POLITECNICO DI MILANO
Corso di Laurea in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Progetto e sviluppo di un reasoner per alcune famiglie di logiche descrittive

Relatore: Prof. Marco Colombetti
Correlatore: Ing. Mario Arrigoni Neri

Tesi di Laurea di:
Davide Eynard, matricola 624151

Anno Accademico 2004-2005

Ai miei genitori e ad Elena,
che hanno avuto la pazienza
(e la longevità)
di aspettare.

Sommario

Le *logiche descrittive* sono un una famiglia di formalismi utilizzati per la *rappresentazione della conoscenza*, recentemente oggetto di grande interesse anche per la loro applicazione nel campo del Web Semantico. Fra le diverse operazioni che è possibile eseguire su una base di conoscenze rappresentata da una logica descrittiva, quella del ragionamento riveste un ruolo particolarmente importante, poiché grazie ad essa è possibile ricavare, per inferenza, nuove conoscenze partendo da quelle di cui già si dispone.

Lo scopo della tesi è lo sviluppo di un motore d'inferenza (o *reasoner*) per le logiche descrittive. Per la sua creazione sono stati tenuti presenti i programmi attualmente disponibili e si è cercato non solo di replicare le loro migliori caratteristiche, ma anche di arricchire il reasoner con contenuti e funzioni originali.

I risultati ottenuti sono stati soddisfacenti: il programma, che è stato chiamato JODIE (acronimo di "a Java OWL-DL Inference Engine"), implementa correttamente gli algoritmi di tableaux per diverse famiglie di logiche (da $\mathcal{ALC}$ a $\mathcal{SHIF}$), insieme ad alcune delle tecniche di ottimizzazione più diffuse; una delle sue caratteristiche principali è la modularità, grazie alla quale è possibile aggiungere in modo semplice il supporto per nuove logiche; inoltre, tramite un'interfaccia grafica semplice e ricca di contenuti, JODIE mette a disposizione dell'utente tutte le informazioni relative alle comunicazioni effettuate con altri programmi (tramite i protocolli HTTP e DIG), alla costruzione della base di conoscenze e al processo di ragionamento; infine, grazie al programma esterno GraphViz, esso consente di seguire anche graficamente il funzionamento degli algoritmi implementati.

Ringraziamenti

Ringrazio, innanzitutto, la mia famiglia ed Elena, per essermi stati sempre vicini, per aver avuto la pazienza di aspettare così tanto e di sopportare i miei momenti di *black out* universitario, emotivo o cerebrale in genere. Ringrazio il prof. Marco Colombetti e l'ing. Mario Arrigoni Neri, per avermi affidato un progetto di tesi veramente *bello* e per avermi aiutato, non solo tramite consigli ma anche con il loro appoggio morale, a portarlo a termine. Ringrazio l'ing. Matteo Matteucci, che mi ha offerto la disponibilità dell'AIRLab, il prof. Andrea Bonarini e tutti gli amici conosciuti al laboratorio con cui ho scambiato idee, consigli tecnici o semplicemente un po' di chiacchiere. Ringrazio il prof. Daniele Turi e il prof. Ian Horrocks dell'Università di Manchester per aver risposto con rapidità, pazienza ma soprattutto tanta simpatia alle mie e-mail. Ringrazio tutti i capi e i colleghi che ho avuto in questi anni, perchè ciò che ho imparato da loro mi è tornato utile anche in questa occasione e, sicuramente, mi aiuterà anche in futuro. Ringrazio i miei amici, quelli che ci sono sempre stati e che ci saranno sempre e quelli che ho perso di vista, ma che sono stati ugualmente importanti durante i miei lunghi anni di università.

Per ultimo, ma di certo non per importanza, ringrazio il prof. Marco Somalvico, che ha risvegliato il mio interesse per l'Intelligenza Artificiale e senza il quale forse mi sarei laureato ugualmente, ma sicuramente non con una tesi di questo tipo.

Grazie a tutti,

Indice

Sommario **I**

Ringraziamenti **III**

1 Introduzione **1**
1.1 Obiettivi e motivazioni 1
1.2 Contenuti originali 4
1.3 Struttura della tesi 5

2 Gli algoritmi di tableaux **9**
2.1 Introduzione alle logiche descrittive 9
2.2 Le famiglie di logiche descrittive 12
2.3 Gli algoritmi di tableaux 13
2.4 $\mathcal{ALC}$ 15
2.5 $\mathcal{ALCH}_{R^+}$ ($\mathcal{SH}$) 17
2.6 $\mathcal{SHF}$ 25
2.7 $\mathcal{SI}$ 27
2.8 $\mathcal{SHI}$ 30
2.9 $\mathcal{SHIF}$ 31

3 Struttura del reasoner **35**
3.1 Descrizione generale 35
3.2 HTTP e DIG 40
3.3 La base di conoscenze 42
3.4 La classe Tableaux 45
3.5 L'interfaccia grafica 48
3.6 I moduli Core, Log e Config 49

4 Strategie di ragionamento **55**
4.1 Strategie di ragionamento 55
4.2 $\mathcal{ALC}$ 58

4.3 $\mathcal{ALCF}$ 59
4.4 $\mathcal{ALCH}_{R^+}$ ($\mathcal{SH}$) 60
4.5 $\mathcal{SHF}$ 61
4.6 $\mathcal{SI}$ 62
4.7 $\mathcal{SHI}$ 64
4.8 $\mathcal{SHIF}$ 64

5 Ottimizzazioni **67**
5.1 Tipologie di ottimizzazioni 67
5.2 Normalizzazione e semplificazione 69
5.3 Lazy unfolding 71
5.4 Boolean Constraint Propagation 73
5.5 Caching 75
5.6 Depth first search 76
5.7 Ordine di applicazione delle regole 77

6 Risultati sperimentali **81**
6.1 Strumenti utilizzati 81
6.2 Tipologie di test 83
6.3 Risultati 86

7 Conclusioni **91**
7.1 Valutazioni finali 92
7.2 Sviluppi futuri 93

Bibliografia **94**

Capitolo 1

Introduzione

Le *logiche descrittive* (DL) sono un una famiglia di formalismi utilizzati per la *rappresentazione della conoscenza*. Una delle loro caratteristiche è, come suggerito dal loro stesso nome, quella di utilizzare una semantica formale che si appoggia a una logica. Uno dei principali servizi che le DL mettono a disposizione è quello del ragionamento, grazie al quale è possibile inferire delle conoscenze implicite a partire da quelle esplicitamente contenute all'interno di una base di conoscenze. Esse, inoltre, presentano degli schemi inferenziali che ricorrono frequentemente nei sistemi di elaborazione delle informazioni e che sono utilizzati anche dagli esseri umani per strutturare e comprendere il mondo, come ad esempio la classificazione dei concetti e degli individui [Baader et al., 2003].

Per questi motivi, e grazie al fatto che negli ultimi anni sono state sviluppate logiche descrittive molto espressive, recentemente l'interesse nei confronti delle DL è notevolmente aumentato: allo stato attuale, le loro possibili applicazioni vanno dalla costruzione e validazione di ontologie per il Web Semantico ([Haarslev and Möller, 2003b], [Sattler, 2003], [Baader et al., 2002]) ai Web Service ([Baader et al., 2005]), dai database ([Calvanese et al., 1998a], [Calvanese et al., 1998b], [A. Borgida, 2003]) all'elaborazione del linguaggio naturale, fino addirittura all'utilizzo in medicina e bioinformatica (si consultino ad esempio [Horrocks, 2002], [Horrocks, 2005], oppure il sito `http://www.cs.man.ac.uk/~seanb/adl/`).

1.1 Obiettivi e motivazioni

Fra le varie operazioni che è possibile eseguire su una base di conoscenze rappresentata da una logica descrittiva, quella del ragionamento (si veda ad es.

	Renamed ABox and Concept Expression Reasoner (RACER)	cwm	DAMLJessKB	Euler proof mechanism	Java Theorem Prover (JTP)	OWLP	OpenCyc	Pellet	Surnia	Semantic Web Enabling Technologies for Jess (SweetJess)	TRIPLE	F-OWL
Language Support	OWL	RDF	DAML+OIL	RDF	DAML+OIL	OWL	DAML+OIL	OWL	OWL	DAML+OIL	DAML+OIL	OWL
Ontology Tool	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Yes
Data Tool	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	Yes
Web Site Assessment	Medium	Medium	Medium	Medium	Medium	Low	Medium	Medium	Low	Medium	Medium	Medium
Installation Assessment	High	Medium	Low	Medium	High	Low	High	Low	Low	Low	Medium	Low
Documentation Assessment	Medium	Medium	Medium	Medium	Medium	Low	Medium	Medium	Low	Medium	Low	Medium
Open Source	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	No	No	Yes
DAML Funded	No	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Assessment Score / Skip Reason	100	88	Does not support OWL	Other	Does not support OWL	Other	Does not support OWL	Failed to install/run	Failed to install/run	Does not support OWL	Does not support OWL	Failed to install/run

Figura 1.1: Una tabella comparativa fra alcuni dei reasoner attualmente disponibili su Internet.

[Giunchiglia and Sebastiani, 1996], [Calvanese et al., 2001b], o il più recente [Horrocks and Sattler, 2005]) riveste un ruolo particolarmente importante. Come già accennato in precedenza, infatti, grazie ad essa è possibile ricavare, per inferenza, nuove conoscenze partendo da quelle di cui già si dispone. Di conseguenza, la costruzione e l'utilizzo di strumenti per il ragionamento automatico sulle DL (i cosiddetti motori d'inferenza, o *reasoner*) sono stati oggetto, negli ultimi anni, di un discreto interesse. Per questo motivo, l'obiettivo principale di questa tesi è lo sviluppo di un motore d'inferenza per le logiche descrittive.

Allo stato attuale sono disponibili numerosi reasoner, caratterizzati da algoritmi differenti e in grado di accettare diversi standard. Alcuni di essi funzionano come *servizi di ragionamento*, in grado di interagire in rete con diverse applicazioni, mentre altri funzionano come applicazioni a sé stanti o vengono distribuiti sotto forma di librerie pronte da utilizzare all'interno di altri programmi. Un'altra caratteristica che li contraddistingue è la licenza: alcuni di essi sono distribuiti assieme al loro codice sorgente, liberamente modificabili e ridistribuibili, altri sono gratuiti ma non liberi, altri ancora possono essere utilizzati gratuitamente all'interno delle università, ma devono essere acquistati in caso di uso commerciale. La tabella presente all'indirizzo `http://semwebcentral.org/assessment/report?type=category&category=Reasoning` fornisce un panorama generale dei reasoner attualmente in circolazione (v. Figura 1.1).

Come si può notare dalla tabella, in questo momento una delle caratteristiche a cui viene data maggiore importanza per la valutazione di un reasoner è la sua capacità di interpretare il linguaggio OWL (*Web Ontology Language*: si veda ad es. [Horrocks et al., 2003], [Horrocks and Patel-Schneider, 2004a], [Horrocks and Patel-Schneider, 2004b]). Esso, infatti, è uno standard W3C

studiato appositamente per essere in grado di costruire ontologie che descrivano, in generale, il World Wide Web e, più in particolare, il Web Semantico (`http://www.w3.org/2004/OWL/`).

Oltre ad OWL, gli altri standard attualmente più diffusi sono DAML+OIL e DIG: il primo (`http://www.daml.org/2001/03/daml+oil-index.html`) è il precursore di OWL e, nonostante ora risulti obsoleto, gode ancora di una certa diffusione; il secondo (`http://dl-web.man.ac.uk/dig/`) costituisce invece un protocollo standard per le comunicazioni fra i reasoner e i programmi che ne fanno uso ([Haarslev and Möller, 2003c], [Bechhofer et al., 1999] e [Dickinson, 2004]). Naturalmente, un programma che sia in grado di comprendere anche DAML+OIL e DIG è da ritenersi più completo rispetto ad uno che non li supporta.

Per questi motivi Racer viene considerato, fra i reasoner presenti in tabella, quello più completo: esso, infatti, oltre ad essere un progetto maturo e ormai piuttosto stabile, supporta OWL, DAML+OIL e DIG. Esso non ha un'interfaccia utente grafica, ma viene eseguito da linea di comando e opera come servizio di ragionamento ([Haarslev and Möller, 2003d], [Haarslev and Möller, 2001]).

Oltre a Racer vi sono altri reasoner che, a prescindere dal voto o addirittura dalla loro presenza nella tabella di Figura 1.1 meritano di essere menzionati. Primo fra tutti FaCT (**Fa**st **C**lassification of **T**erminologies, [Horrocks, 1998b], [Horrocks, 1998a], oppure sul sito `http://www.cs.man.ac.uk/~horrocks/FaCT/`), è in grado di supportare OWL Lite (una versione ridotta della logica di OWL) e, tramite un servlet Java, anche DIG. Esso è stato originariamente scritto in Common Lisp, ma esiste una nuova versione (chiamata FaCT++) scritta in C++, e viene distribuito con licenza GPL, più libera rispetto a quella di Racer. Con una licenza simile (la MIT License) e scritto in Java, Pellet `http://www.mindswap.org/2003/pellet/index.shtml` è un reasoner più recente ma non meno interessante, che ancora non supporta DIG e la versione completa di OWL DL ma sta crescendo rapidamente e ha già una nutrita schiera di utenti. Entrambi i motori d'inferenza, come anche lo stesso Racer, fanno uso dei cosiddetti *algoritmi di tableaux* per ragionare in modo corretto, completo e relativamente efficiente ([Horrocks, 1997], [Horrocks and Sattler, 1999], [Horrocks et al., 1999b], [Horrocks and Sattler, 2005] [Hladik and Model, 2004]).

Nello stabilire gli obiettivi da raggiungere durante lo svolgimento di questo progetto di tesi, si è deciso di prendere ad esempio le caratteristiche considerate migliori dei reasoner appena descritti. Per questo, il motore di inferenza è stato progettato per rispondere almeno ai seguenti requisiti:

- supporto per OWL DL, o perlomeno per OWL Lite (corrispondenti, rispettivamente, alle logiche $\mathcal{SHOIN}$ e $\mathcal{SHF}$)
- supporto per DIG, per semplificare la comunicazione del reasoner con gli altri programmi
- licenza libera, in modo che tutta la comunità di sviluppatori, utenti, studenti e ricercatori possa giovarne
- utilizzo degli algoritmi di tableaux per il ragionamento, accompagnati da eventuali ottimizzazioni per migliorarne le performance
- utilizzo di un linguaggio di programmazione diffuso e multipiattaforma, per consentirne l'uso a più persone possibile (nella fattispecie si è scelto Java)

Poiché il progetto riguarda un motore d'inferenza per OWL DL in Java, la scelta del suo nome è stata quasi inevitabile: *a **J**ava, **O**WL **D**L **I**nference **E**ngine* è diventato, più semplicemente, *JODIE*.

1.2 Contenuti originali

Oltre a proporsi di raggruppare al suo interno le migliori caratteristiche dei reasoner attualmente in circolazione (operazione, di per sé, già originale), JODIE presenta delle funzioni ancora inedite rispetto agli altri motori d'inferenza.

Logiche espressive

Ispirandosi a recenti paper riguardanti gli algoritmi di tableaux, JODIE è in grado di ragionare su logiche piuttosto espressive, nella fattispecie su $\mathcal{SHIF}$ (un'estensione della logica $\mathcal{SHF}$, corrispondente a OWL Lite, in grado però di gestire anche i ruoli inversi).

HTTP e DIG

Il programma ha un parser DIG e supporta il protocollo HTTP. Queste due caratteristiche combinate, ancora non presenti in diversi reasoner, gli permettono di comunicare in rete con altre applicazioni (e, in particolare, con editor di ontologie che gli forniscono la base di conoscenze su cui operare). Il parser è stato sviluppato per essere sufficientemente modulare da poter essere incluso anche in altri reasoner: in particolare, si è cercato di mantenere

un discreto livello di compatibilità con Pellet, il cui codice sorgente è stato prezioso per lo sviluppo di JODIE.

Interfaccia grafica

Anche se per il funzionamento del reasoner non è un elemento indispensabile, JODIE dispone di un'interfaccia grafica che mette a disposizione dell'utente diverse funzioni addizionali, tra cui una gestione avanzata dei log. Tali strumenti permettono sia al "normale" utente sia al programmatore di acquisire una maggiore conoscenza dell'applicazione, dei protocolli in gioco e sui reasoner in generale.

Impostazione didattica

Al progetto è stata data una forte impostazione didattica. Questo si è tradotto in diverse scelte progettuali, volte a fornire all'utente e allo stesso sviluppatore una maggiore consapevolezza riguardo al funzionamento dell'applicazione. In pratica, questo si è tradotto nello sviluppo di GUI e log, ma anche nella scelta di implementare i diversi algoritmi di tableaux in modo incrementale, sottolineando così sia le differenze sia i punti in comune con quelli che li precedono.

