddetta "insiemistica fuzzy" perde quasi ogni valenza parlare di un punto generico x come *appartenente* a un fuzzy set A, a meno di non voler (banalmente) indicare che $\mu_A(x) \geq 0$. Così si può pensare di introdurre due livelli α e β tali che

$$0 < \alpha < 1, \quad 0 < \beta < 1, \quad \alpha > \beta$$

e conseguentemente affermare che x *appartiene ad* A se $\mu_A(x) \geq \alpha$, che x *non appartiene ad* A se $\mu_A(x) \leq \beta$, oppure che x *appartiene con un certo grado incertezza ad* A se $\beta < \mu_A(x) < \alpha$. Questa logica a tre valori o anche "logica a soglie" consente di poter definire con minore vaghezza il grado di appartenenza di una certa x a un insieme fuzzy. Nella figura seguente invece, tratta da [14], si mette in luce come i due punti x_1 e x_2 abbiano lo

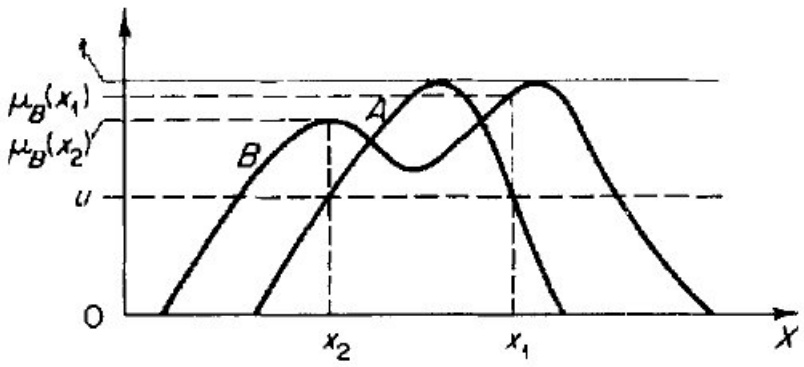


Figura 5.2: Esempio di di fuzzy set sovrapposti.

stesso grado di appartenenza ad A, ma due gradi di appartenenza differenti rispetto a B. In particolare si può vedere che $\mu_B(x_1)$ è maggiore di $\mu_B(x_2)$. Da notare anche come $htg(A) = htg(B)$.

Infine, per concludere questa introduzione ai fuzzy set una illustrazione sempre da [14], nella quale sono rappresentati esempi di intersezione, unione e complemento di due funzioni di appartenenza A e B.

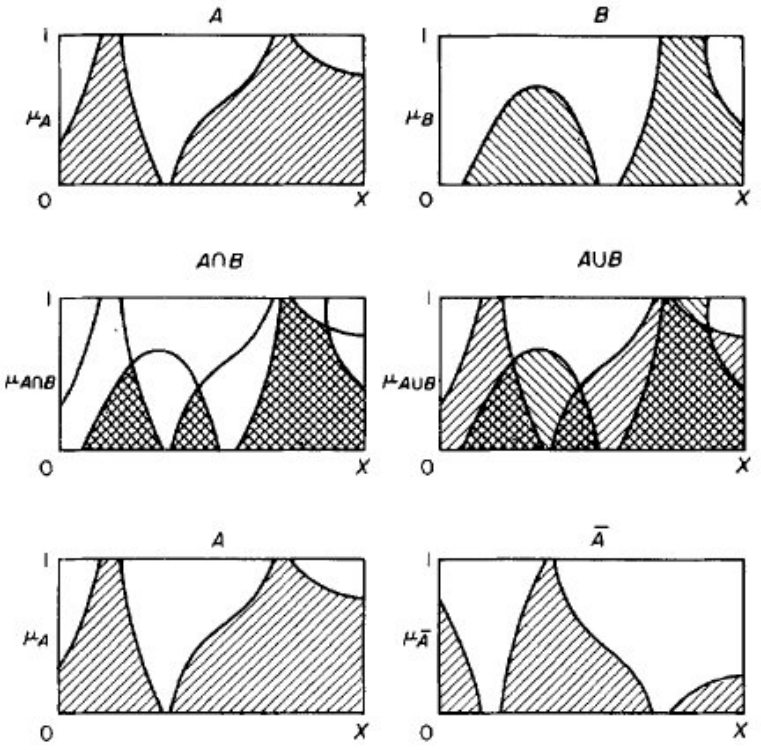


Figura 5.3: Esempio di di fuzzy set sovrapposti.

Defuzzificare

Fino a qui abbiamo analizzato le caratteristiche di fuzzy set, si sono osservate le implicazioni dell'appartenenza a tali insiemi e si sono apprezzate le differenze tra questa braca degli "humanistic systems" e la concezione tradizionale di insiemistica. Così, mentre l'appartenenza di un ragazzo a un (sotto)insieme crisp come quello dei "teenagers" è verificata o meno, un individuo x appartiene solo con un certo grado di verità al sottoinsieme delle "persone giovani". Tuttavia, una volta assegnato il grado di appartenenza e definita la relativa funzione $\mu_A(x)$, come è possibile ritornare al valore crisp? Questa operazione, si capisce, sarà quantomeno complessa e pagherà necessariamente un certo grado di imprecisione.

Dopo i primi lavori di Zadeh sui fuzzy set, per qualche tempo il problema della *defuzzificazione* è stato toccato solo marginalmente, talvolta solo sfiorato, altre volte contestualizzato fortemente al sistema analizzato. In seguito, come si ricorda in [48], emersero principalmente due metodi di defuzzificazione, ovvero il "centro di gravità" (COG) e il "mean of maxima" (MOM). Poi nel 1993 Yager e Filev [63] trasformarono la funzione di appartenenza ai fuzzy set in una distribuzione di probabilità, calcolando così il valore rappresentativo di un determinato sottoinsieme come il valore atteso rispetto a tale distribuzione. Inoltre nel loro approccio consideravano anche l'utilizzo di un parametro[1] detto di "confidenza" o di "fiducia", con il quale intendevano misurare la bontà e la fedeltà del processo di fuzzificazione. Successivamente Saade e Schwarzlander individuarono una nuova soluzione al problema a partire da quelle esistenti e formularono il *total distance criterion* (TDC). La sua applicabilità è ristretta solo a certi tipi di fuzzy set e si basa sull'integrazione lungo l'asse delle ordinate della media aritmetica dell'α-level set di uno specifico sottoinsieme:

$$F(A) = \int_0^1 M(A_\alpha) d\alpha \tag{5.9}$$

[1] Il *confidence parameter* può essere assegnato soggettivamente o grazie all'utilizzo di complessi algoritmi di learning.

con l'α-level set A_α definito come

$$A_\alpha = \{x \in X;\ \mu_A(x) \geq \alpha\} \quad \forall \alpha \in (0, 1]. \tag{5.10}$$

5.2 Fuzzy DL

Tipicamente le logiche descrittive trattano dati crisp, tuttavia in letteratura è possibile trovare alcune estensioni e modifiche di $\mathcal{SHOIN}(\mathcal{D})$ tali da ottenere una generalizzazione fuzzy. A seguire le componenti di questa DL.

Datatype

É possibile ragionare con stringhe e interi per ricreare i fuzzy set: ad esempio *alto* o *basso*, così come la specifica $<_{70}$.

Modificatori

É possibile utilizzare modificatori come *molto*, *poco*, *abbastanza*, *scarsamente* e applicarli ai fuzzy set per modificare le loro funzioni di appartenenza.

Assiomi

La fuzzy TBox è un insieme finito di concetti fuzzy. Tuttavia, oltre a definire i concetti $\mathcal{SHOIN}(\mathcal{D})$ introduce i *fuzzy nominals*, o meglio l'accostamento di uno o più individui o_i ai relativi gradi di appartenenza α_i a un determinato concetto: $\{\alpha_1/o_1, ..., \alpha_m/o_m\}$. Un'altra aggiunta sono i *concetti pesati*. Un concetto pesato C rappresenta un concetto tale che per ogni individuo a, il suo grado di appartenenza a (αC) è data da α volte il grado di appartenenza a C. Analogamente un *concetto con somma pesata* è un concetto che rappresenta la somma pesata dei concetti C_i: $(\alpha_1 * C_1 + ... + \alpha_N * C_N)$. Infine i *concetti con soglia*. Questi ultimi indicano, ad esempio, che il grado con cui un indi-

viduo a è istanza di $([\geq \alpha]C)$ è 0 se a è istanza di C con almeno grado α.

5.3 Fuzziness nei centri di tuple

In qusta sezione si prenderà in considerazione l'ipotesi di adattare i centri di tuple così come li abbiamo analizzati nel capitolo 4 alla fuzziness [36, 35] in maniera da ampliare ulteriormente l'aspetto semantico visto fin qui. Per fare ciò è necessario descrivere alcune modifiche sostanziali da apportare al modello di CT semantico.