Rappresentazione grafica

JODIE è in grado di esportare la struttura di un tableaux sia in formato XML sia nel meno comune formato DOT. Poichè quest'ultimo può essere interpretato dal programma GraphViz (`http://www.graphviz.org/`) per *disegnare* un grafo, è possibile ottenere una rappresentazione grafica delle strutture generate dagli algoritmi di tableaux. JODIE prevede la possibilità di eseguire automaticamente GraphViz e generare una pagina HTML che contenga, contemporaneamente, l'immagine del grafo e la sua versione XML (v. Figura 1.2). Tuttavia, è possibile anche generare solo i file DOT e utilizzarli in un secondo tempo, creando ad esempio una sequenza di immagini che mostrino l'evoluzione dell'algoritmo di tableaux.

1.3 Struttura della tesi

Nel Capitolo 2, dopo una breve introduzione alle logiche descrittive, vengono illustrate in dettaglio le diverse logiche sulle quali JODIE è in grado di ragionare, partendo dalla più semplice ($\mathcal{ALC}$) fino ad arrivare a quella più espressiva ($\mathcal{SHIF}$). Per ognuna di esse, oltre a una descrizione delle

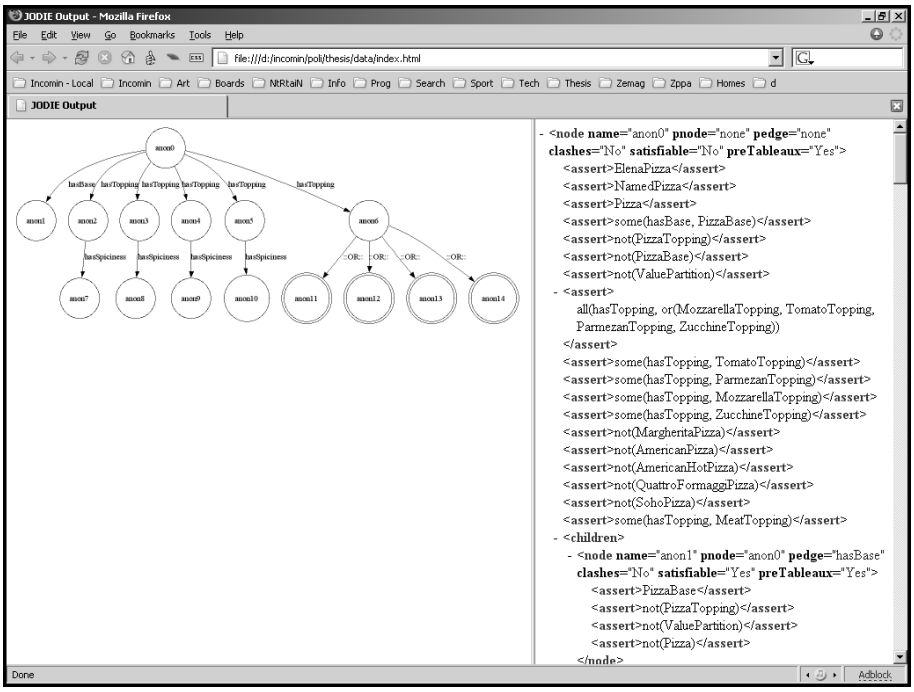

Figura 1.2: L'output di GraphViz insieme alla versione XML di un tableaux.

caratteristiche che la rendono più avanzata rispetto alle precedenti, viene presentato il relativo algoritmo di tableaux.

JODIE è un reasoner complesso e modulare: i suoi componenti svolgono le operazioni più diverse, dalla gestione delle connessioni HTTP al parsing dei messaggi DIG, fino ad arrivare al salvataggio dei dati nella base di conoscenze e alle procedure di ragionamento. Oltre a queste, naturalmente, vi è tutta una serie di operazioni accessorie, ma non meno importanti, come ad esempio la gestione dei log, l'interfaccia grafica e la possibilità di visualizzare graficamente il funzionamento degli algoritmi di tableaux. Con lo scopo di fornire una visione completa del progetto, all'interno del Capitolo 3 viene mostrata la struttura del reasoner, partendo da una descrizione generale fino ad arrivare a quella dei singoli componenti.

Il Capitolo 4 si occupa di descrivere in dettaglio l'aspetto implementativo delle diverse strategie di ragionamento: le strutture dati utilizzate, le funzioni create ad hoc e le scelte adottate per ridurre la complessità spaziale e temporale dell'algoritmo.

Affinché gli algoritmi di tableaux siano in grado di fornire risultati in tempi accettabili, si rende talvolta necessario applicare delle procedure di ottimizzazione. Il Capitolo 5 è dedicato proprio a tali procedure: per ognu-

na di esse si spiegano le motivazioni che hanno portato al suo utilizzo, se ne descrive il funzionamento e si valutano i vantaggi ottenuti dalla sua applicazione.

Il Capitolo 6 contiene i risultati sperimentali relativi ai test eseguiti sul reasoner per verificarne la correttezza e le performance. Ad essi seguono, nel Capitolo 7, le valutazioni finali sul lavoro svolto e alcune proposte riguardanti i possibili sviluppi futuri di JODIE.

Capitolo 2

Gli algoritmi di tableaux

La tecnica di ragionamento adottata all'interno di JODIE è quella, ormai consolidata, degli *algoritmi di tableaux*. Essi si sono rivelati particolarmente utili anche per le logiche descrittive più espressive, in quanto sono completi e, nonostante una buona parte di essi sia caratterizzata da una complessità esponenziale, all'atto pratico sono in grado di fornire risultati in tempi accettabili.

Poiché tali algoritmi si applicano a diverse famiglie di logiche descrittive, la prima parte del capitolo sarà dedicata proprio a tali logiche e alle caratteristiche che differenziano una famiglia dall'altra. La seconda parte, invece, descrive in dettaglio il funzionamento degli algoritmi di tableaux e, logica per logica, le regole che devono essere applicate per portare a termine l'operazione di ragionamento.

2.1 Introduzione alle logiche descrittive

Le logiche descrittive (DL) [Baader et al., 2003] [Baader and Sattler, 2001] [Calvanese et al., 2001a] [Haarslev and Möller, 2003a] sono una famiglia di formalismi per la rappresentazione della conoscenza basati sulla logica. Esse costituiscono ormai un metodo ben strutturato e diffuso per rappresentare ed effettuare ragionamenti sulle conoscenze relative a un particolare campo d'applicazione.

Le nozioni di base all'interno delle logiche descrittive sono quelle corrispondenti ai *concetti* (predicati unari) e ai *ruoli* (relazioni binarie). Concetti e ruoli in forma atomica possono essere combinati, tramite dei *costruttori*, per creare termini complessi. Le varie famiglie di logiche descrittive sono caratterizzate dai costruttori che mettono a disposizione e questi, a loro volta,

determinano la potenza espressiva della DL a cui sono associati (si veda, a questo proposito, la prossima sezione).

In aggiunta ai costruttori, le DL presentano solitamente anche un componente terminologico, chiamato *TBox*. Nella sua forma più semplice, una TBox può essere usata per associare nomi (abbreviazioni) a concetti complessi. Ad esempio, l'espressione

$$\text{DONNA} \equiv \text{PERSONA} \sqcap \text{FEMMINA}$$

è una *definizione terminologica* che consente di definire il termine `DONNA` in funzione dei termini `PERSONA` e `FEMMINA`. Naturalmente, all'interno dell'equazione possono comparire anche dei ruoli: ad esempio,

$$\text{MADRE} \equiv \text{DONNA} \sqcap \exists \text{haFiglio}$$

definisce il termine `MADRE` in quanto `DONNA` che ha almeno un figlio (si noti che, in questo caso, `haFiglio` non è un concetto, bensì un ruolo).

Le definizioni terminologiche sono un caso particolare (quello in cui il termine a sinistra dell'equazione è atomico) di *equivalenze terminologiche*, le quali consentono di esprimere l'equivalenza fra due termini arbitrari. Ad esempio,

$$\text{C} \equiv \text{D}$$

esprime, appunto, l'equivalenza fra i due termini arbitrari `C` e `D`. Un insieme finito di definizioni terminologiche viene chiamato *ontologia*: la sua funzione è quella di assegnare un significato, non ambiguo, ai termini atomici, in base al significato di altri termini. Naturalmente, dato che ogni ontologia è finita, risalendo nella gerarchia dei termini prima o poi è possibile individuarne alcuni privi di una definizione: questi sono considerati *termini primitivi*.

Due termini primitivi solitamente usati in ogni logica sono quelli corrispondenti al *concetto universale* e al *concetto vuoto*: il primo, che denota la totalità degli individui esistenti, viene anche chiamato `TOP` ed è rappresentato dal simbolo $\top$; il secondo, che denota l'insieme vuoto di individui, viene chiamato `BOTTOM` ed è rappresentato dal simbolo $\bot$.

La *semantica* di ruoli e concetti complessi viene definita in base a una *interpretazione* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$. Il dominio $\Delta^{\mathcal{I}}$ di $\mathcal{I}$ è un insieme non vuoto di individui; la funzione di interpretazione $\cdot^{\mathcal{I}}$ mette in corrispondenza ogni concetto con un sottoinsieme di $\Delta^{\mathcal{I}}$ e ogni ruolo con un sottoinsieme di $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, tali che, per tutti i concetti C, D, ruoli R, S, e interi non negativi

Nome del costrutto	Sintassi	Semantica	
concetto atomico	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	$\mathcal{S}$
ruolo atomico	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	
ruolo transitivo	$R \in \mathbf{R}_+$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$	
congiunzione	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	
disgiunzione	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	
negazione	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
quantif. esistenziale	$\exists R.C$	$\{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$	
quantif. universale	$\forall R.C$	$\{x \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ implies } y \in C^{\mathcal{I}}\}$	
gerarchia di ruoli	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{H}$
ruolo inverso	R^-	$\{\langle x, y\rangle \mid \langle y, x\rangle \in R^{\mathcal{I}}\}$	$\mathcal{I}$
cardinalità non qualificate	$\geqslant nR$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \geqslant n\}$	$\mathcal{N}$
	$\leqslant nR$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \leqslant n\}$	
cardinalità qualificate	$\geqslant nR.C$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \geqslant n\}$	$\mathcal{Q}$
	$\leqslant nR.C$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \leqslant n\}$	

Tabella 2.1: Sintassi e semantica delle varie famiglie di logiche descrittive.

n, le proprietà nella Tabella 2.1 siano soddisfatte, dove $\sharp M$ rappresenta la cardinalità di un insieme M.

I sistemi basati sulle logiche descrittive offrono ai propri utenti diverse possibilità per dedurre conoscenza implicita a partire da quella specificata esplicitamente all'interno della TBox. Ad esempio, l'algoritmo di *sussunzione* consente di determinare le relazioni fra sottoconcetti e superconcetti: un concetto D è sussunto da un concetto C rispetto a una TBox T se, in ogni modello di T, ogni istanza di D è anche un'istanza di C. Tale algoritmo può essere utilizzato per calcolare la *tassonomia* di una TBox, cioè la gerarchia di sussunzioni di tutti i concetti introdotti nella TBox. L'algoritmo di *soddisfacibilità*, invece, verifica se un dato concetto potrà mai avere un'istanza.

Definizione 1 *Il concetto C* è sussunto *dal concetto D (e si scrive $C \sqsubseteq D$) se e solo se $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ per ogni interpretazione $\mathcal{I}$; C è* soddisfacibile *se e solo se esiste una interpretazione $\mathcal{I}$ tale che $C^{\mathcal{I}} \neq \emptyset$; infine, C e D sono equivalenti se e solo se $C \sqsubseteq D$ e $D \sqsubseteq C$.*

In presenza dell'operatore di negazione, la sussunzione può essere ricondotta alla soddisfacibilità, e viceversa. Infatti, $C \sqsubseteq D$ se e solo se $C \sqcap \neg D$

è insoddisfacibile; C, d'altra parte, è soddsifacibile se e solo se non vale $C \sqsubseteq P \sqcap \neg P$, dove P è un nome di concetto.

2.2 Le famiglie di logiche descrittive

Come già accennato all'interno di questo capitolo, le logiche descrittive possono essere raggruppate in famiglie a seconda dei costruttori che esse mettono a disposizione. Per ogni costruttore è stata scelta una lettera esplicativa della sua funzione: la combinazione di tali lettere dà origine ai vari nomi delle logiche che verranno trattate in seguito. In particolare:

- la sigla $\mathcal{AL}$ (*attributive language*) è usata per descrivere la logica "capostipite" di queste famiglie, che consente di usare i seguenti termini:
  A (concetto atomico), $\top$, $\bot$, $\neg A$, $C \sqcap D$, $\forall R.C$, $\exists R.\top$
  dove la negazione può essere applicata solamente ai concetti atomici, e solo il concetto `TOP` ($\top$) può essere utilizzato insieme al quantificatore esistenziale;

- la lettera $\mathcal{U}$ (*union*) indica la possibilità di esprimere l'*unione* di concetti:
  $C \sqcup D$;

- la lettera $\mathcal{E}$ (*full existential quantification*) indica la possibilità di associare al quantificatore esistenziale un qualsiasi concetto diverso da `TOP`:
  $\exists R.C$;

- la lettera $\mathcal{C}$ (*complement*) indica la possibilità di applicare la negazione a qualsiasi concetto e non solamente a quelli atomici:
  $\neg C$

A questo punto, si noti che diverse lettere possono essere combinate per dare origine a logiche fra di loro equivalenti dal punto di vista semantico: nella fattispecie, in virtù delle eguaglianze

$$C \sqcup D \equiv \neg(\neg C \sqcap \neg D)$$

e

$$\exists R.C \equiv \neg \forall R.\neg C$$

l'unione e la quantificazione esistenziale completa possono essere espresse utilizzando la negazione (e viceversa), e quindi è possibile chiamare la logica $\mathcal{ALCUE}$, più semplicemente, $\mathcal{ALC}$. Le successive estensioni di $\mathcal{ALC}$ comprendono:

- la lettera $\mathcal{S}$, che viene usata come abbreviazione per $\mathcal{ALC}_{R^+}$. Il pedice R^+ indica la possibilità di utilizzare dei ruoli *transitivi*;
- la lettera $\mathcal{H}$ (da *role Hierarchy*) indica la possibilità di definire relazioni di inclusione fra i ruoli:
  $R \subseteq S$;
- la lettera $\mathcal{O}$ (*One of*) indica la possibilità di definire termini per enumerazione:
  $\{a_1, \ldots, a_n\}$;
- la lettera $\mathcal{I}$ (*Inverse role*) indica la possibilità di definire il *ruolo inverso*:
  R^-;
- la lettera $\mathcal{F}$ (*Functional role*) indica la possibilità di definire ruoli *fuinzionali* (tali, cioè, che ogni elemento del dominio sia in relazione con *al più* un elemento del codominio):
  $\geqslant 1RC$, $\leqslant 1RC$, $= 1RC$;
- la lettera $\mathcal{N}$ (*Number restrictions*) indica la possibilità di definire cardinalità *non qualificate*:
  $\geqslant nR$, $\leqslant nR$, $= n.R$;
- la lettera $\mathcal{Q}$ (*Qualified number restrictions*) indica la possibilità di definire cardinalità *qualificate*:
  $\geqslant nRC$, $\leqslant nRC$, $= nRC$.

2.3 Gli algoritmi di tableaux

Il nome "algoritmi di tableaux" deriva dal fatto che il primo algoritmo di questo tipo (creato da Schmidt-Schauß e Smolka nel 1991, per la DL $\mathcal{ALC}$ [Schmidt-Schauß and Smolka, 1991]), così come quelli creati successivamente per logiche più espressive, poteva essere considerato una specializzazione della tecnica dei tableaux per la logica predicativa del prim'ordine.

L'idea che sta alla base degli algoritmi di tableaux è quella di ricevere in ingresso un concetto C e una gerarchia di ruoli $\mathcal{R}$ e cercare di provare

la soddisfacibilità di C rispetto a $\mathcal{R}$ costruendone un modello. Ciò viene fatto decomponendo sintatticamente C in modo da derivare una serie di condizioni che la struttura del modello dovrà essere in grado di soddisfare. Si consideri, ad esempio, la seguente situazione:

1. ogni modello di C deve, per definizione, contenere un certo individuo x tale che x sia un elemento di $C^{\mathcal{I}}$: si supponga che, in questo caso, C sia della forma $\exists R.D$;

2. come conseguenza di ciò, il modello deve contenere anche un individuo y tale che $\langle x, y\rangle \in R^{\mathcal{I}}$ e y sia un elemento di $D^{\mathcal{I}}$;

3. se D non è atomico, allora proseguendo con la sua decomposizione otterremo altre condizioni;

4. la costruzione del modello fallisce se fra le condizioni da soddisfare è presente un *clash*, cioè una contraddizione ovvia (ad esempio se, dato un concetto C, un individuo z dev'essere contemporaneamente un elemento di C e di $\neg C$).

Gli algoritmi di tableaux sono progettati per terminare sempre e per costruire un modello ogniqualvolta sia possibile: essi, quindi, possono essere usati come procedure decisionali per la soddisfacibilità dei concetti.

All'atto pratico, gli algoritmi di tableaux operano spesso su un grafo a forma di albero (chiamato *completion tree*) che ha una corrispondenza molto stretta con il modello; questo nonostante i modelli possano essere, ad esempio, non finiti (ma rimangono comunque rappresentabili in modo finito) o addirittura non degli alberi (anche se, di solito, mantengono questa struttura). La maggior parte delle volte vengono usati dei grafi dotati di etichette, dove i nodi rappresentano gli individui all'interno del modello e vengono etichettati con un insieme di concetti dei quali sono istanze, mentre gli archi rappresentano relazioni di ruoli fra coppie di individui, e sono etichettati con un insieme di nomi di ruolo.

La decomposizione dei concetti e la costruzione del grafo vengono solitamente effettuate applicando le cosiddette *regole di espansione* del tableaux ai concetti presenti nelle etichette dei nodi. Ognuna di queste regole viene definita in corrispondenza di uno dei costrutti sintattici del linguaggio. Ad esempio, la regola di espansione per la congiunzione fa in modo che C e D vengano aggiunti ad ogni etichetta che già contenga $C \sqcap D$. Per garantire la terminazione, vi sono condizioni aggiuntive che impediscono l'applicazione delle regole se esse non cambiano il grafo o il suo insieme di etichette.

L'unico dei costrutti sintattici per il quale non viene creata una regola *ad hoc* è la negazione: per gestire quest'ultima, infatti, si sceglie solitamente di trasformare tutti i termini in *forma normale negata* (*Negational Normal Form*, NNF). In pratica, la negazione viene spinta all'interno dei termini, attraverso regole di riscrittura come le leggi di De Morgan, finché essa non è applicata ai concetti atomici.

All'interno del processo di espansione compaiono due forme di indeterminismo. Innanzi tutto, molte regole possono essere applicate contemporaneamente e si rende necessario scegliere un ordine per la loro applicazione. Anche se la correttezza di alcuni algoritmi richiede un ordinamento delle regole per priorità, nella maggior parte dei casi questa scelta è irrilevante: se esiste un modello, infatti, esso verrà comunque trovato, indipendentemente dall'ordine di espansione. Tuttavia, come si vedrà più avanti (e in particolare nel Capitolo 5, dedicato alle ottimizzazioni), tale ordine può avere effetti notevoli sull'efficienza dell'algoritmo.

In secondo luogo, alcune regole causano un'espansione non deterministica del grafo; ad esempio, la regola d'espansione per la disgiunzione causa l'aggiunta del concetto C *oppure* del concetto D ad ogni nodo la cui etichetta contenga già $C \sqcup D$. Dal punto di vista della correttezza, la scelta *è* rilevante, poichè una scelta può portare alla costruzione di un modello corretto, mentre un'altra no, e viene solitamente gestita da una ricerca tramite backtracking. Anche se tale tipo di ricerca deve (nel caso peggiore) considerare comunque *tutte* le possibili espansioni, l'ordine in cui queste vengono esplorate può ancora avere un grande effetto sull'efficienza.

2.4 $\mathcal{ALC}$

La logica $\mathcal{ALC}$ è la più semplice e la meno espressiva fra quelle trattate all'interno di questo capitolo. Se, da una parte, essa consente solo la dichiarazione di concetti del tipo $\top$, $\bot$, $C \cap D$, $C \cup D$, $\neg C$, $\forall R.C$ e $\exists R.C$, dall'altra la sua stessa semplicità la rende un ottimo punto di partenza per chi desidera avvicinarsi agli algoritmi di tableaux. Essa, infatti, presenta un algoritmo non troppo complesso, ma già dotato di alcune caratteristiche (come, ad esempio, l'espansione indeterministica della disgiunzione) che ricorrono anche all'interno delle logiche più complicate.

Un algoritmo di tableaux per $\mathcal{ALC}$

L'algoritmo di tableaux utilizza una struttura ad albero per rappresentare il modello che viene costruito: ogni nodo x rappresenta un individuo e viene

regola $\sqcap$: se	1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$ e 2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
allora	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
regola $\sqcup$: se	1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$ e 2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
allora	a. prova $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1\}$ se questo causa un clash ripristina il tableaux e b. prova $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_2\}$
regola $\exists$: se	1. $\exists R.C \in \mathcal{L}(x)$ 2. non c'è alcun y tale che $\mathcal{L}(\langle x, y\rangle) = R$ e $C \in \mathcal{L}(y)$
allora	crea un nuovo nodo y e un nuovo arco $\langle x, y\rangle$ con $\mathcal{L}(y) = \{C\}$ e $\mathcal{L}(\langle x, y\rangle) = R$
regola $\forall$: se	1. $\forall R.C \in \mathcal{L}(x)$ 2. sono presenti degli y tali che $\mathcal{L}(\langle x, y\rangle) = R$ e $C \notin \mathcal{L}(y)$
allora	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

Tabella 2.2: Le regole di espansione per $\mathcal{ALC}$.

etichettato con un insieme $\mathcal{L}(x)$ di termini (concetti semplici o complessi, costruiti con gli operatori della logica $\mathcal{ALC}$) che deve soddisfare:

$$C \in \mathcal{L}(x) \Rightarrow x \in C^{\mathcal{I}}$$

Ogni arco $\langle x, y\rangle$ all'interno dell'albero corrisponde a una coppia di individui all'interno dell'interpretazione e viene etichettato con il nome del ruolo:

$$R = \mathcal{L}(\langle x, y\rangle) \Rightarrow \langle x, y\rangle \in R^{\mathcal{I}}$$

Per determinare la soddisfacibilità di un concetto D, viene inizializzato un albero $\mathbf{T}$ in modo da contenere un singolo nodo x_0, con $\mathcal{L}(x_0) = \{D\}$, e quindi espanso applicando ripetutamente le regole che compaiono nella Tabella 2.2. $\mathbf{T}$ è completamente espanso quando nessuna delle regole può più essere applicata. $\mathbf{T}$ contiene una contraddizione ovvia (o *clash*) quando, per un nodo x e un concetto C, $\bot \in \mathcal{L}(x)$ oppure $\{C, \neg C\} \subseteq \mathcal{L}(x)$.

Al termine della procedura di espansione, un albero $\mathbf{T}$ libero da clash può essere convertito, in modo molto semplice, in un modello che testimonia la soddisfacibilità di D:

$$\begin{aligned}
\Delta^{\mathcal{I}} &= \{x | x \text{ è un nodo di } \mathbf{T}\} \\
\mathrm{CN}^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} | \mathrm{CN} \in \mathcal{L}(x)\} \text{ per tutti i nomi di concetto CN in } D \\
R^{\mathcal{I}} &= \{\langle x, y\rangle | \langle x, y\rangle \text{ è un arco di } \mathbf{T} \text{ e } \mathcal{L}(\langle x, y\rangle) = R\}
\end{aligned}$$

dove le interpretazioni delle diverse espressioni seguono direttamente dalla semantica specificata nella Tabella 2.1 a pagina 11. L'algoritmo di tableaux appena descritto per la logica $\mathcal{ALC}$ è corretto e completo: per una dimostrazione, si rimanda ai testi presenti in letteratura ([Horrocks, 1997], [Schmidt-Schauß and Smolka, 1991], [Baader and Sattler, 2001]).

2.5 $\mathcal{ALCH}_{R^+}$ $(\mathcal{SH})$

La logica $\mathcal{ALCH}_{R^+}$ presenta, rispetto ad $\mathcal{ALC}$, due nuove caratteristiche: la chiusura transitiva dei ruoli primitivi e il loro ordinamento gerarchico attraverso l'operatore di sussunzione. La prima delle due consente di introdurre assiomi del tipo $Rin\mathbf{R}_+$, dove R è il nome di un ruolo e $\mathbf{R}_+$ è l'insieme dei nomi dei ruoli transitivi della terminologia. Il supporto per la *role hierarchy*, invece, permette l'introduzione di assiomi del tipo $R \sqsubseteq S$, dove R ed S sono nomi di ruoli primitivi.

Poiché la logica $\mathcal{ALC}_{R^+}$, in quanto variante sintattica della la logica (multi)modale $\mathbf{S4_{(m)}}$, viene anche chiamata $\mathcal{S}$, il nome di $\mathcal{ALCH}_{R^+}$ può essere abbreviato in $\mathcal{SH}$.

La relazione di sussunzione ($\sqsubseteq$) definisce un ordinamento parziale in $\mathbf{R}$, l'insieme dei ruoli che compaiono nella terminologia. Essa è riflessiva (per ogni ruolo R, $R \sqsubseteq R$), antisimmetrica (per ogni coppia di ruoli R ed S, $R \sqsubseteq S$ e $S \sqsubseteq R$ implica $R = S$) e transitiva (per ogni tripletta di ruoli Q, R, S, $Q \sqsubseteq R$ ed $R \sqsubseteq S$ implica $Q \sqsubseteq S$). Inoltre, in presenza di questa relazione, anche i ruoli (come accade per i concetti) possono essere memorizzati in una gerarchia, un grafo diretto e aciclico in cui ogni ruolo è collegato ai propri *super-ruoli* e ai propri *sub-ruoli* diretti.

Transitività e ordinamento gerarchico dei ruoli aumentano notevolmente l'espressività della logica: la loro interazione, infatti, consente di individuare tutta una nuova tipologia di relazioni di sussunzione. Ad esempio, è possibile esprimere il fatto che *discendente* sia un ruolo transitivo che include sia *figlio* che *figlia*, tramite gli assiomi $discendente \in \mathbf{R}_+$, $figlio \sqsubseteq discendente$ e $figlia \sqsubseteq discendente$. A questo punto, si possono ricavare relazioni di sussunzione del tipo

$$\exists figlia.(\exists figlio.Ingegnere) \sqsubseteq \exists discendente.Ingegnere$$

Un algoritmo di tableaux per $\mathcal{SH}$

Nonostante la maggiore potenza espressiva di $\mathcal{SH}$ rispetto alla logica $\mathcal{ALC}$ (e nonostante la verifica di soddisfacibilità abbia sempre complessità espo-

nenziale), il suo algoritmo di tableaux è piuttosto semplice da implementare e si presta a un'ampia gamma di ottimizzazioni.

Come per gli altri algoritmi, lo scopo è quello di provare la soddisfacibilità di un concetto D dimostrando l'esistenza di un modello di D (un'interpretazione $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ tale che $D^{\mathcal{I}} \neq \emptyset$). Il modello è rappresentato da un albero, all'interno del quale i nodi corrispondono agli individui e sono etichettati con un insieme di concetti. Durante la verifica di soddifacibilità di un concetto D, questi insiemi vengono ristretti ai sottoinsiemi di $sub(D)$, dove $sub(D)$ corrisponde alla chiusura delle sottoespressioni di D e viene definito come segue:

1. se D è nella forma $\neg C$, $\exists R.C$ o $\forall R.C$, allora C è una sottoespressione di D e $sub(D) = \{D\} \cup sub(C)$;

2. se D è nella forma $C_1 \sqcap C_2$ o $C_1 \sqcup C_2$, allora C_1 e C_2 sono sottoespressioni di D e $sub(D) = \{D\} \cup sub(C_1) \cup sub(C_2)$;

3. altrimenti $sub(D) = \{D\}$.

Ogni nodo, quindi, viene etichettato con un insieme $\mathcal{L}(x) \subseteq sub(D)$. Per un nodo x, vale la definizione di clash definita nella sezione precedente: x, cioé, contiene un clash se $\bot \in \mathcal{L}(x)$ oppure, per qualche concetto C, $\{C, \neg C\} \subseteq \mathcal{L}(x)$. $\mathcal{L}(x)$, inoltre, è chiamato *pre-tableaux* se è libero da clash e non contiene concetti con congiunzioni o disgiunzioni ancora non espanse (si noti che $\emptyset$ è un pre-tableaux).

Gli archi all'interno dell'albero possono non avere etichetta, oppure essere etichettati con un nome di ruolo presente in $sub(D)$. Gli archi senza nome vengono creati in seguito all'espansione di concetti del tipo $C_1 \sqcap C_2$ e costituiscono il meccanismo attraverso il quale l'algoritmo cerca possibili espansioni alternative. Gli archi dotati di etichetta vengono create in seguito all'espansione di concetti del tipo $\exists R.C$ e corrispondono a relazioni fra coppie di individui.

Un nodo y è chiamato *R-successore* di un altro nodo x se c'è un arco $\langle x, y \rangle$ etichettato R; y è chiamato *$\sqcup$-successore* di x se c'è un percorso, costituito da archi senza etichette, da x a y. Un nodo x è *antenato* di un altro nodo y se c'è un percorso da x a y, indipendentemente dalle etichette attribuite agli archi. Si noti che sia la relazione di $\sqcup$-successore sia quella di antenato sono riflessive, in quanto ogni nodo è connesso a se stesso attraverso un percorso vuoto.

Infine, ogni nodo può essere marcato come *soddisfacibile* quando si verifica una delle seguenti condizioni:

- $\mathfrak{L}(x)$ è un pre-tableaux che non contiene concetti del tipo $\exists R.C$;
- $\mathfrak{L}(x)$ è un pre-tableaux che ha successori, ognuno dei quali è marcato come *soddisfacibile*;
- $\mathfrak{L}(x)$ non è un pre-tableaux e almeno uno dei suoi $\sqcup$-successori è marcato come *soddisfacibile*.

L'algoritmo inizializza un albero $\mathbf{T}$ in modo che contenga un singolo nodo x_0, con $\mathfrak{L}x_0 = \{D\}$. $\mathbf{T}$ viene quindi espanso, applicando ripetutamente le regole presenti nella Tabella 2.3, finché il nodo radice viene marcato come *soddisfacibile* oppure nessuna regola è più applicabile. Se, a questo punto, il nodo radice è marcato come *soddisfacibile* allora l'algoritmo restituisce *soddisfacibile*, altrimenti restituisce *non soddisfacibile*.

In ultimo, si notino le seguenti osservazioni:

- La regola "$\exists$" incorpora le azioni eseguite da entrambe le regole $\exists$ e $\forall$ nell'algoritmo per la logica $\mathcal{ALC}$; inoltre, essa è in grado di gestire i ruoli transitivi, propagando le etichette $\forall R.C$ agli R-successori quando $R \in \mathbf{R}_+$, e la gerarchia dei ruoli, elaborando $\forall S.C$ ogniqualvolta viene aggiunto un arco $\langle x, y\rangle$ tale che $\mathfrak{L}(\langle x, y\rangle) = R$ e $R \sqsubseteq S$. Infine, la parte b di tale regola costituisce una strategia di *blocking*, che previene la non terminazione in casi come, ad esempio, $D = \exists R.C \sqcap \forall R.(\exists R.C)$ e $R \in \mathbf{R}_+$.
- La combinazione delle regole $\sqcup$ e SAT costituisce un metodo di ricerca dei possibili modelli alternativi: poiché l'espansione di un nodo x non può modificare nessuno dei propri predecessori, la ricerca avviene solo nei sottoalberi di x, che rappresentano i suoi $\sqcup$-successori; la regola SAT marca come *soddisfacibile* i nodi foglia senza clash e completamente espansi, e risale l'albero marcando allo stesso modo i nodi pre-tableaux (quelli, cioè, con degli R-successori) come *soddisfacibile* se *tutti* i loro sottoalberi sono soddisfacibili e i nodi di ricerca (quelli con degli $\sqcup$-successori) se *almeno uno* dei loro sottoalberi è soddisfacibile.

L'algoritmo di tableaux appena descritto per la logica $\mathcal{SH}$ è corretto e completo: per una dimostrazione, si rimanda ai testi presenti in letteratura ([Horrocks, 1997]).