Ontologia fuzzy

Come detto anche in 5.2 il modo migliore per descrivere una ontologia fuzzy è quello di utilizzare $\mathcal{SHOIN}(\mathcal{D})$. Conseguentemente risultano molto interessanti i lavori su Fuzzy OWL [54] con particolare riferimento alla letteratura prodotta da Bobillo e Straccia [7, 8] in merito a Fuzzy OWL2.

Tuple fuzzy

Come nel caso delle tuple crisp anche le tuple fuzzy rappresentano gli individui della ABox. Però in questo caso viene associato un grado di verità per l'appartenenza di ogni elemento della ABox ad un concetto o un ruolo definito nella TBox. Così ad esempio

```
FantasyBook(a_game_of_thrones) ≥ 0.75
hasWriter(a_game_of_thrones, martin) = 1
```

poiché `a_game_of_thrones` appartiene alla categoria dei `FantasyBook` solo con un grado di `0.75`, mentre possiamo affermare con assoluta certezza che il suo autore è `martin`; quest'ultimo valore non è crisp e partanto viene associato a `a_game_of_thrones` mediante il ruolo `hasWriter` con grado `1`.

Template fuzzy

Come detto in precedenza $\mathcal{SHOIN}(\mathcal{D})$ fornisce gli strumenti per definire concetti fuzzy. In riferimento all'esempio di prima, potremmo voler stabilire che ci interessa considerare `FantasyBook` sono gli individui legati a questo concetto con grado almeno pari al `0.7`.

Primitive fuzzy

Come conseguenza dell'introduzione della fuzziness nel sistema, anche le primitive andranno modificate, o meglio, estese. in, out, e rd dovranno ora operare con tuple fuzzy: frequentemente sarà di interesse selezionare (quindi si parla di in e rd) elementi dal tuple-set in relazione al loro grado, visto che è proprio questa la caratteristica dominante di questo modello. Così, mentre nei CT visti fino ad ora il principio di selezione di una tupla era basato sul non determinismo, ora si possono compiere potenzialmente scelte alternative. Pur restando valida l'idea di scegliere una tupla "a caso", si può pensare di adottare altri principi che individuino, ad esempio, quella col grado maggiore o la risultante di algoritmi basati sulla distribuzione di probabilità. In ogni caso è chiaro che violare il principio di non determinismo significherebbe dover ripensare e riscrivere anche le politiche di comunicazione e coordinazione.

Reazioni fuzzy

Anche in questo caso il comportamento ricalca ed estende quello delle reazioni nei CT visti fin'ora. anche esse andranno estese per poter gestire tuple fuzzy, tenendo presente quanto dello al punto precedente: la modifica del comportamento delle primitive (e quindi in definitiva anche del loro significato) può sconvolgere interamente le politiche e i meccanismi di comunicazione, ma soprattutto di coordinazione, in funzione dei quali le *reaction* vengono concepite.

Capitolo 6

Architettura

In questo capitolo si uniranno le conoscenze acquisite nei capitoli precedenti e si cercherà di convergere verso un'analisi dello stato attuale del progetto TuCSoN semantico, apprezzandone i pregi e valutandone i difetti. Poi verranno illustrati eventuali percorsi di sviluppo alternativi per il futuro. É possibile ripensare l'architettura del sistema? Quali sono le opzioni? E infine, quali sono le tecnologie che meglio si sposano con la nuove strutture analizzate?

6.1 Lo stato dell'arte: pregi e difetti

Come descritto nel capitolo 4, TuCSoN semantico presenta un'architettura ibrida, nel senso che mantiene del tutto separate le tuple cosiddette "sintattiche" da quelle "semantiche". Tale eterogeneità si pone in evidenza in parte nel linguaggio, ma soprattutto a livello implementativo. Sebbene venga utilizzato sempre lo stesso set di operazioni - `in()`, `out()`, `rd()`, `rdp()`, `inp()` -, questa rappresenta solo un'omonimia formale, in quanto la procedura d'inserimento di una tupla semantica è differente, più complessa e più costosa - di quella di una tupla sintattica. A differenza di altri contesti, come Semantic Web Spaces [55] ad esempio, nel nostro caso gli universi sintattico e semantico rimangono completamente slegati, sia come tuple, che come operazioni su di esse. Inoltre, come detto, anche l'architettura stessa si presenta come fortemente eterogenea: infatti il mo-

tore che si occupa dell'unificazione sintattica non è lo stesso che si occupa di quella semantica. Questi due blocchi rimangono completamente estranei l'uno all'altro, come estranee e disgiunte sono le due working memory sulle quali essi operano.

In [44] vengono raccolte alcune delle motivazioni che hanno spinto verso la direzione fin qui illustrata. La scelta di continuare a supportare sia le tuple di tipo sintattico che quelle di tipo semantico va controcorrente rispetto alla tendenza generale del *semantic tuplespace computing* [26] che è quella di condensare tutto in un unico TS, eliminando di fatto le tuple non semantiche. Tuttavia in quel momento l'opzione di dover modellare tutta l'informazione nel TS sulla TBox è parsa eccessivamente restrittiva e limitante: poter gestire molteplici strutture dati - in modo differente - può rivelarsi davvero utile, in quanto ci si può facilmente immaginare come tuple sintattiche possano essere utilizzate come medium comunicativo piuttosto che non per collezionare dati che non si intende o che non si è in grado di inserire nella TBox. Infine in [35] viene anche messo in luce quale sia il costo della computazione semantica: il linguaggio ReSpecT, ma soprattutto il ragionatore Pellet comportano una flessione notevole delle performance rispetto al caso della semplice valutazione di matching sintattico.

Fermo restando quanto si è detto fin qui, in questa tesi ci si interroga su quali possano essere in vantaggi di una eventuale architettura differente da quella attuale, specie in relazione ai potenziali guadagni in termini di efficienza ed espressività. Ma per fare questo è necessario comprendere quali siano i possibili modelli di riferimenti e quali siano in generale i pro e i contro che li contraddistinguono.

6.2 Architetture ibride

Con l'espressione "architettura ibrida" si vogliono intendere tutti quegli schemi di costruzione di sistemi software i quali presentino disomogeneità tra i loro vari moduli costitutivi concreti o concettuali. Questo tipo di architetture presenta in genere tutti i vantaggi derivanti da un alto grado di modularità, ma altresì porta in dote anche svantaggi non trascurabili

come la necessità di fare interagire componenti che trattano dati differenti o che usano linguaggi differenti. Lo sforzo verso l'integrazione di componenti totalmente eterogenee porta spesso alla generazione di livelli di interfacciamento come saldatura tra essi. In altri casi si può optare per la fusione dei blocchi stessi. Quale strada prendere e perché viene determinato dalle specifiche politiche di *coupling* che si intende adottare.

6.2.1 *Cohesion* e *coupling*

Una breve dissertazione su questi due aspetti della progettazione di un sistema software aiuterà l'analisi delle sezioni successive. In un sistema informatico *cohesion* e *coupling* sono due indici importanti per la misurazione delle interconnessioni esistenti tra le varie componenti e dentro di esse. La coesione, che in questa trattazione ci interesserà marginalmente, rappresenta quanto fortemente siano interconnessi i blocchi funzionali interni ai vari moduli di un sistema. Un modulo altamente coesivo compie pochi (o uno soltanto) task all'interno di una procedura, dovendo così richiedere una bassa interazione con procedure avviate altrove. Invece, l'accoppiamento esprime il grado di dipendenza tra i vari moduli applicativi. In [45] viene definito come segue:

> *Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.*

Tuttavia lo stesso autore di questa definizione, nel libro sopra citato fa a sua volta riferimento a [53], articolo in cui Constantine stabilisce la cosiddetta *software quality metrics*, una misurazione quantitativa rigorosa - e pertanto oggettiva e perfettamente quantificabile - dei due parametri sin qui illustrati. In particolare qui verrà illustrata solamente quella relativa al coupling, in quanto, come detto, di maggior interesse per questa trattazione.

In riferimento alla tabella 6.1 risulta evidente che i tipi di accoppiamento siano classificati in ordine, a partire da quelli più deboli fino ad arrivare

No coupling	I vari moduli non hanno nessun tipo di contatto, scambio o comunicazione tra loro.
Message coupling	Il più basso livello di coupling. La comunicazione avviene per scambio di messaggi.
Data coupling	Avviene uno scambio di dati elementari, tipicamente attraverso l'interfaccia di uno dei due moduli.
Stamp coupling	Attraverso l'interfaccia di uno dei due moduli avviene uno scambio di parte di strutture dati complessi.
Control coupling	Cessione del flusso di controllo da un modulo all'altro.
External coupling	Avviene quando un moduli è connesso a un ambiente esterno al software, tipicamente a un dispositivo specifico con un formato o un protocollo di comunicazione ad hoc.
Common coupling	Due o più moduli condividono dati globali. Sono necessarie politiche di coordinazione.
Content coupling	Un modulo fa uso di dati contenuti all'interno di un altro modulo.

Tabella 6.1: Coupling metric.

a quelli più forti. Questi ultimi rientrano nella definizione generica di *tight coupling*, ovvero contesti nel quali c'è un forte scambio di informazione, forte interdipendenza e pertanto anche la necessità di adeguate politiche di sincronizzazione. Al contrario, si definiscono *loose coupling* quelle situazioni nelle quali avvenga un minor scambio di dati e vi sia una minor interdipendenza tra determinati blocchi costitutivi di un sistema.

6.2.2 Modelli differenti

In questa sezione verranno proposti quattro differenti sistemi, nei quali due moduli interni ad essi siano posti ad interagire con modi differenti.

Ai fini dell'analisi non è di rilievo sapere se tali sistemi contengano anche altri blocchi operativi oltre quelli rappresentati, in quanto il ragionamento verterà solamente sull'interazione tra i due in questione.

Loosely integrated modules

Il primo caso prevede che i due blocchi operativi siano totalmente disgiunti e che ciascuno di essi possegga una propria working memory contenente dati utili al suo funzionamento ma anche gli output della sua computazione. Tuttavia ci immaginiamo uno scenario in cui vi sia l'esigenza di dover porre in comunicazione questi due blocchi: tale interazione avviene sia perché essi intendono invocare procedure dell'altro modulo, sia perché richiedono dati non contenuti nella propria working memory. Certamente

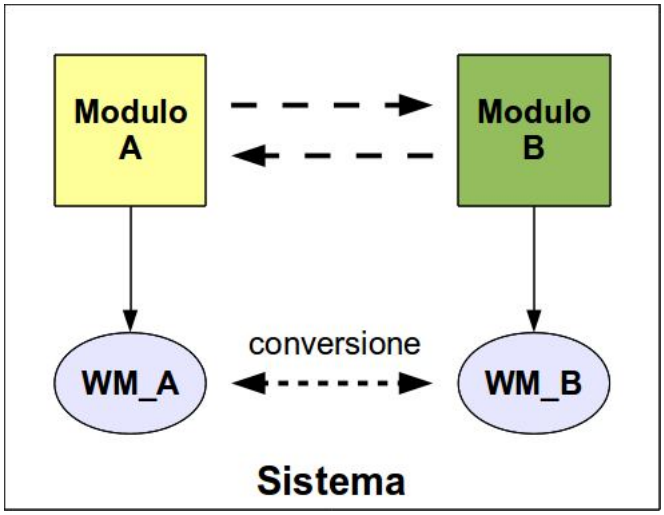


Figura 6.1: Moduli loosely coupled, con working memory disgiunte.

un sistema configurato in questa maniera è certamente modulare e come tale presenta dei vantaggi. Come prima cosa favorisce la fase di analisi, in quanto è più facile per il progettista concettualizzare il modello complessivo se lo si suddivide in componenti elementari, ciascuno specializzato verso uno specifico aspetto della computazione. Inoltre così facendo si agevola la fase di sviluppo e in modo particolare la fase di debug, a causa del fatto che una interazione limitata comporta una limitata propagazione degli (eventuali) errori. Un altro vantaggio, in genere, è quello che una ar-

chitettura di questo tipo spesso conduce, se ben sviluppata, a considerare i componenti come delle *black box* e come tali spinge a non addentrarsi nella modifica di componenti interne al fine di aumentare l'omogeneità tra i moduli, quanto piuttosto allo sviluppo di middleware di comunicazione ed eventualmente coordinazione. Quanto possano risultare complessi questi livelli di interazione è invece uno degli aspetti negativi di questo sistema. Ad essi, tra le altre cose, sono demandate, ove necessarie, tutte le politiche di conversione dei formati e dei protocolli; il tutto con costi che rappresentano infine un fattore grandemente rilevante nella valutazione del modello. In quest'ottica, un altro aspetto non trascurabile è il carico di lavoro da svolgere ogni volta che si renda necessario un aggiornamento a uno dei due moduli, tale da rendere obsoleti e inefficienti i meccanismi di interazione tra esso e il resto del sistema. Da ultimo si pone in luce anche la questione della trasparenza nei confronti dell'utente. Non in tutti i contesti questo è importante, però in certi casi può essere rilevante sapere se l'utente ha piena coscienza di entrambi i moduli e soprattutto se la presenza due essi si ripercuote ad alto livello, fino ad influenzare il comportamento dell'utilizzatore.

Non sarà difficile convenire sul fatto che l'architettura appena descritta, sebbene sommaria e parziale, combaci abbastanza con TuCSoN semantico così come esposto al capitolo 4: da una parte vi è il centro di tuple che incorpora un motore prolog e dall'altra c'è Pellet a cui sono delegati i compiti di ragionamento semantico; TuCSoN si occupa del matching sintattico delle tuple e agisce, per così dire, nella sua working memory, mentre Pellet (che attua inferenze sulla base di un modello a triple) necessita di uno strato intermedio per essere acceduto (Jena) e opera all'interno di una sua memoria di lavoro. Poi va detto che la suddivisione appena evidenziata non è trasparente all'utente, che ne ha sempre piena coscienza, dal momento che l'inserimento di tuple semantiche avviene mediante un formato differente. Queste considerazioni torneranno utili in seguito.

Tightly integrated modules

A partire dal modello esposto in figura 6.1, l'evoluzione più naturale verso una tightness maggiore è quello di unificare la working memory sulla quale i due moduli lavorano. Tale casistica rientra evidentemente in quel tipo di accoppiamento che in tabella 6.1 si era definito come *common coupling*: un livello così alto di interconnessione comporta che sulle aree condivise debbano essere applicate severe politiche di coordinazione, al fine di evitare conflitti critici sull'accesso a risorse condivise. Come nel caso

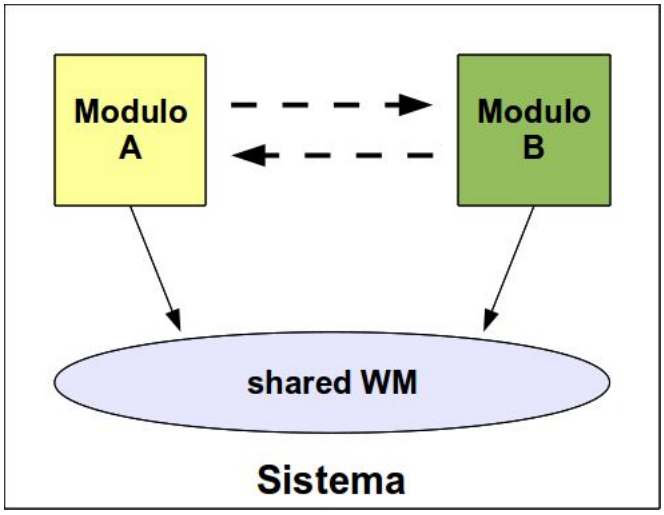


Figura 6.2: Moduli tightly coupled, con working memory condivisa.

del sistema precedente, anche qui può rendersi utile la costruzione di un layer intermedio tra i due moduli e permangono le eventuali problematiche legate all'aggiornamento disgiunto delle due parti. Tuttavia è possibile anche pensare di implementare una politica di comunicazione che sfrutti al massimo la memoria condivisa come medium comunicativo al fine di ridurre il carico computazionale sulle spalle del livello intermedio tra i moduli.

Un altro modello, più particolareggiato è quello illustrato in figura 6.3. In questo caso i due blocchi operativi sono maggiormente integrati, quasi fusi l'uno nell'altro. É da notare come il processo che precede l'attuazione di una configurazione simile possa essere estremamente faticoso, in quanto le funzionalità disgiunte di due moduli eterogenei sono di certo difficil-

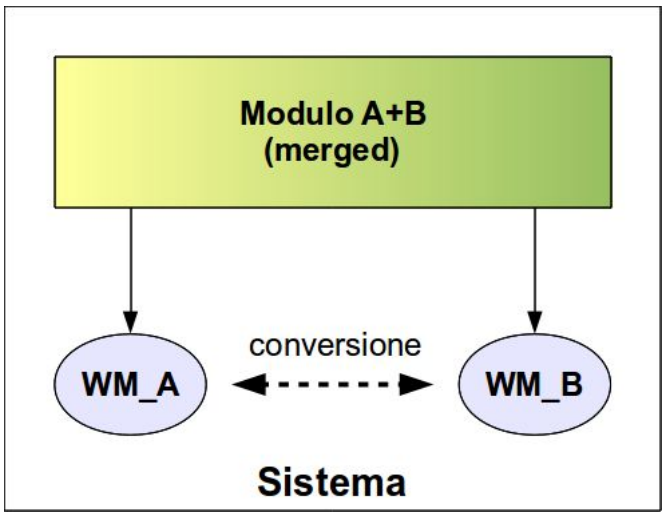


Figura 6.3: Moduli coupled, con working memory separate.

mente unificabili. Una considerazione da fare è che tuttavia in questo caso non sarà più necessario alcun interfacciamento, in quanto ora i due blocchi tratteranno lo stesso tipo di dati e lo faranno mediante l'invocazione di metodi non più esterni ma interni - per intenderci, ora questa dinamiche passano da un problema di accoppiamento a un problema di coesione. Ma perché mantenere due memorie disgiunte? Il motivo potrebbe risiedere nel fatto che i dati contenuti in esse sono differenti, sia sintatticamente che concettualmente. É possibile che i due moduli abbiano preservato la caratteristica, per semplicità o comodità implementativa, di interagire con un determinato formato o che intendano mantenere disgiunte le due working memory per non fondere informazioni appartenenti a due domini applicativi disgiunti.