Esempi

Poiché il funzionamento dell'algoritmo di tableaux per $\mathcal{SH}$ è decisamente più complesso rispetto a quelli usati per le logiche meno espressive, all'in-

$\sqcap$:	se 1. x è foglia di $\mathbf{T}$, $\mathcal{L}(x)$ è libero da clash, $C_1 \sqcap C_2 \in \mathcal{L}(x)$ 2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$ allora $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
$\sqcup$:	se 1. x è foglia di $\mathbf{T}$, $\mathcal{L}(x)$ è libero da clash, $C_1 \sqcup C_2 \in \mathcal{L}(x)$ 2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$ allora crea due $\sqcup$-successori y, z di x con: $\mathcal{L}(y) = \mathcal{L}(x) \cup \{C_1\}$ $\mathcal{L}(z) = \mathcal{L}(x) \cup \{C_2\}$
$\exists$:	se x è una foglia di $\mathbf{T}$ e $\mathcal{L}(x)$ è un pre-tableaux allora per ogni $\exists R.C \in \mathcal{L}(x)$: a. $\ell_{Rx} := \{C\} \cup \{D \mid \forall S.D \in \mathcal{L}(x) \text{ e } R \sqsubseteq S\}$ $\cup \{\forall S.D \mid \forall S.D \in \mathcal{L}(x),\ S \in \mathbf{R}_+ \text{ e } R \sqsubseteq S\}$ b. se per qualche antenato w di x, $\ell_{Rx} \subseteq \mathcal{L}(w)$ allora crea un R-successore y di x con $\mathcal{L}(y) = \emptyset$ c. altrimenti crea un R-successore y di x con $\mathcal{L}(y) = \ell_{Rx}$
SAT: se	un nodo x non è marcato come *soddisfacibile* e vale una delle seguenti: a. $\mathcal{L}(x)$ è un pre-tableaux che non contiene concetti del tipo $\exists R.C$ b. $\mathcal{L}(x)$ è un pre-tableaux che ha successori, ognuno dei quali è marcato come *soddisfacibile* c. $\mathcal{L}(x)$ non è un pre-tableaux e almeno uno dei suoi $\sqcup$-successori è marcato come *soddisfacibile* allora marcalo come *soddisfacibile*.

Tabella 2.3: Le regole di espansione per $\mathcal{SH}$.

terno di questa sezione vengono utilizzati alcuni esempi per spiegarne le caratteristiche principali.

Esempio 1: Interazioni complesse fra ruoli

Questo esempio mostra una verifica di sussunzione risultante dall'interazione fra ruoli transitivi e la gerarchia di ruoli. Data una terminologia $\mathcal{T}$:

$$\{R \sqsubseteq Q,\, S \sqsubseteq Q,\, Q \in \mathbf{R}_+\} \subseteq \mathcal{T}$$

la gerarchia di ruoli rappresentata da $\mathcal{T}$ è mostrata nella Figura 2.1, con la notazione $Q^{(+)}$ usata per dichiarare che $Q \in \mathbf{R}_+$.

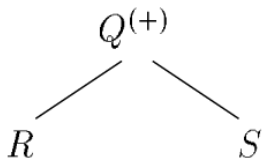

Figura 2.1: La gerarchia di ruoli rappresentata da $\mathcal{T}$.

È possibile mostrare che $\exists R.(\exists S.C) \sqsubseteq_{\mathcal{T}} \exists Q.C$ dimostrando che $\exists R.(\exists S.C) \sqcap \neg\exists Q.C$ è insoddisfacibile rispetto a $\mathcal{T}$:

1. converti $\exists R.(\exists S.C) \sqcap \neg\exists Q.C$ in forma normale negata:

$$\exists R.(\exists S.C) \sqcap \forall Q.\neg C$$

2. inizializza $\mathbf{T}$ in modo che contenga un singolo nodo x_0 etichettato come segue:

$$\mathfrak{L}(x_0) = \{\exists R.(\exists S.C) \sqcap \forall Q.\neg C\}$$

3. applica la regola $\sqcap$ a $\exists R.(\exists S.C) \sqcap \forall Q.\neg C \in x_0$:

$$\mathfrak{L}(x_0) \longrightarrow \mathfrak{L}(x_0) \cup \{\exists R.(\exists S.C), \forall Q.\neg C\}$$

4. applica la regola $\exists$ a $\exists R.(\exists S.C) \in \mathfrak{L}(x_0)$:

   - (a) $\ell_{Rx_0} := \{\exists S.C, \neg C, \forall Q.\neg C\}$
   - (b) Non c'è alcun antenato w di x_0 con $\ell_{Rx_0} \subseteq \mathfrak{L}(w)$, quindi crea un R-successore x_1 di x_0 con $\mathfrak{L}(x_1) = \ell_{Rx_0}$

5. applica la regola $\exists$ a $\exists S.C \in \mathcal{L}(x_1)$:
   - (a) $\ell_{Sx_1} := \{C, \neg C, \forall Q.\neg C\}$
   - (b) Non c'è alcun antenato w di x_1 con $\ell_{Sx_1} \subseteq \mathcal{L}(w)$, quindi crea un S-successore x_2 di x_1 con $\mathcal{L}(x_2) = \ell_{Sx_1}$

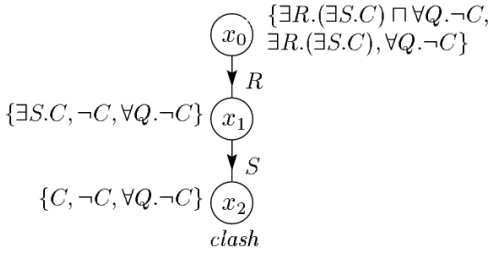

Figura 2.2: L'albero $\mathbf{T}$ *espanso per* $\exists R.(\exists S.C) \sqcap \neg\exists Q.C$.

Nessuna delle regole di espansione è più applicabile a $\mathbf{T}$, quindi l'albero è completamente espanso (v. Figura 2.2). Poichè $\mathcal{L}(x_2)$ contiene un clash ($\{C, \neg C\} \subseteq \mathcal{L}(x_2)$) esso non costituisce un pre-tableaux e la regola SAT non vale per esso. Come conseguenza, né x_1 né x_0 possono essere marcati come *soddisfacibile* e quindi l'algoritmo restituisce *non soddisfacibile*.

Esempio 2: Blocking

Questo esempio mostra il funzionamento del meccanismo di blocking. Data una terminologia $\mathcal{T}$:

$$\{R \in \mathbf{R}_+\} \subseteq \mathcal{T}$$

si può mostrare che $\exists R.C \not\sqsubseteq_{\mathcal{T}} \exists R.(\forall R.\neg C)$ dimostrando che $\exists R.C \sqcap \neg\exists R.(\forall R.\neg C)$ è soddisfacibile rispetto a $\mathcal{T}$:

1. converti $\exists R.C \sqcap \neg\exists R.(\forall R.\neg C)$ in forma normale negata:

$$\exists R.C \sqcap \forall R.(\exists R.C)$$

2. inizializza $\mathbf{T}$ in modo che contenga un singolo nodo x_0 etichettato come segue:

$$\mathcal{L}(x_0) = \{\exists R.C \sqcap \forall R.(\exists R.C)\}$$

3. applica la regola $\sqcap$ a $\exists R.C \sqcap \forall R.(\exists R.C) \in x_0$:

$$\mathfrak{L}(x_0) \longrightarrow \mathfrak{L}(x_0) \cup \{\exists R.C, \forall R.(\exists R.C)\}$$

4. applica la regola $\exists$ a $\exists R.C \in \mathfrak{L}(x_0)$:

   - (a) $\ell_{Rx_0} := \{C, \exists R.C, \forall R.(\exists R.C)\}$
   - (b) Non c'è alcun antenato w di x_0 con $\ell_{Rx_0} \subseteq \mathfrak{L}(w)$, quindi crea un R-successore x_1 di x_0 con $\mathfrak{L}(x_1) = \ell_{Rx_0}$

5. applica la regola $\exists$ a $\exists R.C \in \mathfrak{L}(x_1)$:

   - (a) $\ell_{Rx_1} := \{C, \exists R.C, \forall R.(\exists R.C)\}$
   - (b) C'è un antenato x_1 di x_1 (si ricorda che tale relazione è riflessiva) con $\ell_{Rx_1} \subseteq \mathfrak{L}(x_1)$, quindi crea un R-successore x_2 di x_1 con $\mathfrak{L}(x_2) = \emptyset$

6. applica la regola SAT a x_2: esso non è marcato *soddisfacibile* e $\mathfrak{L}(x_2)$ è un pre-tableaux (si ricorda che $\emptyset$ costituisce un pre-tableaux) che non contiene alcun concetto della forma $\exists R.C$, quindi marca x_2 come *soddisfacibile*.

7. applica la regola SAT a x_1: esso non è marcato *soddisfacibile* e $\mathfrak{L}(x_1)$ è un pre-tableaux con successori, ognuno dei quali è marcato come *soddisfacibile*.

8. applica la regola SAT a x_0: esso non è marcato *soddisfacibile* e $\mathfrak{L}(x_0)$ è un pre-tableaux con successori, ognuno dei quali è marcato come *soddisfacibile*.

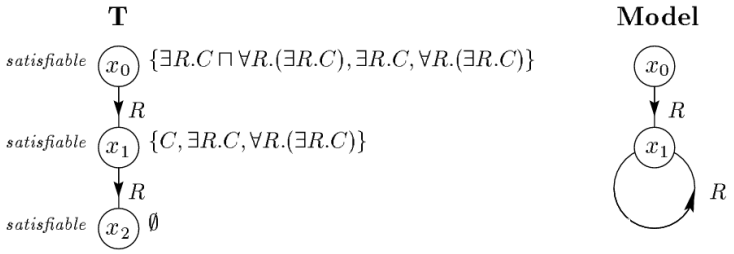

Figura 2.3: L'albero **T** *espanso e un modello per* $\exists R.C \sqcap \forall R.(\exists R.C)$.

Nessuna delle regole di espansione è più applicabile a **T**, quindi esso è completamente espanso. Poiché $\mathcal{L}(x_0)$ è marcato come *soddisfacibile*, l'algoritmo restituisce *soddisfacibile*. Sia l'albero **T** completamente espanso sia il modello che esso rappresenta sono mostrati nella Figura 2.3: si noti che $\mathcal{L}(x_2) = \emptyset$ rappresenta un ciclo all'interno del modello.

Esempio 3: Disgiunzioni

Questo esempio illustra il modo in cui l'algoritmo utilizza gli $\sqcup$-successori per cercare i possibili modelli alternativi. Data una terminologia $\mathcal{T}$ che non contiene alcun assioma relativo ai ruoli, si può mostrare che $\exists R.(C \sqcup D) \sqcap \forall R.\neg C$ è soddisfacibile rispetto a $\mathcal{T}$:

1. inizializza **T** in modo che contenga un singolo nodo x_0 etichettato come segue:

$$\mathcal{L}(x_0) = \{\exists R.(C \sqcup D) \sqcap \forall R.\neg C\}$$

2. applica la regola $\sqcap$ a $\exists R.(C \sqcup D) \sqcap \forall R.\neg C \in x_0$:

$$\mathcal{L}(x_0) \longrightarrow \mathcal{L}(x_0) \cup \{\exists R.(C \sqcup D), \forall R.\neg C\}$$

3. applica la regola $\exists$ a $\exists R.(C \sqcup D) \in \mathcal{L}(x_0)$:

   - (a) $\ell_{Rx_0} := \{C \sqcup D, \neg C\}$
   - (b) Non c'è alcun antenato w di x_0 con $\ell_{Rx_0} \subseteq \mathcal{L}(w)$, quindi crea un R-successore x_1 di x_0 con $\mathcal{L}(x_1) = \ell_{Rx_0}$

4. applica la regola $\sqcup$ a $C \sqcup D \in \mathcal{L}(x_1)$: crea due $\sqcup$-successori x_2 e x_3 con $\mathcal{L}(x_2) = \mathcal{L}(x_1) \cup \{C\}$ e $\mathcal{L}(x_3) = \mathcal{L}(x_1) \cup \{D\}$.

5. applica la regola SAT a x_3: esso non è marcato *soddisfacibile* e $\mathcal{L}(x_3)$ è un pre-tableaux che non contiene alcun concetto della forma $\exists R.C$, quindi marca x_3 come *soddisfacibile*.

6. applica la regola SAT a x_1: esso non è marcato *soddisfacibile*, $\mathcal{L}(x_1)$ non è un pre-tableaux e x_1 ha un successore (x_3) che è marcato come *soddisfacibile*.

7. applica la regola SAT a x_0: esso non è marcato *soddisfacibile* e $\mathcal{L}(x_0)$ è un pre-tableaux con successori, ognuno dei quali è marcato come *soddisfacibile*.

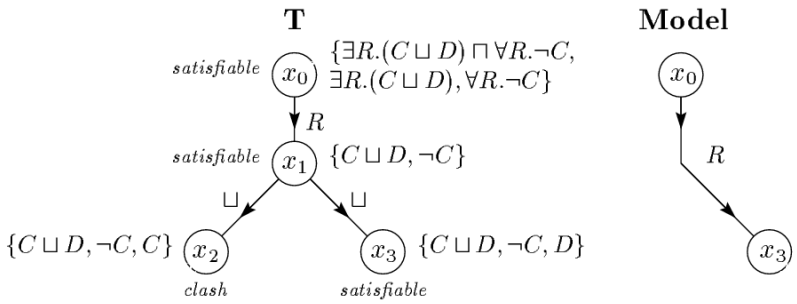

Figura 2.4: L'albero **T** *espanso e un modello per* $\exists R.(C \sqcup D) \sqcap \forall R.\neg C$.

Nessuna delle regole di espansione è più applicabile a **T**, quindi esso è completamente espanso. Poiché $\mathcal{L}(x_0)$ è marcato come *soddisfacibile*, l'algoritmo restituisce *soddisfacibile*. Sia l'albero **T** completamente espanso sia il modello che esso rappresenta sono mostrati nella Figura 2.4.

2.6 $\mathcal{SHF}$

La logica $\mathcal{SHF}$ (o $\mathcal{ALCHF}_{R^+}$) è un'estensione di $\mathcal{SH}$ che consente l'utilizzo degli attributi (o *ruoli funzionali*). In particolare, essa estende la sintassi di $\mathcal{SH}$ consentendo l'introduzione di assiomi del tipo:

$$A \in \mathbf{F}$$

e permettendo l'utilizzo di concetti complessi del tipo:

$$\exists A.C \mid \forall A.C$$

dove A è il nome di un attributo, C è un concetto semplice o complesso e **F** è l'insieme dei nomi di attributo.

Si noti che ogni ruolo che ha un attributo come super-ruolo deve, a sua volta, essere un attributo (se $B^{\mathcal{I}}$ è funzionale e $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$ allora anche $A^{\mathcal{I}}$ dev'essere funzionale). Inoltre, in $\mathcal{SHF}$ l'insieme dei ruoli transitivi è disgiunto da quello degli attributi ($\mathbf{R}_+ \cup \mathbf{F} = \emptyset$), poiché al prezzo di una ragionevole restrizione garantisce una notevole semplificazione dell'algoritmo di tableaux.

Naturalmente, la semantica descritta in precedenza (v. Tabella 2.1) dev'essere modificata per contemplare anche la presenza di ruoli funzionali. Innanzitutto, un'interpretazione $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ per $\mathcal{SHF}$ deve soddisfare la

Sintassi	Semantica
$\exists A.C$	$\{d \in dom\ A^{\mathcal{I}} \mid A^{\mathcal{I}}(d) \in C^{\mathcal{I}}\}$
$\forall A.C$	$\{d \in \Delta^{\mathcal{I}} \mid d \in dom\ A^{\mathcal{I}} \Rightarrow A^{\mathcal{I}}(d) \in C^{\mathcal{I}}\}$

Tabella 2.4: La semantica delle espressioni $\mathcal{SHF}$.

condizione aggiuntiva che, per ogni $A \in \mathbf{F}$, $A^{\mathcal{I}}$ sia una funzione parziale a un singolo valore:

$$A^{\mathcal{I}} : dom\ a^{\mathcal{I}} \longrightarrow \Delta^{\mathcal{I}}$$

Inoltre, la semantica relativa ai quantificatori esistenziale e universale viene definita come nella Tabella 2.4.

Un algoritmo di tableaux per $\mathcal{SHF}$

L'algoritmo di tableaux per $\mathcal{SH}$ può essere creato a partire da quello per $\mathcal{SH}$, modificando semplicemente la regola $\exists$ affinché sia in grado di gestire anche i ruoli funzionali. La nuova regola tratta gli attributi in modo simile ai ruoli: le espressioni $\exists A.C$ all'interno di un nodo pre-tableaux x, infatti, causano la creazione di nuovi nodi **A**-successori y_i e di archi $\langle x, y_i \rangle$ dotati di etichetta. Tuttavia, in alcuni casi la regola può raggruppare diverse espressioni $\exists A.C$ all'interno dell'etichetta di x, creando un singolo **A**-successore y, etichettando l'arco $\langle x, y \rangle$ con un insieme di nomi di attributo **A**.