Integrazione omogenea

Per l'ultimo sistema proposto si fa riferimento a figura 6.4, nella quale viene mostrato un modulo che rispetto a quello visto in figura 6.3 presenta anche una memoria condivisa. Questa situazione presenta una forte omogeneità e con essa anche il vantaggio che, dopo il costo di integrazione che (come detto nel caso visto in 6.2) potrebbe non essere indifferente, i costi computazionali e di mantenimento (aggiornamento) saranno relativamen-

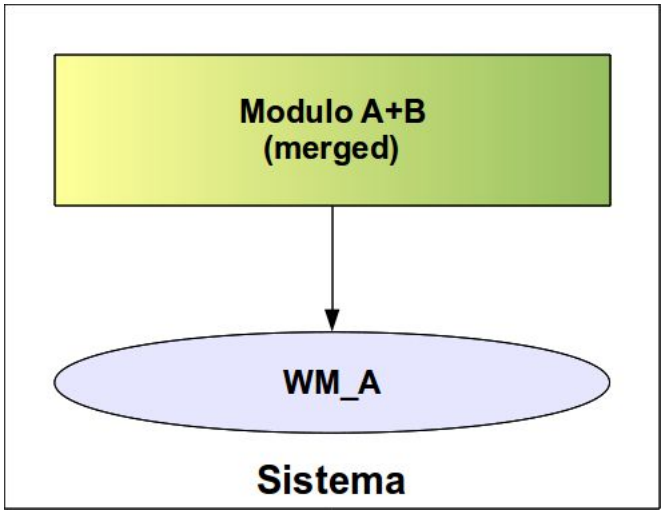


Figura 6.4: Moduli coupled, con working memory condivisa.

te bassi. Tuttavia, ancora una volta non si può prescindere da adeguate politiche di sincronizzazione per l'interfacciamento alla working memory.

É evidente che qualsiasi considerazione di merito sui modelli proposti non può essere svincolate dal contesto applicativo nel quali questi sistemi dovranno essere posti in essere. Nella sezione 6.2.3 si cercherà di capire come applicare quanto visto ora alle architetture semantiche.

6.2.3 Architetture semantiche ibride

Posto che i due moduli citati negli esempi e nelle illustrazioni di sezione 6.2.2 sono generici e possono essere assimilati a qualsivoglia contesto, cerchiamo ora di applicare quanto visto nell'ambito delle architetture semantiche. In questo campo la dicotomia più studiata in letteratura è quella tra i sistemi basati sulle Description Logics e quelli a regole. Nel caso delle logiche descrittive, anche grazie ai progetti sul web semantico (si veda il capitolo 3) sono già stati sviluppati diversi linguaggi e strumenti per la rappresentazione di determinati domini applicativi e per lo sviluppo di un ragionamento su di essi - sui loro individui, concetti e sulle eventuali inconsistenze tra essi. Invece i sistemi *rule-based* vengono tipicamente utilizzati per esprimere applicazioni logiche in termini di regole. La diversità di questi due sistemi giace nel fatto che mentre nei sistemi logici vengono

utilizzate formule logiche prive di side-effect, nei sistemi a regole la fase di matching sulle premesse è poi seguita da una fase di azione sulla working memory, come conseguenza della verifica (o meno) di tali premesse[1]. Per cercare di trarre beneficio da entrambi questi sistemi si è tentata la strada dell'integrazione: alcune strade percorse conducevano a una integrazione più *loose* [15, 3], altre invece a una integrazione più *tight* [25, 47]. In ogni caso il maggiore problema evidenziato era quello di far coesistere un sistema OWA e un sistema CWA [24, 16]. Senza tornare nel merito di quanto già chiarito nella sezione 3.1.3, una delle difficoltà risiede nel fatto che le Closed World Assumption - sfruttate nei sistemi *rule-based* - siano non-monotoniche, ovvero ogni nuovo inserimento nella base di conoscenza può potenzialmente invalidare le deduzioni fatte fino a quel momento. Al contrario i sistemi *DL-based*, con OWA sono molto più dinamici, ma introducono la possibilità di non poter determinare sempre se una data affermazione sia vera o falsa.

Possibili componenti del sistema

Sulla base di quanto visto fin ora, è possibile reperire in letteratura diverse soluzioni che aderiscono al modello logico piuttosto che non a quello a regole. Al primo gruppo aderiscono la maggior parte dei *reasoner* comunemente utilizzati:

- **Fact++** [56].
  Ragionatore basato su tableau e scritto in C++ che supporta ontologie scritte in OWL e OWL2. Si può piegare a diversi utilizzi, come server stand-alone o come ragionatore back-end attraverso OWL-API.

- **HermiT** [50]
  Implementato in Java, è stato il primo ragionatore ad a supportare un nuovo e più efficiente tipo di calcolo detto "hypertableau". Il completo supporto di OWL2 unitamente alle ottime prestazioni

[1] Più precisamente le fase sono tre: confronto, risoluzione dei conflitti e azione sulla working memory

lo rendono un componente di notevole interesse. Rilasciato sotto LGPL.

- **Pellet** [51, 52]
  Ragionatore sviluppato in Java che supporta il ragionamento in OWL DL e anche in OWL2. Anch'esso implementa un algoritmo a tableau ed è in grado di trattare con ontologie con espressività $\mathcal{SROIQ}(\mathcal{D})$. Per prodotti open-source è disponibile con licenza AGPL.

- **KAON2** [28]
  Strumento piuttosto ampio, scritto in Java e adatto a supportare OWL-DL, SWRL e F-Logic. e dotato di interfaccia per interrogazione tramite query SPARQL.

- **RacerPro** [23]
  Ragionatore commerciale basato su LISP, il quale attraverso una procedura a tableau tratta ontologie di tipo $\mathcal{SHIQ}$. Supporta OWL-API e interfacce DIG.

Al contrario, fanno invece parte dei sistemi a regole:

- **Jena** [31, 27]
  Framework che adotta un approccio tendenzialmente *loosely-based*: include un motore a regole ibrido forward/backward chaining ma consente anche di interfacciarsi con differenti ragionatori, in modo tale da poter supportare anche altre modalità di ragionamento.

- **Drools - RETE** [12]
  Business rule management system (BRMS) dotato di motore a regole basato su inferenza forward chaining. Questo sistema a regole di produzione utilizza una sofisticata implementazione dell'algoritmo RETE [18, 19, 20] , adattata e riscritta in Java. Grazie ad esso risulta semplice concepire una conversione (quasi) diretta tra *object* e concetti di una ontologia, cosa che, come vedremo in seguito, potrebbe essere strategicamente rilevante.

Le entità coinvolte

Discussi gli strumenti e le modalità del ragionamento semantico, quali possono essere le varie entità coinvolte in questo processo? Quali possono essere le componenti di basso livello utilizzate per rappresentare la conoscenza? La risposta a queste domande rappresenta la questione nodale attorno alla quale si articolano tutte le altre considerazioni in merito all'architettura di sistema, poiché, si evince facilmente, da ciò dipende il modo di consumare e produrre conoscenza del nostro modello e anche tutti i costi computazionali ad esso collegati.

- **Triple**
  Un primo modo di pensare alle informazioni di una base di conoscenza è quello che viene utilizzato comunemente nel contesto dell'RDF. Alla sezione 3.2.2 si è accennato a esso e si è visto come l'elemento fondante di questo linguaggio sia la tripla, composta da un *soggetto*, da un *predicato* e da un *oggetto*. All'interno del *triplestore* vengono allocate le triple che poi, nel loro insieme, vanno a definire il Grafo RDF, come ad esempio quello di figura 3.8. L'illustrazione 6.5 mostra la composizione dell'architettura di Pellet e si può nota-

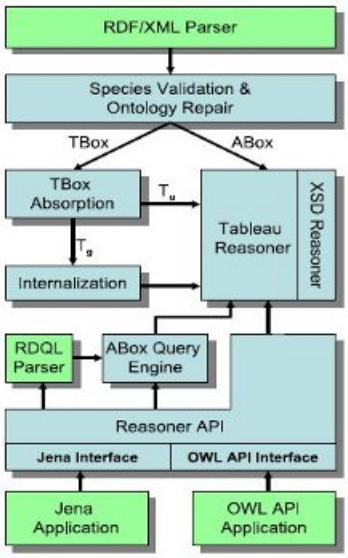


Figura 6.5: Architettura di Pellet.

re come i ragionatori semantici debbano effettuare come prima cosa

un parsing del triplestore per poi poter andare a Riempire la TBox, la ABox e per inferire conoscenza. In generale, le triple RDF sono lo strumento maggiormente utilizzato per scomporre e rappresentare un dominio ontologico da passare a un ragionatore semantico. Questi meccanismi verranno analizzati un po' più in dettaglio alla sezione 6.3.2.

- **Tuple**
  Come visto nel capitolo 4 una tupla semantica in TuCSoN è composta come segue:

  ```
  semantic pluto :  'Dog'(hasOwner :  pippo).
  ```

  Nell'attuale architettura di TuCSoN questo modo di descrivere istanze appartenenti a uno specifico contesto ontologico deve poi passare attraverso a una conversione in query SPARQL - e anche ReSpecT - prima di poter essere passato a Pellet. Questo procedimento, come illustrato in [35], non è esente da costi, ma è la scelta migliore e più logica per rappresentare "istanze semantiche" ad alto livello.

- **Oggetti**
  Gli oggetti di un linguaggio OO possono essere presi in considerazione per rappresentare le entità della nostra architettura semantica. Prendendo il Java ad esempio, a una classe potrebbe essere delegato il compito di rappresentare uno specifico concetto, le cui proprietà vengano descritte attraverso mediante i campi. Ogni istanza viene interpretata (e verificata) coerentemente con quanto asserito nella TBox.

6.2.4 Sistemi esistenti

Un esempio di architettura ibrida con integrazione *loosely coupled* è stato illustrato in [10, 11]. In questo modello vengono sfruttati sia una conoscenza più "operazionale" con il sistema a regole, siano una conoscenza "ontologica" riferita a un determinato dominio. A tele scopo, me-

diante l'interfacciamento offerto dal framework Jena, sono stati integrati i due ragionatori DL Pellet e FuzzyDL [6]: il primo per una gestione standard dell'ontologia messa a disposizione, il secondo per aggiungere di ragionamento su elementi non crisp.

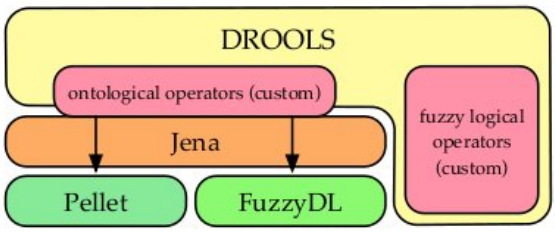


Figura 6.6: Architettura del sistema ibrido.

Un'altro sistema noto è O-Device [32, 33, 34]. Questo è un ragionatore per OWL che sfrutta regole di produzione. L'approccio proposto è quello di trasformare le ontologie in uno schema di classi e oggetti e successivamente quello di applicare le regole ad esso al fine di derivare le semantiche formali del linguaggio.

6.2.5 Verso la fuzziness

A quali sistemi ci possiamo appoggiare nel caso si voglia dotare il sistema di logica fuzzy? Come illustrato in [10] due possibili scelte sono le estensioni dei sistemi a regole Clips[2] e Jess[3] (FuzzyClips[4] e FuzzyJess[5]), ma entrambi i progetti sono stati sospesi. In alternativa esistono sistemi basati sulle logiche descrittive. Il più completo è FuzzyDL [6] che, tra gli altri pregi ha quello di essere implementato in Java. Esso rappresenta una concretizzazione di tutta la ricerca fatta a partire dai primi anni novanta sull'applicazione della vaghezza e dell'imprecisione alle logiche descrittive [64, 30]. Un'altro strumento, anche se non maturo come il precedente, può essere DeLorean[6]

[2] http://clipsrules.sourceforge.net/
[3] http://www.jessrules.com/
[4] http://awesom.eu/ cygal/archives/2010/04/22/fuzzyclips_downloads/index.html
[5] http://www.csie.ntu.edu.tw/sylee/courses/FuzzyJ/FuzzyJess.htm
[6] http://webdiis.unizar.es/~fbobillo/delorean.php

6.3 Verso una implementazione omogenea?

Nelle sezioni precedenti si è illustrato quali possano essere le alternative per integrare due parti disgiunte in un unico modello. Quel discorso è stato fatto da prima in maniera generica, senza cioè riferimenti a sistemi semantici, mentre in seguito sono stati introdotti alcuni esempi presi dalla (abbastanza vasta) letteratura in materia. In questo ambito è possibile pensare ad un sistema completamente omogeneo sia per la working memory che per la parte computazionale?

6.3.1 Tableau su un sistema a regole

Senza entrare troppo nel merito degli algoritmi *tableau-based* [5], è sufficiente sapere che questi stanno alla base della computazione semantica e che in questo contesto svolgono un ruolo centrale: essi controllano la validità della ABox e i concetti, i ruoli e i vincoli su di essi nella TBox. In altre parole verificano la consistenza di una ontologia. Questo genere di algoritmi è caratterizzato da *forward chaining*[7] e tipicamente l'esplorazione dell'albero risolutivo avviene tramite backtracking. Gli algoritmi come RETE utilizzano anch'essi forward chianing (pur senza backtracking) e questo rende i sistemi a regole piuttosto integrabili, almeno in teoria, con un tableau; in linea teorica sono molto più integrabili di un sistema il cui motore è basato su prolog (che adotta backward chaining). Tuttavia sarebbe assurdo pensare di riscrivere il core di TuCSoN con uno strumento che non sia tuProlog. Infatti quest'ultimo si adatta alla perfezione alle esigenze del nostro centro di tuple e i costi di integrazione di un nuovo strumento al suo posto sarebbero nettamente superiori ai benefici ottenuti con una maggiore omogeneità (possibile in linea teorica, ma poi tutta da "implementare") del modello nel suo complesso. Pertanto, dopo alcune valutazione in questo senso, si è optato per mantenere un minor grado di

[7] Il *forward chaining* è un metodo di ragionamento utilizzato spesso nell'ambito dell'intelligenza artificiale. Esso usa regole di inferenza a partire da dati noti fino al raggiungimento del goal. Al contrario, il *backward chaining* parte dai goal (o ipotesi) e cerca di verificarle mediante il raggiungimento dei dati di partenza.

integrazione, sebbene questo comporti necessariamente continuare a tenere separate le working memory e richieda dei costi computazionali per la conversione dei dati tra un sistema e l'altro.

6.3.2 Centri di tuple e ragionatori DL

Quando a suo tempo venne effettuata la scelta architetturale che ha portato ad affiancare Pellet a TuCSoN, si mirò a ottenere il massimo da questi due strumenti: da una parte il riconoscimento semantico col caricamento di una ontologia e il riconoscimento degli elementi che la compongono, dall'altra la possibilità di continuare a sfruttare le grandi capacità di medium di comunicazione e coordinazione (nonché di persistenza dei dati) di un centro di tuple. Già in precedenza, in documenti dove si è analizzato questo modello e le sue potenzialità [35, 37, 38], erano state mostrati pregi e difetti di una architettura di questo tipo, e in particolare il trade-off tra modularità e costi di conversione (e aggiornamento) di componenti piuttosto differenti tra loro. Nella sezione precedente si è accennato a costi di conversione tra componenti diversi e questo è dovuto al fatto che vengono coinvolte e utilizzate entità differenti con differenti caratteristiche e rappresentazioni (si veda 88). La figura 6.7, che è una variante della figura 6.5, mostra come sia composta l'architettura di Pellet e da essa si può partire per compiere alcune considerazioni sulle trasfor-

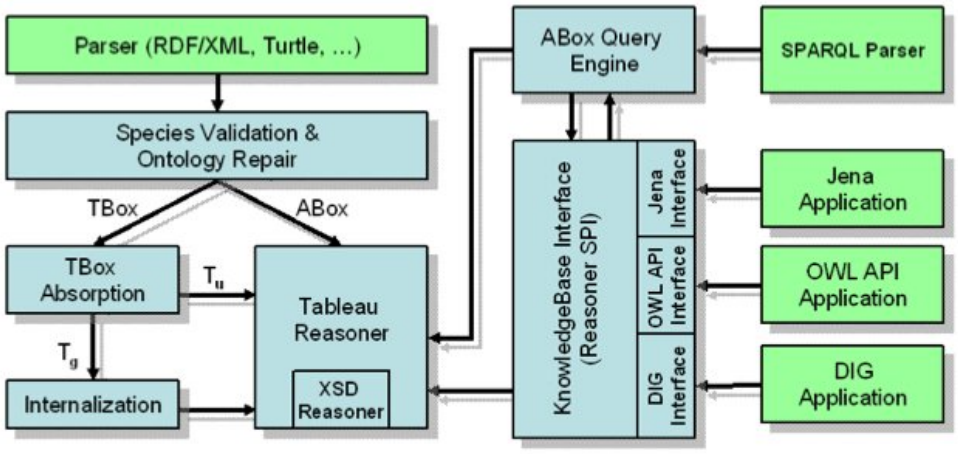


Figura 6.7: Architettura di Pellet (2).

```
public interface IOntology {

  public String getBase();

  public void setBase();

  public boolean assertIndividual(IndividualDescr idescr)
     throws SemanticsException;

  public boolean removeIndividual(String name);

  public IndividualDescr getIndividualDescrThroughJena(String
      name) throws Exception;

  public IndividualDescr
     getIndividualDescrThroughJenaWithExtractor(String name)
     throws PrologException;

  public IndividualDescr getIndividualDescrThroughPellet(
     String name) throws Exception;

  public String getPath();

  public OntModel getModel() ;

  boolean changeIndividual(IndividualDescr iDescr) throws
     SemanticsException;

}
```

Sorgente 6.1: Interfaccia IOntology.java

mazioni da tuple semantiche a elementi di una ABox e dall'ontologia alla composizione della TBox; cerchiamo di capire da dove hanno origine gli oneri computazionali citati in precedenza.