Il raggruppamento di diverse espressioni del tipo $\exists A.C$ deve avvenire quando, nel modello rappresentato dall'albero, diversi $A^{\mathcal{I}}(x)$ sono vincolati ad essere lo stesso individuo (ad esempio, quando ci sono diverse espressioni $\exists A.C$ contenenti lo stesso attributo A). L'interazione fra gli attributi e la gerarchia di ruoli definita dalla relazione $\sqsubseteq$ ha come conseguenza il fatto che, per due espressioni $\exists A.C_1 \in \mathcal{L}(x)$ e $\exists B.C_2 \in \mathcal{L}(x)$, $A^{\mathcal{I}}(x)$ e $B^{\mathcal{I}}(x)$ sono vincolati ad essere lo stesso individuo non solo nel caso in cui $A = B$, ma anche quando $A \sqsubseteq B$ (poiché $A^{\mathcal{I}} \subseteq B^{\mathcal{I}}$) o $B \sqsubseteq A$ (poiché $B^{\mathcal{I}} \subseteq A^{\mathcal{I}}$).

In base a queste considerazioni, si definisce un attributo B come *direttamente vincolato* da un attributo A in $\mathcal{L}(x)$ se:

$$(\exists A.C \in \mathcal{L}(x) \text{ e } A \sqsubseteq B) \text{ o } (\exists B.C \in \mathcal{L}(x) \text{ e } B \sqsubseteq A)$$

e un attributo B come, semplicemente, *vincolato* da un altro attributo A in $\mathcal{L}(x)$ se B è direttamente vincolato da A in $\mathcal{L}(x)$ o se, per un particolare attributo A', A' è direttamente vincolato da A in $\mathcal{L}(x)$ e B è vincolato da A' in $\mathcal{L}(x)$. Per un attributo B e un nodo x, l'insieme di attributi che sono vincolati da B in $\mathcal{L}(x)$ viene chiamato $\mathbf{A}_{Bx}$:

$\exists$: se	x è una foglia di $\mathbf{T}$ e $\mathcal{L}(x)$ è un pre-tableaux
allora	per ogni $\exists R.C \in \mathcal{L}(x)$, in cui $R \notin \mathbf{F}$:
	a. $\ell_{Rx} := \{C\} \cup \{D \mid \forall S.D \in \mathcal{L}(x) \text{ e } R \sqsubseteq S\}$ $\cup \{\forall S.D \mid \forall S.D \in \mathcal{L}(x),\ S \in \mathbf{R}_+ \text{ e } R \sqsubseteq S\}$
	b. se per qualche antenato w di x, $\ell_{Rx} \subseteq \mathcal{L}(w)$ allora crea un R-successore y di x con $\mathcal{L}(y) = \emptyset$
	c. altrimenti crea un R-successore y di x con $\mathcal{L}(y) = \ell_{Rx}$
e	per ogni $\exists A.D \in \mathcal{L}(x)$, in cui $A \in \mathbf{F}$:
	a. se, per qualche $\mathbf{A}$-successore y di x, $A \in \mathbf{A}$, allora non fare nulla.
	b. altrimenti
	i. $\mathbf{A} := \mathbf{A}_{Ax}$
	ii. $\ell_{Ax} := \bigcup_{B \in \mathbf{A}} (\{C \mid \exists B.C \in \mathcal{L}(x)\} \cup$ $\{C \mid \forall S.C \in \mathcal{L}(x) \text{ e } B \sqsubseteq S\} \cup$ $\{\forall S.C \mid \forall S.C \in \mathcal{L}(x), S \in \mathbf{R}_+ \text{ e } B \sqsubseteq S\})$
	iii. se per qualche predecessore w di x, $\ell_{Ax} \subseteq \mathcal{L}(w)$ allora crea un $\mathbf{A}$-successore y di x con $\mathcal{L}(y) = \emptyset$
	iv. altrimenti crea un $\mathbf{A}$-successore y di x con $\mathcal{L}(y) = \ell_{Ax}$

Tabella 2.5: La regola $\exists$ modificata per $\mathcal{SHF}$.

$$\mathbf{A}_{Bx} = \{A \mid A \text{ è vincolato da } B \text{ in } \mathcal{L}(x)\}$$

Stanti queste premesse, l'algoritmo di tableaux per $\mathcal{SHF}$ segue le stesse regole di quello per $\mathcal{SH}$, fatta eccezione per la regola $\exists$ che viene sostituita da quella in Tabella 2.5. Si noti che la prima parte della regola è identica a quella definita per la logica $\mathcal{SH}$: il risultato è che per tutti i concetti che non contengono attributi gli alberi generati dai due algoritmi sono identici.

L'algoritmo appena descritto è corretto e completo: per una dimostrazione, si rimanda ai testi presenti in letteratura ([Horrocks, 1997]).

2.7 $\mathcal{SI}$

La logica $\mathcal{SI}$ è un'estensione della DL $\mathcal{ALC}$ con ruoli transitivi e ruoli inversi. I ruoli transitivi vengono interpretati come per le DL $\mathcal{SH}$ e $\mathcal{SHF}$, descritte in precedenza, mentre l'introduzione dei ruoli inversi costituisce la vera novità di questa logica rispetto alle precedenti. Tale estensione si traduce nella possibilità di utilizzare, per ogni ruolo $R \in \mathbf{R}$, un ruolo R^-, interpretato come l'inverso di R. In questo modo, l'insieme dei ruoli $\mathbf{R}$ si arricchisce

Sintassi	Semantica
$S \in \mathbf{R}$	$\langle x, y\rangle \in S^{\mathcal{I}}$ se e solo se $\langle y, x\rangle \in S^{-\mathcal{I}}$
$R \in \mathbf{R}_+$	se $\langle x, y\rangle \in R^{\mathcal{I}}$ e $\langle y, z\rangle \in R^{\mathcal{I}}$, allora $\langle x, z\rangle \in R^{\mathcal{I}}$

Tabella 2.6: La semantica dei concetti $\mathcal{SI}$.

diventando $\mathbf{R} \cup \{R^- \mid R \in \mathbf{R}\}$ e la semantica comprende anche gli enunciati presenti nella Tabella 2.6:

Per semplificare le considerazioni successive, vengono introdotte due funzioni sui ruoli:

1. La relazione inversa sui ruoli è simmetrica e, per evitare di dover considerare ruoli del tipo R^{--}, viene definita una funzione Inv che restituisce l'inverso di un ruolo. Più precisamente, $\mathsf{Inv}(R) = R^-$ se R è un nome di ruolo e $\mathsf{Inv}(R) = S$ se $R = S^-$.

2. Naturalmente, un ruolo R è transitivo se e solo se $\mathsf{Inv}(R)$ è a sua volta transitivo. Tuttavia, questo può essere stabilito sia dalla presenza di R sia da quella del suo inverso in $\mathbf{R}_+$. Viene perciò definita una funzione Trans che restituisce *vero* se e solo se R è un rolo transitivo, indipendentemente dal fatto che esso sia il nome di un ruolo o l'inverso di un nome di ruolo. Più precisamente, $\mathsf{Trans}(R) = \mathit{true}$ se e solo se $R \in \mathbf{R}_+$ o $\mathsf{Inv}(R) \in \mathbf{R}_+$.

Un algoritmo di tableaux per $\mathcal{SI}$

Come negli altri algoritmi descritti finora, anche quello per $\mathcal{SI}$ cerca di provare la soddisfacibilità di un concetto D costruendone un modello, rappresentato da un albero all'interno del quale ogni nodo corrisponde a un individuo nel modello ed è etichettato con un insieme di concetti (ristretto a un sottoinsieme di $sub(D)$, come descritto nella Sezione 2.5). L'unica differenza è che gli archi possono essere creati espandendo concetti che possono avere sia la forma $\exists R.C$ sia quella inversa $\exists R^-.C$.

Se i nodi x e y sono connessi fra loro da un arco $\langle x, y\rangle$, allora y è chiamato *successore* di x e x è chiamato *predecessore* di y; la definizione di *antenato* corrisponde alla chiusura transitiva di *predecessore*.

Un nodo y è chiamato R-vicino di un nodo x se y è successore di x e $\mathcal{L}(\langle x, y\rangle) = R$ o y è predecessore di x e $\mathcal{L}(\langle y, x\rangle) = \mathsf{Inv}(R)$.

Un nodo x è *bloccato* se, per qualche antenato y, y è bloccato o $\mathcal{L}(x) = \mathcal{L}(y)$. Un nodo è *indirettamente bloccato* se il suo predecessore è bloccato, altrimenti è *direttamente bloccato*. Si noti che se x è direttamente bloccato,

regola $\sqcap$:	se	1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$, x non è indirettamente bloccato e 2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
	allora	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$
regola $\sqcup$:	se	1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$, x non è indirettamente bloccato e 2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
	allora	$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$ per un $C \in \{C_1, C_2\}$
regola $\exists$:	se	1. $\exists S.C \in \mathcal{L}(x)$, x non è bloccato, e 2. x non ha S-vicini y con $C \in \mathcal{L}(y)$
	allora	crea un nuovo nodo y con $\mathcal{L}(\langle x, y \rangle) = \{S\}$ e $\mathcal{L}(y) = \{C\}$
regola $\forall$:	se	1. $\forall S.C \in \mathcal{L}(x)$, x non è indirettamente bloccato e 2. c'è un S-vicino y di x con $C \notin \mathcal{L}(y)$
	allora	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$
regola $\forall_+$:	se	1. $\forall S.C \in \mathcal{L}(x)$, $\mathsf{Trans}(S)$, x non è indirettamente bloccato, e 2. c'è un S-vicino y di x con $\forall S.C \notin \mathcal{L}(y)$
	allora	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\forall S.C\}$

Tabella 2.7: Le regole di espansione per $\mathcal{SI}$.

allora ha un unico antenato y tale che $\mathcal{L}(x) = \mathcal{L}(y)$: se, infatti, esistesse un altro antenato s tale che $\mathcal{L}(x) = \mathcal{L}(z)$, allora y o z dovrebbero essere bloccati. Se x è direttamente bloccato e y è l'unico antenato tale che $\mathcal{L}(x) = \mathcal{L}(y)$, allora si può dire che y *blocca* x.

L'algoritmo inizializza un albero $\mathbf{T}$ con un singolo nodo x_0, chiamato il *nodo radice*, con $\mathcal{L}(x_0) = \{D\}$, dove D è il concetto la cui soddisfacibilità dev'essere verificata. $\mathbf{T}$ viene quindi espanso, applicando ripetutamente le regole descritte nella Tabella 2.7.

L'albero è *completo* quando, per un nodo x, $\mathcal{L}(x)$ contiene un clash, oppure quando nessuna delle regole è più applicabile. Se, per un particolare concetto in ingresso D, le regole di espansione possono essere applicate in modo da generare un albero completo e senza clash, allora l'algoritmo restituisce *soddisfacibile*, altrimenti esso restituisce *non soddisfacibile*.

L'algoritmo di tableaux appena descritto per la logica $\mathcal{SI}$ è corretto e completo: per la dimostrazione, si rimanda ai testi presenti in letteratura ([Horrocks and Sattler, 1999], [Horrocks et al., 1999b], [Horrocks et al., 1999a]).

regola $\forall'_+$: se	1. $\forall S.C \in \mathcal{L}(x)$, x non è indirettamente bloccato e
	2. c'è un R tale che $\mathsf{Trans}(R)$ e $R \sqsubseteq^* S$, e
	3. c'è un R-vicino y di x con $\forall R.C \notin \mathcal{L}(y)$
allora	$\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\forall R.C\}$

Tabella 2.8: La regola $\forall_+$ di $\mathcal{SI}$ modificata per funzionare con $\mathcal{SHI}$.

2.8 $\mathcal{SHI}$

La logica $\mathcal{SHI}$ è un'estensione di $\mathcal{SI}$ in grado di gestire anche le *gerarchie di ruoli*, tramite un operatore di inclusione come quello già visto nella Sezione 2.5. Tale operatore può essere applicato allo stesso modo a ruoli transitivi o intransitivi, così come a ruoli semplici oppure inversi. Ad esempio, è possibile esprimere il fatto che un ruolo R sia simmetrico semplicemente aggiungendo i due assiomi $R \sqsubseteq R^-$ e $R^- \sqsubseteq R$.

L'interazione fra ruoli transitivi, inversi e gerarchie di ruoli rende necessaria una nuova definizione:

Definizione 2 *Un* assioma di inclusione di ruoli *ha la forma*

$$R \sqsubseteq S,$$

per due ruoli (eventualmente inversi) R ed S. Per un insieme di assiomi di inclusione di ruoli $\mathcal{R}$,

$$\mathcal{R}^+ := (\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\},\ \sqsubseteq^*)$$

viene chiamato gerarchia di ruoli*, dove $\sqsubseteq^*$ è la chiusura transitiva e riflessiva di $\sqsubseteq$ su $\mathcal{R} \cup \{\mathsf{Inv}(R) \sqsubseteq \mathsf{Inv}(S) \mid R \sqsubseteq S \in \mathcal{R}\}$.*

Oltre a essere corretta per i concetti di $\mathcal{SI}$, una interpretazione di $\mathcal{SHI}$ deve anche soddisfare, per ogni coppia i ruoli R, S con $R \sqsubseteq^* S$, la condizione aggiuntiva

$$\langle x, y \rangle \in R^{\mathcal{I}} \text{ implica } \langle x, y \rangle \in S^{\mathcal{I}}.$$

Un algoritmo di tableaux per $\mathcal{SHI}$

L'algoritmo di tableaux per $\mathcal{SHI}$ può essere creato in modo incrementale a partire da quello per $\mathcal{SI}$, sostituendo la definizione di R-vicino e la regola $\forall$ affinchè prendano in considerazione la nozione di gerarchia di ruoli. La nuova regola, chiamata $\forall'_+$, è mostrata nella Tabella 2.8; la nuova definizione di R-vicino, invece, compare qui di seguito.

Definizione 3 *Dato un albero, un nodo y è chiamato* S-vicino *di un nodo x se, per qualche R con $R \sqsubseteq^* S$, y è un successore di x e $\mathcal{L}(\langle x, y\rangle)$, oppure y è un predecessore di x e $\mathcal{L}(\langle y, x\rangle) = \mathsf{Inv}(R)$.*

L'algoritmo di tableaux appena descritto per la logica $\mathcal{SHI}$ è corretto e completo: per la dimostrazione, si rimanda ai testi presenti in letteratura ([Horrocks and Sattler, 1999]).

2.9 $\mathcal{SHIF}$

La logica $\mathcal{SHIF}$ è costruita a partire da $\mathcal{SHI}$, aggiungendo il supporto per le restrizioni funzionali. Il modo più generale per effettuare questa estensione è consentire, per un generico ruolo R (eventualmente inverso), concetti del tipo $(\leqslant 1R)$. Poiché la logica comprende anche l'uso dell'operatore di negazione, è necessario consentire anche restrizioni funzionali negate, del tipo $\neg(\leqslant 1R)$; queste, in forma normale negata (NNF), vengono espresse come $(\geqslant 2R)$.

Si noti che, all'interno di $\mathcal{SHIF}$, i ruoli che possono apparire all'interno di restrizioni funzionali sono limitati ad essere *semplici*, cioè non transitivi e privi di sub-ruoli transitivi. Senza questa limitazione, l'algoritmo di costruzione del tableaux sarebbe decisamente più complesso, a causa della possibilità di dover collassare catene di successori all'interno di un singolo nodo. Questo potrebbe verificarsi, ad esempio, nel caso in cui $(\leqslant 1S)$ venga aggiunto all'etichetta di un nodo x, dove $R \in \mathbf{R}_+$, x abbia già una catena di R-successori e $R \sqsubseteq^* S$.

I concetti appena espressi sono riassunti dalla

Definizione 4 *$\mathcal{SHIF}$ è l'estensione di $\mathcal{SHI}$ ottenuta aggiungendo il supporto per le restrizioni funzionali: per un ruolo semplice R, $(\leqslant 1R)$ è anch'esso un concetto $\mathcal{SHIF}$. Un ruolo è* semplice *se e solo se $R \notin \mathbf{R}_+$ e, per ogni $S \sqsubseteq^* R$, anche S è un ruolo semplice.*

Come accade anche per la logica $\mathcal{ALCFI}_{R^+}$ (chiamata anche $\mathcal{SFI}$, o $\mathcal{SIF}$, non descritta all'interno di questo testo), l'interazione all'interno di $\mathcal{SHIF}$ fra ruoli inversi e restrizioni numeriche (delle quali i ruoli funzionali sono un caso particolare) ha come conseguenza la perdita della *proprietà del modello finito* (finite model property). Tale proprietà stabilisce che se fra due concetti non sussiste una relazione di sussunzione è allora possibile creare un modello finito in grado di mostrarlo; in sua assenza, invece, è possibile che esistano ontologie consistenti che ammettono però solo modelli infiniti. Per gestire casi come questo è necessario, in fase di implementazione

dell'algoritmo di tableaux, adottare una nuova tecnica di blocking, che viene descritta in dettaglio all'interno della prossima sezione.