L'inserimento di un individuo in ABox

Valutiamo l'inserimento di un individuo all'interno della ABox. Per fare ciò si fa riferimento all'interfaccia `IOntology`. Come si può vedere anche dal listato 6.1 essa contiene la dichiarazione di diversi metodi per settare od ottenere il riferimento all'ontologia caricata. Inoltre consente di rimuovere o asserire un individuo e in particolare questi metodi vengono poi implementati dalla classe `PelletOntology`. Come si può vedere dal

```
178 public boolean assertIndividual(IndividualDescr idescr)
      throws SemanticsException {
    try {
180   System.out.println(model.listIndividuals().toList().size
        ());
      _IndividualCount++;
182   ...
      ind = PelletReasonerUtils.createIndividual(getBase() +
        name, getBase() + concept, model);
184   ...
      PelletReasonerUtils.fillPropertyValues(ind, getBase(), rf
        , model);
186   ...
      PelletReasonerUtils.incrementalCheckABoxConsistency(model
        );
188   ...
    } catch (SemanticsException ex) {
190   ...
    }
192 return true;
}
```

Sorgente 6.2: Classe PelletOntology.java

sorgente 6.2 che ne riporta un frammento qui vengono utilizzati tre metodi importanti:

i) `PelletReasonerUtils.createIndividual`
per creare un individuo;

ii) `PelletReasonerUtils.fillPropertyValues`
per definire le sue proprietà

iii) `PelletReasonerUtils.incrementalCheckABoxConsistency`
per verificare la consistenza della ABox

Tutti e tre questi metodi richiamano e sfruttano oggetti della libreria Jena, i quali a loro volta accedono all'interfaccia della base di conoscenza di Pellet. É interessante notare come il primo dei tre (`createIndividual`, riquadro 6.3) debba richiedere un "check" per verificare l'esistenza del concetto relativo all'individuo nella TBox di Pellet. Naturalmente questa verifica si rende indispensabile in quanto, come detto alla sezione 4.2.6, la TBox deve mantenersi rigorosamente CWA, ovvero tollerare solo elementi il cui

```
48 public static Individual createIndividual(String name, String
       concept, OntModel model) throws SemanticsException {

50   if (name.equals(""))
       throw new SemanticsException();
52
   OntClass c = PelletReasonerUtils.checkConceptExistence(
       model, concept);
54
   return model.createIndividual(name, c);
56 }
```

Sorgente 6.3: Classe PelletReasonerUtils - metodo createIndividual

```
   public static void incrementalCheckABoxConsistency(OntModel
       model) throws SemanticsException {
214
   PelletInfGraph pellet = (PelletInfGraph) model.getGraph();
216  boolean consistent = pellet.getKB().isConsistent();

218  if(!consistent)
       throw new SemanticsException("Inconsistent ontology");
220 }
```

Sorgente 6.4: Classe PelletReasonerUtils - metodo incrementalCheckABoxConsistency

concetto di appartenenza sia stato specificato a priori, onde evitare problemi di consistenza e di coordinazione sulla piattaforma. L'ultimo dei tre metodi (riquadro 6.4) lavora su oggetto di tipo `PelletInfGraph` ottenuto a partire dal modello ontologico attuale reso come grafo. La base di conoscenza espressa attraverso questo oggetto viene presa in esame per verificare se il novo elemento (inserito nelle fasi precedenti abbia generato delle inconsistenze).

Poiché entrambe le operazioni prese in esame prevedono dei controlli di consistenza, si capirà facilmente come anche l'inserimento di un sola tupla semantica possa potenzialmente risultare computazionalmente oneroso; tale costo è destinato a crescere con l'aumentare della base di conoscenza, in quanto una KB più vasta richiederà un tempo maggiore per essere analizzata nella sua interezza.

```
20 ...
public SparqlQuery generateQuery(IConcept template, Var var,
    IndividualDescr individual, IOntology ont) throws
    SemanticsException {
22  String query_string = buildQueryString(template, var,
      individual, ont);
  return new SparqlQuery(query_string);
24 }

26 private static String buildQueryString(IConcept template, Var
     var, IndividualDescr individual, IOntology ont) throws
    SemanticsException {
  StringBuildVisitor visitor = new SPARQLStringBuildVisitor(
      ont);
28  String query = template.accept(visitor, var);

30  String varname = var.getName();
  String individualName = ont.getBase() + individual.getName
      ();
32
  query += "FILTER ( ?" + varname + " = <" + individualName +
       "> ).";
34  query = completeQuery(query);

36  return query;
}
38 ...
```

Sorgente 6.5: Classe SparqlQueryGeneratorImpl.java

Una query SPARQL

Esaminiamo ora la composizione delle interrogazioni SPARQL. L'interfaccia chiave in questo senso è `ISparqlQueryGenerator` che dichiara due metodi `generateQuery`. Questi poi verranno implementati nella classe `SparqlQueryGeneratorImpl`, della quale in 6.5 si è riportato solo un frammento contenente uno dei due metodi pubblici e un metodo privato. Si noterà come in `buildQueryString` venga utilizzato un oggetto di tipo `SPARQLStringBuildVisitor` (sorgente 6.6) del quale, attraverso una `accept` su `IConcept`, viene utilizzato il metodo `visit`. Anche in questo caso, viene effettuato un controllo sull'esistenza dei concetti coinvolti, incrementando in questo modo il tempo di esecuzione. Tuttavia si rileva che anche in questo caso, come già nei precedenti, tale verifica è del tutto

```
74 @Override
   public String visit(ConceptName concept, Var var) throws
       SemanticsException {
76   // CONCEPT EXISTENCE CHECK
     String conceptName = ont.getBase() + concept.getConceptName
         ();
78   PelletReasonerUtils.checkConceptExistence(model,
         conceptName);

80   String s = concatAll(new String[] { "\n ?", var.getName(),
         " ", RDF_TYPE, " <", conceptName, "> " });

82   return s;
   }
```

Sorgente 6.6: Classe SPARQLStringBuildVisitor - metodo visit

necessaria se si vuole agire in coerenza al modello descritto e considerare valida l'assunzione di Closed World per la TBox.

Un ontologia come grafo

Tra tutti gli oggetti delle librerie Jena, sicuramente `Graph` ricopre un ruolo importante perché svolge il ruolo centrale di rappresentazione del Grafo RDF. Come visto anche in 3.2.2, questi grafi strutturano l'informazione semantica come una intelaiatura di nodi e connessioni composti a triple: soggetto, predicato, oggetto. In buona sostanza un grafo è un insieme di frasi in merito a un determinato dominio ontologico, espresse come terne di concetti e ruoli, ciascuno dotato di eventuali attributi. In Pellet, come si vedrà, la distinzione tra soggetto, predicato e oggetto rimane relegata a un piano meramente teorico, poiché entrambi questi elementi vengono di fatto rappresentati tutti allo stesso modo, ovvero come oggetto di tipo `Node`. Come si può vedere dal riquadro 6.7, la verifica sulla presenza nel grafico di una determinata tripla avviene col metodo `contains`, il quale può avere come input sia tre oggetti `Node`, sia un oggetto `Triple`.

Un oggetto preposto all'inserimento di una tripla all'interno di un determinato grafico è `TripleAdder` delle librerie di Pellet che viene riportato in 6.8.

```
public interface Graph  extends GraphAdd {

   /**
   *  An immutable empty graph.
   */
  public static final Graph emptyGraph = new GraphBase() {
    public ExtendedIterator<Triple> graphBaseFind(
        TripleMatch tm ) {
      return Triple.None;
    }
  };

  boolean dependsOn( Graph other );

  QueryHandler queryHandler();

  TransactionHandler getTransactionHandler();

  BulkUpdateHandler getBulkUpdateHandler();

  Capabilities getCapabilities();

  GraphEventManager getEventManager();

  GraphStatisticsHandler getStatisticsHandler();

  Reifier getReifier();

  PrefixMapping getPrefixMapping();

  void delete(Triple t) throws DeleteDeniedException;

  ExtendedIterator<Triple> find(TripleMatch m);

  ExtendedIterator<Triple> find(Node s,Node p,Node o);

  boolean isIsomorphicWith(Graph g);

  boolean contains( Node s, Node p, Node o );

  boolean contains( Triple t );

  void close();

  boolean isEmpty();

  int size();

  boolean isClosed();
}
```

Sorgente 6.7: Interfaccia Graph.java

```
34 public class TripleAdder {
   public static void add(Graph graph, Node s, Node p, Node o)
       {
36   graph.add( Triple.create( s, p, o ) );
   }
38
   public static void add(Graph graph, Node s, Resource p,
       Node o) {
40   add( graph, s, p.asNode(), o );
   }
42
   public static void add(Graph graph, Node s, Resource p,
       RDFNode o) {
44   add( graph, s, p.asNode(), o.asNode() );
   }
46
   public static void add(Graph graph, Resource s, Resource p,
       RDFNode o) {
48   add( graph, s.asNode(), p.asNode(), o.asNode() );
   }
50 }
```

Sorgente 6.8: Classe TripleAdder.java

Nodi e Risorse

Dal riquadro 6.8 si può notare che l'inserimento di elementi nel grafico sia piuttosto flessibile, ovvero venga garantito anche per elementi formalmente eterogenei, ma che siano in sostanza la rappresentazione di informazioni coerenti con quelle che dovrebbero comparire sul nodo di un albero. In questo senso vengono presi in ingresso sia `Node`, che `RDFNode` o `Resource` (che è una estensione di `RDFNode`). `Resource` è una interfaccia che rappresenta una generica risorsa RDF, che presenta metodi per ottenere o aggiungere risorse.

Dai nodi alle triple

Nelle librerie Jena, come detto, esiste la classe `Triple`, che rappresenta i mattoni indispensabili per la costruzione dei grafi. Dal codice 6.10, che riporta il costruttore di questa classe, si può notare come l'oggetto "triple" quando viene istanziato necessita dell'inserimento dei tre nodi che costituiscono la tripla, ciascuno denominato rispettivamente `s`, `p` e `o`. I valori di

```
92    ...
      public boolean hasURI( String uri );
94
      public String getURI();
96
      public String getNameSpace();
98    ...
      public Statement getProperty( Property p );
100   ...
      public StmtIterator listProperties();
102
      public Resource addLiteral( Property p, boolean o );
104   ...
      public Resource addProperty( Property p, String o );
106   ...
      public boolean hasProperty( Property p );
108   ...
      public boolean hasLiteral( Property p, boolean o );
110
      public Resource removeProperties();
112   ...
```

Sorgente 6.9: Interfaccia Resource.java (stralci)

```
22  public class Triple implements TripleMatch {
      private final Node subj, pred, obj;
24
      public Triple( Node s, Node p, Node o ) {
26      if (s == null) throw new UnsupportedOperationException( "
            subject cannot be null" );
        if (p == null) throw new UnsupportedOperationException( "
            predicate cannot be null" );
28      if (o == null) throw new UnsupportedOperationException( "
            object cannot be null" );
        subj = s;
30      pred = p;
        obj = o;
32    }
      ...
34  }
```

Sorgente 6.10: Classe Triple.java (campi e costruttore)

```
public EdgeList getRNeighborEdges(Role r) {
678   EdgeList neighbors = outEdges.getEdges( r );

680   Role invR = r.getInverse();
      // inverse of datatype properties is not defined
682   if( invR != null )
        neighbors.addEdgeList( inEdges.getEdges( invR ) );
684
      return neighbors;
686 }
```

Sorgente 6.11: Classe Individual.java - metodo getRNeighborEdges

queste variabili verranno poi associati ai tre campi privati soggetto (`subj`), predicato (`pred`) e oggetto (`obj`). Si nota come non venga tollerato il mancato inserimento in input di anche uno solo dei tre Nodi necessari a comporre la tripla. Tuttavia non è sempre possibile (né comodo) poter istanziare una tripla a partire da tre nodi, così, attraverso la classe `TripleAdder` delle librerie Pellet è possibile convertire eventuali input da `Resource` o `RDFNode` in `Node` e aggiungerla direttamente a uno specifico grafo.

La rappresentazione degli individui e dei ruoli

All'interno di Pellet la classe `Individual` estende la classe `Node`. Gli oggetti di questa classe vengono generati in relazione ad una specifica ABox e, più nello specifico, all'interno di questa sono collegati ad altri individui che compongono l'insieme dei loro *edges*, divisi a loro volta tra *parent* e *successors*. Per favorire l'esplorazione di questo albero vengono forniti dei metodi come `getRSuccessors`, `getParent`, `getEdgesTo`, `getDepth`, `isLeaf` (per verificare che non abbia figli) o `getRNeighborEdges`, per citarne alcuni. L'ultimo è riportato nel riquadro 6.11.

Per collegare tra loro gli `Individual` esiste la classe `Role`. I ruoli possono essere derivare o essere derivati da altri ruoli (*subRole* o *superRole*) oppure esserne equivalenti. Inoltre è possibile stabilire se i ruoli siano transitivi, piuttosto che non simmetrici o riflessivi.

Capitolo 7

Conclusioni

Questo lavoro è stato centrato sul centro di tuple TuCSoN e sulla sua architettura. Per comprenderla a fondo si è iniziato ad analizzare la storia degli spazi di tuple a partire dal modello Linda, fino ad arrivare alle sue estensioni; in particolare si è considerata l'estensione semantica di questo modello. Dopo aver dato uno sguardo alle logiche descrittive e alle altre tecnologie utili per descrivere un dominio ontologico ci si è concentrati sull'impiego di queste all'interno dell'attuale architettura di TuCSoN, riesaminando le scelte progettuali che hanno portato il sistema fino allo stato attuale. Infine, nel capitolo 5 si è studiato il concetto di *fuzziness*. Nel sesto capitolo si è cercato di compiere un analisi un po' più approfondita sulle possibili architetture alternative a quella attuale, evidenziandone i pregi così come di difetti. Appurato che il un sistema con un elevato *coupling* tra i suoi componenti presenta in linea teorica, rispetto a quello presente, dei vantaggi in termini di costi computazionali, si sono osservati brevemente i principali sistemi che compongono lo scenario attuale nel contesto del ragionamento semantico.

A questo punto ci si è interrogati sulle varie opzioni presentate ed è stato piuttosto evidente come TuCSoN semantico, sebbene non rappresenti la migliore alternativa per quanto riguarda l'efficienza e l'ottimizzazione, riesce a garantire una ottima espressività grazie all'utilizzo di ontologie owl DL e alla capacità di sfruttare queste ultime fornita da Pellet. Inoltre il vero punto di forza di questo sistema e della sua architettura for-

temente eterogenea sta nella capacità di sfruttare appieno la dualità tra tuple semantiche e tuple sintattiche. Questo aspetto è cruciale in quanto garantisce una grande flessibilità e la possibilità di poter approfittare del matching sintattico come mezzo aggiuntivo di comunicazione e/o coordinazione. Inoltre alla fine del capitolo 6 si è visto come i costi di un'operazione semantica siano potenzialmente molto più elevati di quelli su una tupla sintattica, specie nel caso di rimozione (piuttosto che di inserimento) in uno spazio molto affollato di elementi. Così, dopo aver osservato e compreso alcune delle strutture che costituiscono le librerie di Pellet, si è potuto apprezzare quali siano le ragioni implementative che stanno dietro ai costi sopracitati, evidenziandone l'imprescindibilità.

Sviluppi futuri

A partire da quanto visto, con la premessa che una architettura loosely coupled come quella attuale presenta più vantaggi che svantaggi, una possibile estensione dell'attuale sistema potrebbe prevedere l'inserimento della fuzzyness, così come illustrata nel capitolo 5. Già in [35] si era messo in luce come FuzzyDL potesse essere il candidato ideale a tale scopo. Per consentire questa integrazione, come suggerito alla sezione 5.3, sarà necessario anche garantire il supporto ad una ontologia di tipo fuzzy, che dovrà essere integrata tanto nel sistema quanto negli attori vi accedono.

Un'altra possibile estensione potrebbe essere ottenuta col proseguimento sulla via della distribuzione del centro di tuple. Al momento ogni istanza del CT possiede una working memory non condivisa; in altre parole tutti gli elementi della ABox del centro di tuple sono contenuti esclusivamente in quello stesso centro di tuple. Tuttavia si potrebbe immaginare un sistema all'interno del quale avvenga una vera e propria condivisione della ABox e degli elementi (e relative connessioni) che la compongono. Naturalmente in un sistema del genere, non sarebbero distribuiti solo i dati, ma anche il ragionamento stesso dovrà essere concepito come tale e suddiviso sui vari CT che comporranno il macro-sistema. E ovviamente il vero vantaggio sarebbe proprio questo.

Bibliografia

[1] aliCE Research Group. Respect home. `http://alice.unibo.it/xwiki/bin/view/ReSpecT/`.

[2] aliCE Research Group. Tucson home. `http://alice.unibo.it/xwiki/bin/view/TuCSoN/`.

[3] Grigoris Antoniou. Nonmonotonic rule systems on top of ontology layers. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 394–398, London, UK, 2002. Springer-Verlag.

[4] Franz Baader and Werner Nutt. *The description logic handbook*. Cambridge University Press New York, 2003.

[5] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69:5–40, 2001.