Un algoritmo di tableaux per $\mathcal{SHIF}$

L'algoritmo di tableaux per $\mathcal{SHIF}$ è un'estensione di quello ideato per $\mathcal{SHI}$ in grado di gestire anche i ruoli funzionali. Poiché l'interazione fra restrizioni funzionali e gerarchie di ruoli può causare la fusione di più nodi figli, originariamente uniti al padre tramite archi corrispondenti a ruoli dal nome differente, è necessario che il tableaux per $\mathcal{SHIF}$ utilizzi archi etichettati con *insiemi* di ruoli, anziché con ruoli semplici.

La conseguenza di questa modifica è che anche le definizioni di vicini e successori devono essere aggiornate, insieme alla regola $\exists$. Infine, la definizione di clash viene aggiornata per comprendere i casi in cui vi siano più restrizioni funzionali in conflitto fra loro. In particolare:

- dato un albero $\mathbf{T}$, un nodo y è chiamato *R-successore* di un nodo x se y è un successore di x e $S \in \mathcal{L}(\langle x, y\rangle)$ p er qualche S con $S \boxed{*} R$; y è chiamato *R-vicino* di x se è un R-successore di x, o se x è un $\mathsf{Inv}(R)$-successore di y;

- un nodo x è *direttamente bloccato* se nessuno dei suoi antenati è bloccato e se ha degli antenati x', y e y' tali che

  1. x è un successore di x' e y è un successore di y' e
  2. $\mathcal{L}(x) = \mathcal{L}(y)$ e $\mathcal{L}(x') = \mathcal{L}(y')$ e
  3. $\mathcal{L}(\langle x', x\rangle) = \mathcal{L}(\langle y', y\rangle)$

  In questo caso, si può dire che y *blocca* x;

- un nodo è *indirettamente bloccato* se il suo predecessore è bloccato e, per evitare espansioni inutili dopo l'applicazione di una regola $\leqslant$, si conviene che un nodo y sia indirettamente bloccato se è successore di un nodo x e $\mathcal{L}(\langle x, y\rangle) = \emptyset$;

- un nodo x contiene un *clash* se contiene un clash $\mathcal{SHI}$ oppure se contiene due ruoli R ed S tali che $\{(\leqslant 1R), (\geqslant 2S)\} \subseteq \mathcal{L}(x)$ e $S \boxed{*} R$.

Come si può notare, anche la definizione di blocco è cambiata: grazie alla nuova tecnica adottata, che si chiama *pair-wise blocking*, è possibile gestire agevolmente anche dei modelli non finiti. Si, consideri, ad esempio, il problema di provare la non soddisfacibilità del concetto

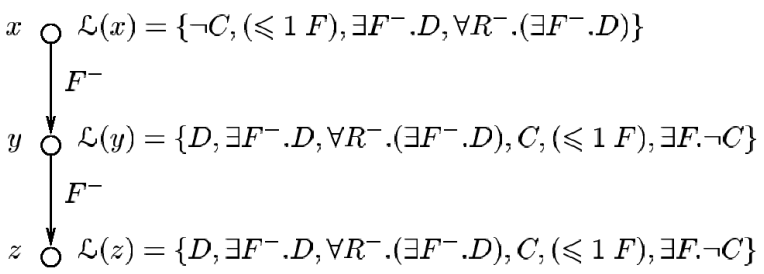

Figura 2.5: Un tableau per il quale il pair-wise blocking è fondamentale.

$$\neg C \sqcap (\leqslant 1F) \sqcap \exists F^{-}.D \sqcap \forall R^{-}.(\exists F^{-}.D),$$

dove $F \sqsubseteq R$ e D rappresenta il concetto

$$C \sqcap (\leqslant 1F) \sqcap \exists F.\neg C.$$

Utilizzando il classico algoritmo di blocking, il nodo z (v. Figura 2.5) verrebbe bloccato da y e l'albero risultante non potrebbe rappresentare un modello ciclico in cui y sia collegato con se stesso tramite un ruolo F^{-}, in quanto questo andrebbe in conflitto con $(\leqslant 1F) \in \mathcal{L}(y)$. Quindi, per rappresentare il modello infinito generato, sarebbe necessario sostituire, all'interno dell'albero, ogni ricorrenza di z con una copia del sottoalbero che ha radice in y. Questo modello, tuttavia, non sarebbe valido, poiché, quando z viene sostituito con una copia di y, $\exists F.\neg C \in \mathcal{L}(y)$, che era soddisfatto da $\neg C \in \mathcal{L}(x)$, non è più soddisfatto nella sua nuova posizione.

Utilizzando il pair-wise blocking, z non è più bloccato da y in quanto le etichette dei loro predecessori (rispettivamente, y e x) sono diverse, quindi l'algoritmo continua ad espandere $\mathcal{L}(z)$. L'espansione di $\exists F.\neg C \in \mathcal{L}(z)$ richiede l'esistenza di un nodo la cui etichetta comprenda $\neg C$ e che sia connesso a z da un arco etichettato con F. Poiché, tuttavia, $(\leqslant 1F) \in \mathcal{L}(z)$, questo nodo dev'essere per forza y e questo genera una contraddizione, in quanto sia C sia $\neg C$ appartengono a $\mathcal{L}(y)$.

Le regole per l'algoritmo sono mostrate nella Tabella 2.9: esso è corretto e completo (per la dimostrazione, si rimanda ai testi presenti in letteratura [Horrocks and Sattler, 1999], [Horrocks et al., 1999b]).

regola $\sqcap$: se 1. $C_1 \sqcap C_2 \in \mathcal{L}(x)$, x non è indirettamente bloccato e
2. $\{C_1, C_2\} \not\subseteq \mathcal{L}(x)$
allora $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C_1, C_2\}$

regola $\sqcup$: se 1. $C_1 \sqcup C_2 \in \mathcal{L}(x)$, x non è indirettamente bloccato e
2. $\{C_1, C_2\} \cap \mathcal{L}(x) = \emptyset$
allora $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{C\}$ per un $C \in \{C_1, C_2\}$

regola $\exists$: se 1. $\exists S.C \in \mathcal{L}(x)$, x non è bloccato, e
2. x non ha S-vicini y con $C \in \mathcal{L}(y)$
allora crea un nuovo nodo y con
$\mathcal{L}(\langle x, y\rangle) = \{S\}$ e $\mathcal{L}(y) = \{C\}$

regola $\forall$: se 1. $\forall S.C \in \mathcal{L}(x)$, x non è indirettamente bloccato e
2. c'è un S-vicino y di x con $C \notin \mathcal{L}(y)$
allora $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{C\}$

regola $\forall'_+$: se 1. $\forall S.C \in \mathcal{L}(x)$, x non è indirettamente bloccato e
2. c'è un R tale che $\mathsf{Trans}(R)$ e $R \sqsubseteq^* S$, e
3. c'è un R-vicino y di x con $\forall R.C \notin \mathcal{L}(y)$
allora $\mathcal{L}(y) \longrightarrow \mathcal{L}(y) \cup \{\forall R.C\}$

regola $\geqslant$: se 1. $(\geqslant 2R) \in \mathcal{L}(x)$, x non è bloccato, e
2. non ci sono R-vicini y di x con $A \in \mathcal{L}(y)$
allora crea due nodi y_1, y_2 con $\mathcal{L}(\langle x, y_1\rangle) = \{R\}$,
$\mathcal{L}(\langle x, y_2\rangle) = \{R\}$, $\mathcal{L}(y_1) = \{A\}$ e $\mathcal{L}y_2 = \{\neg A\}$

regola $\leqslant$: se 1. $(\leqslant 1R) \in \mathcal{L}(x)$, x non è indirettamente bloccato,
2. x ha due R-vicini y e z t.c. y non sia antenato di z,
allora 1. $\mathcal{L}(z) \longrightarrow \mathcal{L}(z) \cup \mathcal{L}(y)$ e
2. se z è antenato di y
allora $\mathcal{L}(\langle z, x\rangle) \longrightarrow \mathcal{L}(\langle z, x\rangle) \cup \mathsf{Inv}(\mathcal{L}(\langle x, y\rangle))$
altrimenti $\mathcal{L}(\langle x, z\rangle) \longrightarrow \mathcal{L}(\langle x, z\rangle) \cup \mathcal{L}(\langle x, y\rangle)$
3. $\mathcal{L}(\langle x, y\rangle) \longrightarrow \emptyset$

Tabella 2.9: Le regole di espansione per $\mathcal{SHIF}$.

Capitolo 3

Struttura del reasoner

JODIE è un reasoner complesso e modulare: i suoi componenti svolgono le operazioni più diverse, dalla gestione delle connessioni HTTP al parsing dei messaggi DIG, fino ad arrivare al salvataggio dei dati nella base di conoscenze e alle procedure di ragionamento. Oltre a queste, naturalmente, vi è tutta una serie di operazioni accessorie, ma non meno importanti, come ad esempio la gestione dei log, l'interfaccia grafica e la possibilità di visualizzare graficamente il funzionamento degli algoritmi di tableaux. Con lo scopo di fornire una visione completa del progetto, all'interno di questo capitolo viene mostrata la struttura del reasoner, partendo da una descrizione generale fino ad arrivare a quella dei singoli componenti.

3.1 Descrizione generale

JODIE è stato programmato per rispondere a particolari requisiti funzionali e strutturali: i primi (supporto per gli standard OWL e DIG, uso degli algoritmi di tableaux, possibilità di utilizzo su diverse architetture hardware/software) sono già stati descritti all'interno del Capitolo 1. Ad essi si aggiungono i seguenti:

- **Modularità**: dev'essere possibile aggiungere funzioni al programma in modo semplice. In particolare, le varie tecniche di ragionamento devono essere intercambiabili, associando a logiche diverse algoritmi differenti o permettendo all'utente di utilizzare vari algoritmi per la stessa logica.
- **Semplicità**: il reasoner, funzionando come servizio, deve "partire e funzionare", cioè dev'essere in grado di operare senza necessità di in-

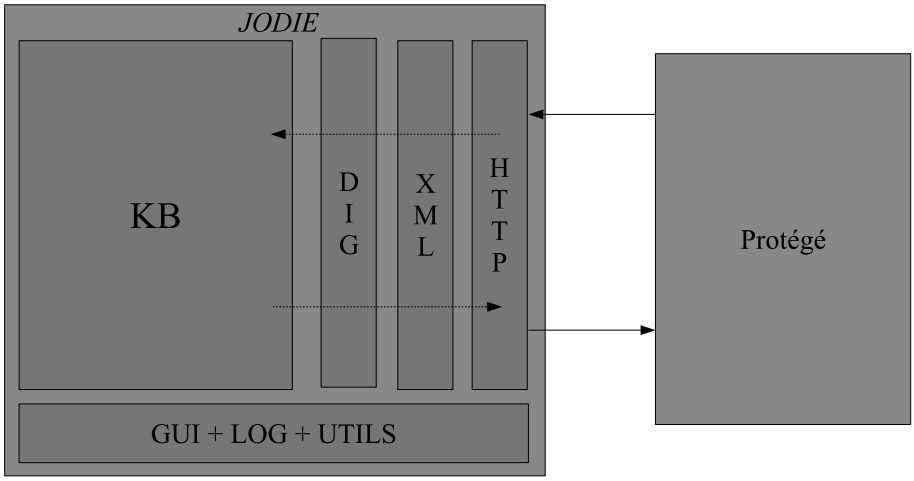

Figura 3.1: La struttura di JODIE e la comunicazione con l'editor di ontologie.

terazione da parte dell'utente. Nella versione con interfaccia grafica, l'utilizzo deve risultare semplice e intuitivo. Infine, essendo prevista la distribuzione del programma insieme al suo codice sorgente, quest'ultimo dev'essere semplice da comprendere e da modificare.

Queste considerazioni hanno riconfermato la scelta di Java come linguaggio di programmazione per il reasoner. Grazie ad esso, infatti, è stato possibile non solo creare un'applicazione portabile, ma anche, avvalendosi dell'architettura a classi che è propria del linguaggio, produrre del codice pulito e modulare.

La caratteristica principale di JODIE è quella di operare come *servizio di ragionamento*. Esso, cioè, consente a qualsiasi programma, in esecuzione sulla stessa macchina o su una collegata in rete, di effettuare il salvataggio dei dati nella base di conoscenze e la sua conseguente interrogazione attraverso un protocollo di comunicazione standard. Tale protocollo si chiama DIG ed è ormai piuttosto diffuso fra i reasoner e i programmi che comunicano con essi. Il funzionamento di JODIE in congiunzione a una di queste applicazioni (nella fattispecie, l'editor di ontologie Protégé) è esemplificato nella Figura 3.1.

Le comunicazioni che avvengono durante la conversazione fra JODIE e il generico client che si collega ad esso non si limitano, naturalmente, ai semplici protocolli HTTP (che si occupa del trasferimento dei dati in rete) e DIG (usato per la codifica dei messaggi): queste sono, infatti, solo comunicazioni *esterne* all'applicazione, mentre *internamente* ad essa i vari moduli

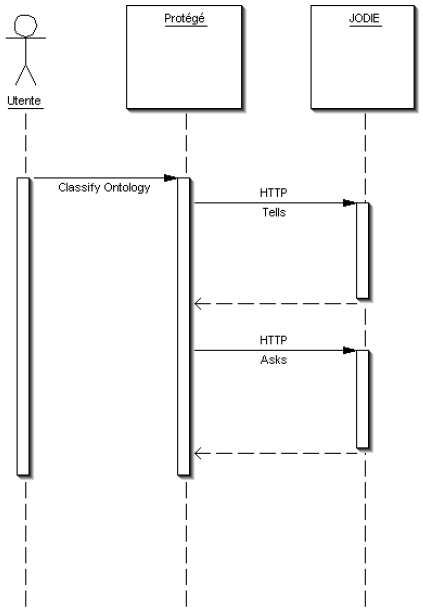

Figura 3.2: La comunicazione fra editor di ontologie e JODIE può contenere messaggi di tipo diverso.

dialogano fra di loro. D'altra parte, le modalità attraverso cui queste comunicazioni interne hanno luogo possono cambiare a seconda delle richieste esterne.

Ad esempio, per portare a termine un'operazione come la classificazione di un'ontologia, un programma come Protégé deve prima inviare al reasoner dei messaggi che, secondo il protocollo DIG, appartengono alla categoria delle *tells*, cioè le affermazioni riguardanti gli elementi costitutivi della base di conoscenze. In seguito, per classificare ogni singolo concetto, esso deve inviare dei messaggi di tipo *asks*, tramite i quali esso chiede al reasoner se determinati concetti possono o meno implicarne degli altri. Tutto questo può avvenire in momenti separati ma, più frequentemente, accade nel corso di una sola connessione (v. Figura 3.2).

Analizzando le operazioni compiute dal reasoner a fronte di diverse tipologie di messaggio, è possibile constatare che esso reagisce in modi differenti a seconda delle richieste che gli pervengono. In particolare, quando esso riceve una *tells*, le comunicazioni fra i vari moduli avvengono come segue (v. Figura 3.3):

1. L'editor di ontologie invia una richiesta in formato DIG al reasoner, attraverso il protocollo HTTP.

2. Il modulo HTTP riceve la richiesta e la invia al modulo XML.

3. Il modulo XML, innanzitutto, interpreta l'XML attraverso il parser DOM e, quindi, effettua il parsing dei comandi in formato DIG passandoli al modulo omonimo.

4. I comandi interpretati dal modulo DIG vengono inviati alla KB, che effettua il vero e proprio salvataggio dei dati, nonché il controllo di correttezza dei comandi che le vengono inviati.

5. Alla fine delle operazioni, o in caso di errore, la KB restituisce l'esito dei comandi a DIG.

6. In base alla risposta della KB, il modulo DIG crea una risposta nel formato omonimo e la restituisce ad XML, che a sua volta la invia ad HTTP.

7. HTTP invia la risposta DIG all'editor di ontologie.

Nella Figura 3.4, invece, è possibile vedere in che modo vengono scambiati i messaggi fra i vari moduli nel caso in cui il reasoner riceva una richiesta di tipo *asks*. In particolare:

1. L'editor di ontologie invia una richiesta in formato DIG al reasoner, attraverso il protocollo HTTP.

2. Il modulo HTTP riceve la richiesta e la invia al modulo XML.

3. Il modulo XML effettua il parsing del messaggio, appoggiandosi al parser DOM, quindi il modulo DIG chiede alla base di conoscenze di effettuare l'operazione di ragionamento richiesta.