[6] Fernando Bobillo and Umberto Straccia. fuzzyDL: An expressive fuzzy description logic reasoner. In *IEEE International Conference on Fuzzy Systems, 2008. FUZZ-IEEE 2008.(IEEE World Congress on Computational Intelligence)*, pages 923–930, 2008.

[7] Fernando Bobillo and Umberto Straccia. An owl ontology for fuzzy owl 2. In *Proceedings of the 18th International Symposium on Foundations of Intelligent Systems*, ISMIS '09, pages 151–160, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Fernando Bobillo and Umberto Straccia. Fuzzy ontology representation using owl 2. *Int. J. Approx. Reasoning*, 52:1073–1094, October 2011.

[9] Ronald J. Brachman and James G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[10] Stefano Bragaglia, Federico Chesani, Anna Ciampolini, Paola Mello, Marco Montali, and Davide Sottara. An hybrid architecture integrating forward rules with fuzzy ontological reasoning. In *HAIS (1)'10*, pages 438–445, 2010.

[11] Stefano Bragaglia, Federico Chesani, Paola Mello, and Davide Sottara. A rule-based implementation of fuzzy tableau reasoning. In *Proceedings of the 2010 international conference on Semantic web rules*, RuleML'10, pages 35–49, Berlin, Heidelberg, 2010. Springer-Verlag.

[12] JBoss Community. Drools - businness logic integration platform. `http://www.jboss.org/drools/`.

[13] Edsger W. Dijkstra. *Cooperating sequential processes*, pages 65–138. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[14] D. Dubois and H. Prade. *Fuzzy sets and systems - Theory and applications*. Academic press, New York, 1980.

[15] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artif. Intell.*, 172:1495–1539, August 2008.

[16] Thomas Eiter, Giovambattista Ianni, Axel Polleres, Roman Schindlauer, and Hans Tompits. Reasoning with rules and ontologies. In *In Reasoning Web 2006*, pages 93–127. Springer, 2006.

[17] Dieter Fensel. Triple-space computing: Semantic web services based on persistent publication of information. *Intelligence in Communication Systems*, pages 43–53, 2004.

[18] Charles Forgy. A network match routine for production systems. *Working Paper*, 1974.

[19] Charles Forgy. *On the efficient implementation of production systems.* PhD thesis, Carnegie-Mellon University, 1979.

[20] Charles Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[21] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, January 1985.

[22] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35:97–107, February 1992.

[23] Volker Haarslev and Ralf Möller. Racer: An owl reasoning agent for the semantic web. *Proceedings of the International Workshop on Applications Products and Services of Webbased Support Systems*, page 91–95, 2003.

[24] Ian Horrocks, Boris Motik, Riccardo Rosati, and Ulrike Sattler. Can OWL and logic programming live together happily ever after? pages 501–514, 2006.

[25] Ian Horrocks and Peter F. Patel-Schneider. A proposal for an owl rules language. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 723–731, New York, NY, USA, 2004. ACM.

[26] Lyndon j. b. Nixon, Elena Simperl, Reto Krummenacher, and Francisco Martin-recuerda. Tuplespace-based computing for the semantic web: A survey of the state-of-the-art. *Knowl. Eng. Rev.*, 23:181–212, June 2008.

[27] Jena. Jena – a semantic web framework for java. `http://jena.sourceforge.net/`.

[28] KAON. Kaon2. `http://kaon2.semanticweb.org/`.

[29] Thomas Lukasiewicz and Umberto Straccia. Managing uncertainty and vagueness in description logics for the semantic web. *Web Semant.*, 6:291–308, November 2008.

[30] Thomas Lukasiewicz and Umberto Straccia. Managing uncertainty and vagueness in description logics for the semantic web. *Web Semant.*, 6:291–308, November 2008.

[31] Brian McBride. Jena: Implementing the rdf model and syntax specification. In *Proceedings of the 2nd International Workshop on the Semantic Web.*, Hongkong, May 2001.

[32] Georgios Meditskos and Nick Bassiliades. Towards an object-oriented reasoning system for owl. In *OWLED*, 2005.

[33] Georgios Meditskos and Nick Bassiliades. O-device: An object-oriented knowledge base system for owl ontologies. In *Proc. Fourth Hellenic Conf. Artificial Intelligence*, pages 256–266, 2006.

[34] Georgios Meditskos and Nick Bassiliades. A rule-based object-oriented OWL reasoner. *IEEE Transactions on Knowledge and Data Engineering*, 20(3):397–410, 2008.

[35] Elena Nardini. *Semantic Coordination Through Programmable Tuple Spaces*. PhD thesis, Università di Bologna, 2011.

[36] Elena Nardini, Andrea Omicini, and Mirko Viroli. Description spaces with fuzziness. In Mathew J. Palakal, Chih-Cheng Hung, William Chu, and W. Eric Wong, editors, *26th Annual ACM Symposium on Applied Computing (SAC 2011)*, volume II: Artificial Intelligence & Agents, Information Systems, and Software Development, pages 869–876, Tunghai University, TaiChung, Taiwan, 21–25 March 2011. ACM.

[37] Elena Nardini, Andrea Omicini, and Mirko Viroli. Semantic tuple centres. *Science of Computer Programming*, January 2011.

[38] Elena Nardini, Mirko Viroli, and Emanuele Panzavolta. Coordination in open and dynamic environments with tucson semantic tuple centres. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2037–2044, New York, NY, USA, 2010. ACM.

[39] Lyndon J. B. Nixon, Elena Simperl, Reto Krummenacher, and Francisco Martín-Recuerda. Tuplespace-based computing for the Semantic Web: A survey of the state-of-the-art. *The Knowledge Engineering Review*, 23(2):181–212, 2008.

[40] Andrea Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

[41] Andrea Omicini and Enrico Denti. Formal ReSpecT. *Electronic Notes in Theoretical Computer Science*, 48:179–196, June 2001. Declarative Programming – Selected Papers from AGP 2000, La Habana, Cuba, 4–6 December 2000.

[42] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, November 2001.

[43] Andrea Omicini and Franco Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, 1999.

[44] Emanuele Panzavolta. *Coordinazione semantica in TuCSoN*. PhD thesis, Università di Bologna, 2009.

[45] R.S. Pressman. *Software engineering: a practitioner's approach*. Number v. 1 in McGraw-Hill series in computer science. McGraw Hill, 2001.

[46] A. Ricci. Tucson guide, version 1.4.5, November 2006.

[47] Riccardo Rosati. On the decidability and complexity of integrating ontologies and rules. *Web Semant.*, 3:61–73, July 2005.

[48] Jean J. Saade. A unifying approach to defuzzification and comparison of the outputs of fuzzy controllers. *IEEE Transactions on Fuzzy Systems*, 4(3):227–237, August 1996.

[49] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48:1–26, 1991.

[50] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In Alan Ruttenberg, Ulrile Sattler, and Cathy Dolbear, editors, *Proc. of the 5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008 EU)*, Karlsruhe, Germany, October 26–27 2008.

[51] Sirin, B Parsia, BC Grau, A Kalyanpur, and Y Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, June 2007.

[52] Evren Sirin and Bijan Parsia. Pellet: An owl dl reasoner. In *Description Logics*, 2004.

[53] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Syst. J.*, 13:115–139, June 1974.

[54] Giorgos Stoilos, Giorgos B. Stamou, Vassilis Tzouvaras, Jeff Z. Pan, and Ian Horrocks. Fuzzy OWL: Uncertainty and the semantic web. In Bernardo Cuenca Grau, Ian Horrocks, Bijan Parsia, and Peter F. Patel-Schneider, editors, *OWLED*, volume 188 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2005.

[55] Robert Tolksdorf, Lyndon Nixon, and Elena Simperl. Towards a tuplespace-based middleware for the semantic web. *Web Intelli. and Agent Sys.*, 6:235–251, August 2008.

[56] D. Tsarkov and I. Horrocks. Fact++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[57] W3C. Web services activity. `http://www.w3.org/2002/ws/`.

[58] W3C. Owl web ontology language. `http://www.w3.org/TR/owl-features/`, February 2004.

[59] W3C. Owl web ontology language - use cases and requirements. `http://www.w3.org/TR/webont-req/`, February 2004.

[60] W3C. Rdf primer. `http://www.w3.org/TR/rdf-syntax/`, February 2004.

[61] W3C. Rdf vocabulary description language 1.0: Rdf schema. `http://www.w3.org/TR/rdf-schema/`, February 2004.

[62] W3C. Owl 2 web ontology language. `http://www.w3.org/TR/owl2-overview/`, October 2009.

[63] Ronald R. Yager and Dimitar Filev. On the issue of defuzzification and selection based on a fuzzy set. *Fuzzy Sets and Systems*, 55(3):255 – 271, 1993.

[64] John Yen. Generalizing term subsumption languages to fuzzy logic. In *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 1*, pages 472–477, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[65] L.A. Zadeh. Fuzzy sets. *Information Control*, 8:338–353, 1965.

[66] Lotfi A. Zadeh. The concept of a linguistic variable and its application to approximate reasoning - i. *Inf. Sci.*, 8(3):199–249, 1975.