4. La KB utilizza la classe *Tableaux* per risolvere la query con un algoritmo di tableaux.

5. Una volta ricevuta la risposta dall'oggetto tableaux, la KB restituisce l'esito a DIG

6. Il modulo DIG crea una risposta per l'editor di ontologie e la invia ad HTTP.

7. HTTP invia la risposta DIG all'editor di ontologie.

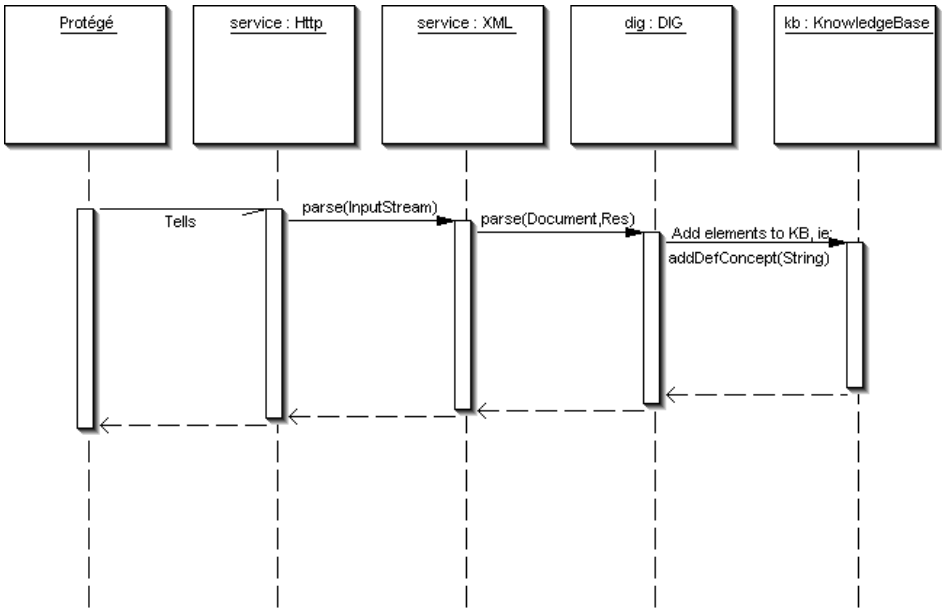

Figura 3.3: Lo scambio di informazioni fra l'editor di ontologie, JODIE e i suoi moduli nel caso di un messaggio di tipo tells.

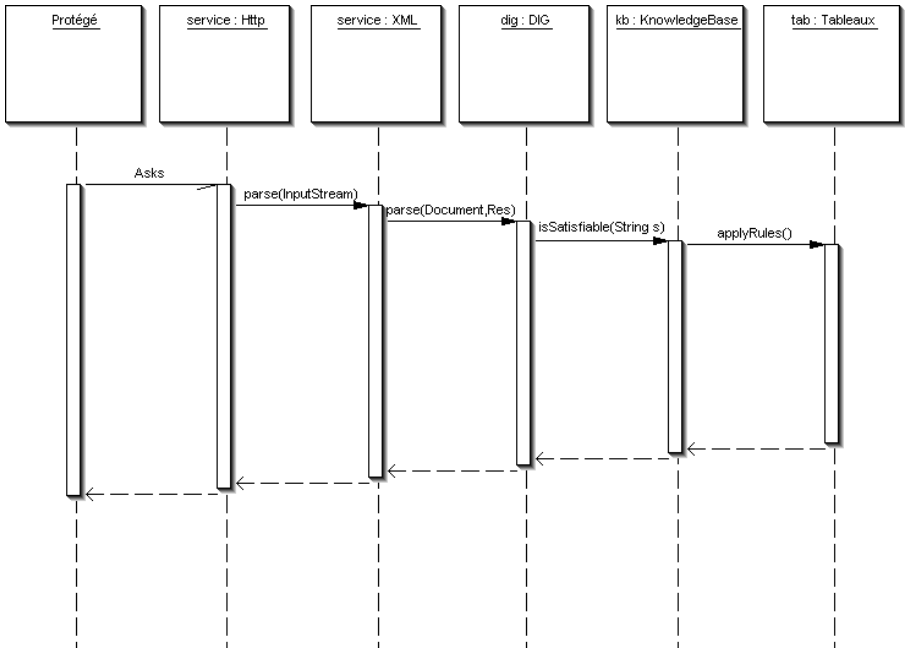

Figura 3.4: Lo scambio di informazioni fra l'editor di ontologie, JODIE e i suoi moduli nel caso di un messaggio di tipo asks.

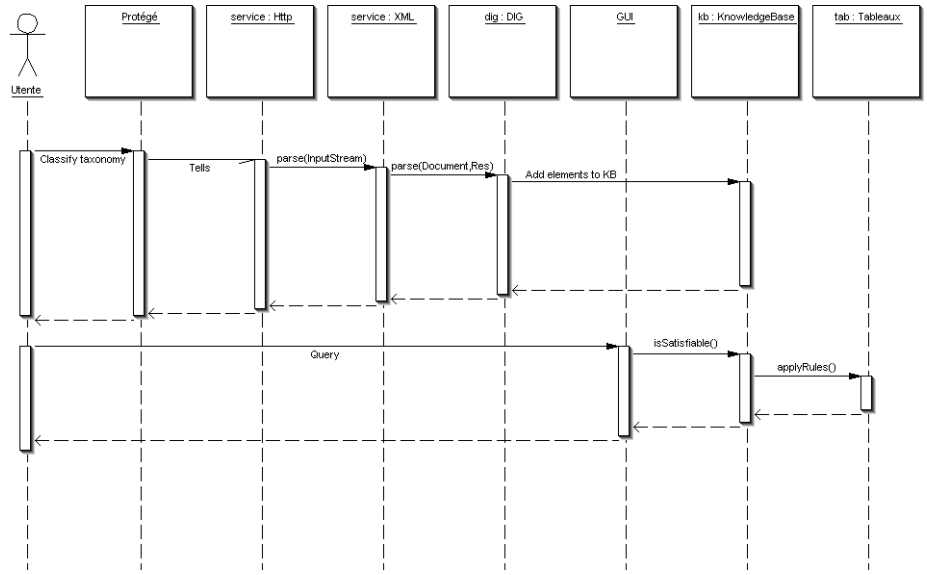

Figura 3.5: In seguito al salvataggio dei dati nella KB è possibile, tramite la GUI, inserire manualmente delle query.

Tutto questo, naturalmente, avviene solo nel caso in cui la comunicazione esterna sia totalmente effettuata attraverso i protocolli HTTP e DIG. Nel caso in cui si decida di eseguire JODIE in modalità grafica è possibile, una volta inviate le tells tramite HTTP/DIG, eseguire manualmente delle query nella base di conoscenze. In questo caso, l'applicazione non riceve delle *asks* tramite HTTP, ma una richiesta diretta da parte dell'utente: di conseguenza, il funzionamento del programma cambia come mostrato nella Figura 3.5:

3.2 HTTP e DIG

Per quanto riguarda l'implementazione del server HTTP, si è scelto di usare una libreria già pronta: **Simple** (`http://simpleweb.sourceforge.net`) è, come afferma il suo stesso nome, un server semplice da utilizzare, liberamente utilizzabile grazie alla sua licenza GPL, leggero (il file jar occupa meno di 200KB) e sufficientemente completo da poter essere utilizzato per un progetto come JODIE.

A livello pratico, usare Simple equivale semplicemente a creare una classe che estenda l'oggetto *BasicService*, messo a disposizione dalla libreria, con un metodo `process` che riceve in ingresso un oggetto di tipo *Request* e ne restituisce uno di tipo *Response*. Nel caso di JODIE (v. Figura 3.6) si è deciso di usare questo metodo per

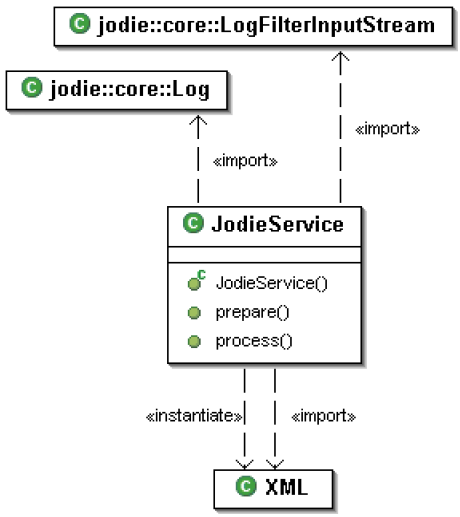

Figura 3.6: La classe JodieService *e quelle da lei importate.*

- inviare al parser XML/DIG il messaggio ricevuto
- inviare al client una risposta appropriata, creata in base all'esito delle richieste DIG sulla base di conoscenze
- salvare tutti i messaggi in ingresso e in uscita attraverso il sistema di log

Di queste operazioni, le prime due competono a una classe di JODIE chiamata *XML*, mentre l'ultima viene gestita da *Log* (descritto in dettaglio nella sezione 3.6). Per salvare i log dei messaggi ricevuti è stato creato un nuovo oggetto, chiamato *LogFilterInputStream*, che intercetta il flusso di dati in ingresso e li salva automaticamente prima di elaborarli.

Per eseguire il parsing dei messaggi DIG, la classe *XML* si appoggia a due diversi oggetti: il primo è un parser DOM, che riceve in ingresso uno stream di dati e restituisce una struttura ad albero (il cosiddetto "DOM tree"); il secondo è *DIG*, la classe che riceve il DOM tree e ne estrae i comandi destinati alla base di conoscenze, restituendo quindi la risposta della KB in formato DIG, pronta da inviare al server HTTP.

Per la creazione della classe *DIG* sono stati utilizzati come riferimento diversi documenti: [Haarslev and Möller, 2003c], [Dickinson, 2004] ma soprattutto [Bechhofer et al., 1999], che contiene una descrizione dettagliata dello standard **DIG/1.1**, compatibile con gli editor di ontologie più recenti. La scelta del parser DOM nonostante fossero già disponibili, sul sito

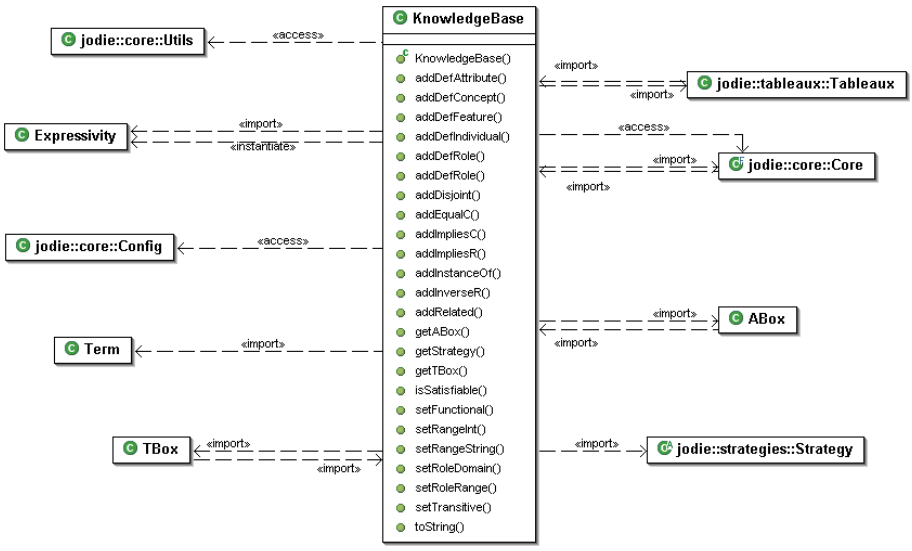

Figura 3.7: La struttura della classe KnowledgeBase.

`http://dig.sourceforge.net`, dei JavaBean per la gestione dei messaggi XML, è la naturale conseguenza a diversi esperimenti e considerazioni. Innanzi tutto, lo standard non è ancora sufficientemente formale da eliminare ambiguità e permettere a un programma di funzionare efficientemente solo con i JavaBean. Inoltre, l'uso dei Bean risulta notevolmente più lento rispetto a quello del parser DOM. A supporto di questa affermazione, sono state sviluppate due diverse versioni della classe DIG, ognuna delle quali utilizzava una delle due tecnologie, e le si sono cronometrate: i JavaBean comportavano un tempo fisso di attivazione di diversi secondi, che magari può risultare trascurabile per basi di conoscenze molto grosse, ma che sicuramente incide in modo notevole su basi di piccola/media grandezza.

Infine, l'utilizzo di DOM comporta un ulteriore vantaggio: la sua struttura ad albero, infatti, è particolarmente compatibile con quella dell'oggetto *Term*, descritto in seguito, che viene usato per salvare i generici termini (siano essi dichiarazioni di individui, concetti o ruoli) all'interno della base di conoscenze.

3.3 La base di conoscenze

La classe *KnowledgeBase* di JODIE racchiude, al suo interno, tutti i dati e gli oggetti che costituiscono la base di conoscenze del reasoner. La struttura della classe è mostrata nella Figura 3.7: come si può notare, oltre alle classi

comuni a diversi moduli (*Config*, *Utils*, *Core*) ne vengono utilizzate alcune specifiche, che compongono la struttura tipica della knowledge base come compare in letteratura (v. Capitolo 2).

In particolare, la base di conoscenze fa riferimento, per la gestione di concetti e ruoli, a una classe *TBox* (Figura 3.8) e, per le operazioni sugli individui, a una classe chiamata *ABox* (Figura 3.9): entrambi gli oggetti espongono una serie di metodi che vengono chiamati direttamente dalla knowledge base e che consentono di eseguire qualsiasi operazione sui propri elementi. Ruoli, concetti e individui sono, a loro volta, oggetti appartenenti a classi indipendenti (rispettivamente, *Role*, *Concept* e *Individual*) e vengono salvati all'interno di TBox e ABox in strutture hash che ne consentono un rapido accesso.

L'oggetto *KnowledgeBase* non funge solo da contenitore per i suoi componenti, ma è un punto di riferimento, almeno per quanto riguarda tutte le operazioni relative alla base di conoscenze, per tutti gli altri moduli dell'applicazione. Esso, infatti:

- espone i metodi di *TBox* e *ABox* per quanto riguarda la gestione di concetti, ruoli e individui (questi metodi vengono chiamati, ad esempio, dall'oggetto *DIG*).
- fa da interfaccia con la classe *Tableaux* per il ragionamento: le query, infatti, sono rivolte alla knowledge base, ed è quest'ultima che si occupa di inizializzare ed eseguire l'algoritmo di tableaux.
- sceglie, in base all'espressività della logica in uso e ai parametri di configurazione impostati dall'utente, la strategia di ragionamento da utilizzare per espandere il tableaux.

Sia la base di conoscenze sia i suoi elementi costituitivi (partendo da TBox e ABox fino ad arrivare a concetti, ruoli e individui) utilizzano un componente di base, chiamato *Term*, che rappresenta il generico "termine" usato per descrivere concetti. Un termine può essere *atomico* e far riferimento a un semplice concetto (ad esempio, `Pizza`), oppure *complesso* e contenere ruoli, concetti e individui collegati assieme da connettivi logici, come ad esempio

```
some(haBase, BaseCroccante)
```

oppure

```
all(haIngrediente, or(Pomodoro, Mozzarella)).
```

Come già è possibile notare dagli esempi citati, la notazione che viene usata da JODIE per scrivere un termine è quella in forma prefissa: i connettivi logici, infatti, sono sempre seguiti dai propri operandi racchiusi fra

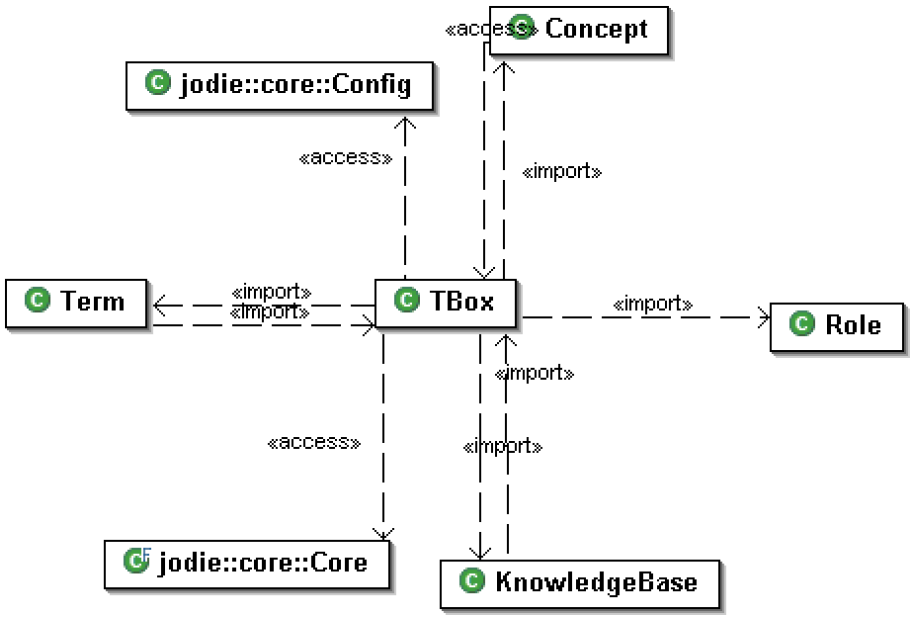

Figura 3.8: La struttura della classe TBox.

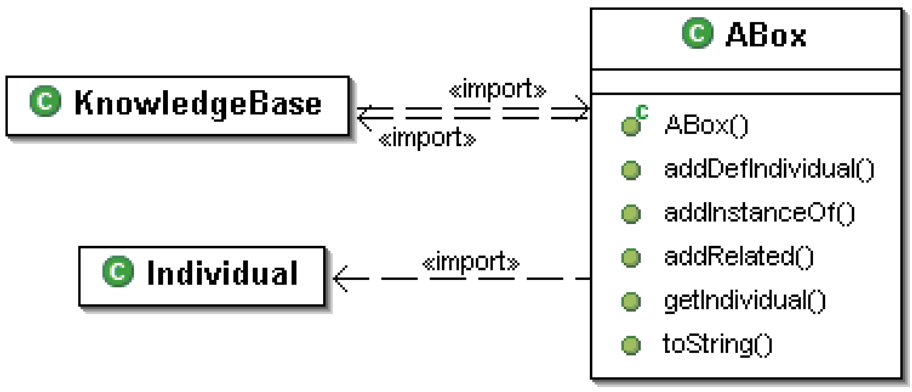

Figura 3.9: La struttura della classe ABox.

parentesi. Analogamente, la struttura con cui i termini complessi vengono salvati in memoria è un albero, la cui radice corrisponde all'operatore e le cui foglie rappresentano gli operandi. Tale struttura è particolarmente pratica per svolgere tutte quelle operazioni di semplificazione e, più in generale, trasformazione dei termini che sono necessarie all'interno di un reasoner. Nella fattispecie, i metodi inclusi nella classe *Term* consentono:

- la **negazione** di un termine (che consiste semplicemente nella creazione di una nuova radice `NOT`, il cui figlio è il termine corrente, seguita eventualmente da una semplificazione).
- la **semplificazione**, che consente di eliminare elementi ridondanti dai termini (ad esempio, due `NOT` in cascata) o di individuare clash all'interno di un singolo termine (come nel caso `AND(A,NOT(A))`).
- la **normalizzazione**, che trasforma tutte le forme ambigue in un particolare formato deciso a priori, permettendo agli altri metodi di operare su termini che rispondono a un particolare standard (ad esempio, `or(A,B,C)` diventa `NOT(AND(NOT(A),NOT(b),NOT(C)))`).
- l'**unfolding** di un concetto, che espande il suo termine in base alle definizioni presenti nella base di conoscenze.
- la trasformazione in **NNF** (forma normale negata), che sposta tutte le negazioni di un termine accanto ai semplici concetti. Questa è la forma in cui si devono trovare tutti i termini prima dell'applicazione delle regole degli algoritmi di tableaux.
- la verifica dell'**eguaglianza** fra due termini.

3.4 La classe Tableaux

Come descritto nel Capitolo 2, gli algoritmi di tableaux si propongono di dimostrare la soddisfacibilità di un termine cercando di costruirne un modello che non contenga contraddizioni. La classe *Tableaux* si occupa della costruzione di tale modello, tramite l'applicazione delle regole proprie della strategia di ragionamento adottata (si veda, a proposito, il Capitolo 4). Inoltre, essa contiene la struttura dati del tableaux, solitamente ad albero, sotto forma di nodi e archi interconnessi.

Di tutto l'albero, *Tableaux* punta semplicemente al nodo radice, dal quale poi si dipana tutta la struttura. Ogni nodo, infatti, può avere diversi figli, ad esso collegati tramite degli archi. Sia ai nodi che agli archi viene associato

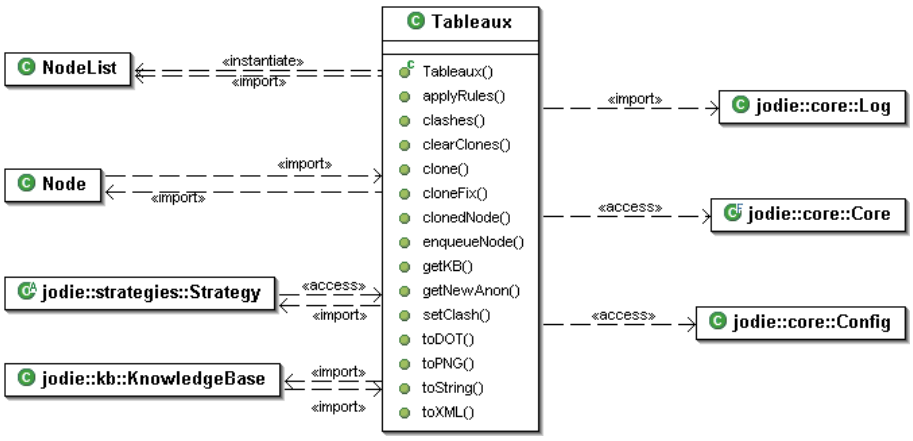

Figura 3.10: La struttura della classe Tableaux.

un nome: per i primi esso corrisponde al nome di un individuo, mentre nel secondo caso è il nome del ruolo che connette i due nodi. Dato un oggetto di tipo *Node*, è sempre possibile ottenere l'elenco dei suoi figli e dei suoi predecessori; dato un oggetto di tipo *Edge*, invece, è sempre possibile ricavare i nodi che esso connette.

Allo stesso modo in cui la classe *KnowledgeBase* usa, come elemento base, l'oggetto *Term*, *Tableaux* e le classi ad essa associate hanno un altro oggetto di riferimento, chiamato *Assertion*: esso non è altro che un "contenitore" per i termini, che ne permette l'utilizzo in modo quasi trasparente ma che aggiunge in più numerose informazioni utili agli algoritmi di tableaux. Esso, infatti, contiene una serie di flag che determinano lo stato del termine (ad esempio, se è già stato semplificato o convertito in forma normale negata) e un'etichetta, che viene usata dall'algoritmo di tableaux per effettuare un rollback ogniqualvolta l'espansione non deterministica di una disgiunzione conduca a un clash.

Come già accennato nel capitolo introduttivo, la classe *Tableaux* permette l'esportazione della propria struttura non solo sotto forma di stringa (con il classico metodo `toString()`), ma anche in XML, DOT e come immagine, richiamando direttamente il programma esterno GraphViz. Questa varietà di opzioni permette all'utente di seguire l'evoluzione dell'algoritmo, o di vederne il risultato finale, in diversi modi: quello offerto nativamente da JODIE è una pagina Web contenente sia l'immagine del grafo generato, la quale fornisce un'idea immediata dell'esito dell'algoritmo, sia la versione XML della struttura del tableaux, più complessa ma sicuramente anche più completa.

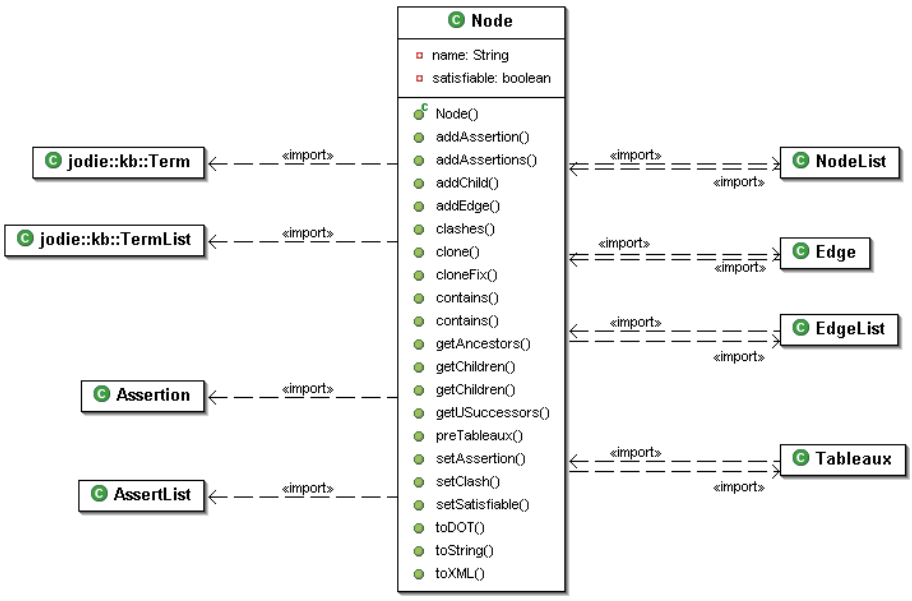

Figura 3.11: La struttura della classe Node.

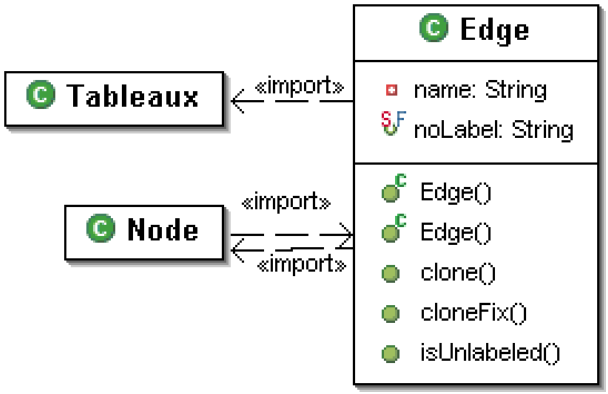

Figura 3.12: La struttura della classe Edge.

3.5 L'interfaccia grafica

JODIE, in quanto servizio di ragionamento, può essere eseguito da linea di comando e funzionare in modalità testo su un qualsiasi server. Tuttavia, per i sistemi che mettono a disposizione dell'utente un ambiente grafico, JODIE offre anche la possibilità di utilizzare un'interfaccia più intuitiva, capace di gestire efficacemente i numerosi messaggi generati dal reasoner e di operare direttamente sulla base di conoscenze.

Per lo sviluppo dell'interfaccia grafica si è scelto di utilizzare le librerie SWT. Uno dei principali motivi che hanno portato a tale decisione è la possibilità di mettere a disposizione dell'utente un ambiente familiare, con lo stesso *look and feel* del sistema operativo che sta utilizzando (si confrontino, ad esempio, le figure 3.13 e 3.14); ad esso si aggiunge la semplicità di utilizzo di SWT, la disponibilità di buona documentazione e la maggiore velocità rispetto ad altre librerie grafiche (come, ad esempio, Swing). L'unico aspetto negativo che tale scelta comporta è rappresentato dalla necessità di distribuire, insieme all'applicazione, anche la versione nativa delle librerie per il sistema operativo su cui la si desidera eseguire: poiché, tuttavia, esse sono liberamente disponibili e allo stesso tempo la GUI non è un elemento indispensabile per il corretto funzionamento del reasoner, si è scelto ugualmente di usare SWT.

L'interfaccia nel suo complesso è piuttosto semplice: essa, infatti, è composta solamente di una finestra, contenente un menu e diverse schede. La stragrande maggioranza delle schede è adibita a un unico scopo, quello cioè di conservare i messaggi generati dai vari componenti del reasoner. Una sola di esse, invece, consente di eseguire delle nuove azioni: il suo nome è **KB Query** e al suo interno è presente un modulo che consente di inviare manualmente delle query al reasoner.

Quando il programma viene avviato in modalità grafica, l'utente ha la possibilità di eseguire diverse operazioni in più rispetto alla versione da linea di comando. Le varie sezioni del menu mostrano tutte le funzioni a disposizione: da `File`, ad esempio, è possibile usare il comando `Open` per aprire un file in formato DIG. Il server HTTP che risponde alle richieste provenienti dai programmi esterni può essere avviato o arrestato tramite un comando richiamabile dal menu `DIG`: in questo caso, l'esito viene mostrato all'interno della scheda predefinita, chiamata **Jodie**. Infine, dal menu `KB` è possibile aprire o nascondere la scheda **KB Query**.

Nella modalità di funzionamento standard (quella, cioè, in cui viene avviato il server HTTP e l'editor di ontologie comunica con JODIE), quando un client riesce a collegarsi al reasoner, ha inizio una comunicazione fra i due

programmi. La conversazione viene salvata in due diverse schede all'interno dell'applicazione: rispettivamente, le richieste che arrivano al reasoner fanno capo alla scheda chiamata **DIG req** (Figura 3.13), mentre le risposte che JODIE invia ai client vengono salvate nella scheda **DIG res** (Figura 3.14).

3.6 I moduli Core, Log e Config

Tutte le classi che sono state descritte finora per funzionare si appoggiano ad alcuni oggetti di base, che costituiscono il nocciolo dell'applicazione. Tali oggetti sono:

- *Core*, che fa da punto di collegamento fra i vari moduli e consente loro l'accesso alle classi principali
- *Log*, che si occupa della visualizzazione dei messaggi prodotti dal reasoner
- *Config*, che gestisce la configurazione del sistema, permettendo all'utente di modificarne i parametri di funzionamento e mettendoli a disposizione dei vari componenti che ne fanno richiesta

In particolare, *Core* è in grado di fornire, a qualsiasi classe ne faccia richiesta, i seguenti servizi:

- avvio e arresto del server HTTP
- accesso alla base di conoscenze
- accesso alla finestra principale di JODIE
- accesso al servizio di log principale
- accesso al servizio di timer, usato per misurare le prestazioni del sistema

La classe *Log*, invece, è stata creata per gestire, nel modo più trasparente possibile, i messaggi generati dai moduli di JODIE in diversi momenti della sua esecuzione. Essendo nata come strumento di supporto alla GUI, essa è naturalmente in grado di visualizzare tali messaggi in modalità grafica, tuttavia offre anche la possibilità di mostrarli a terminale o, in caso l'applicazione diventi troppo verbosa, disattivarli del tutto. Questo consente di seguire in ogni momento il funzionamento del reasoner, partendo dalle

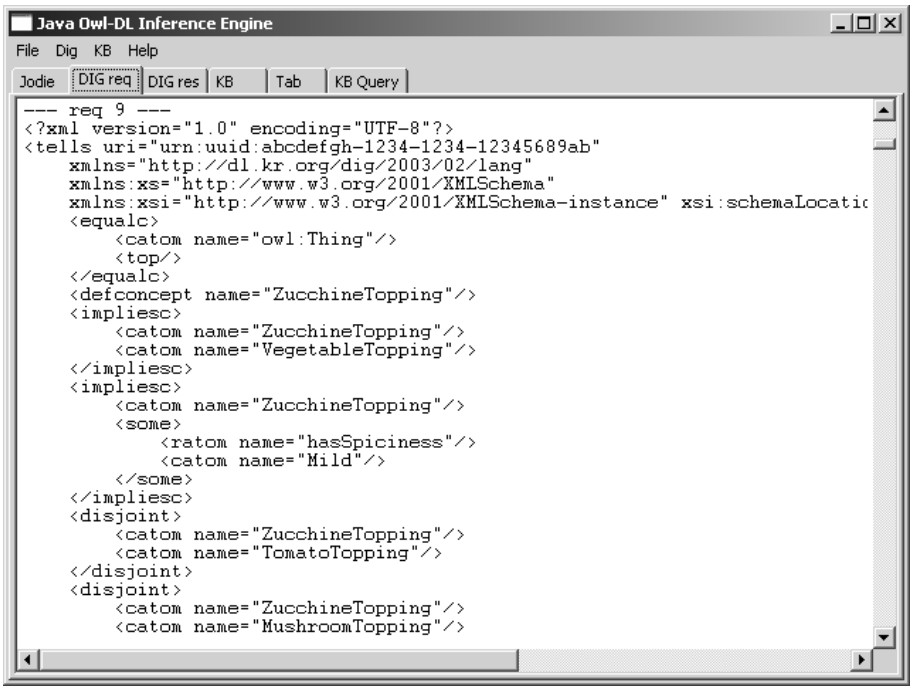

Figura 3.13: La scheda **DIG req** *contiene tutte le richieste DIG che arrivano al reasoner.*

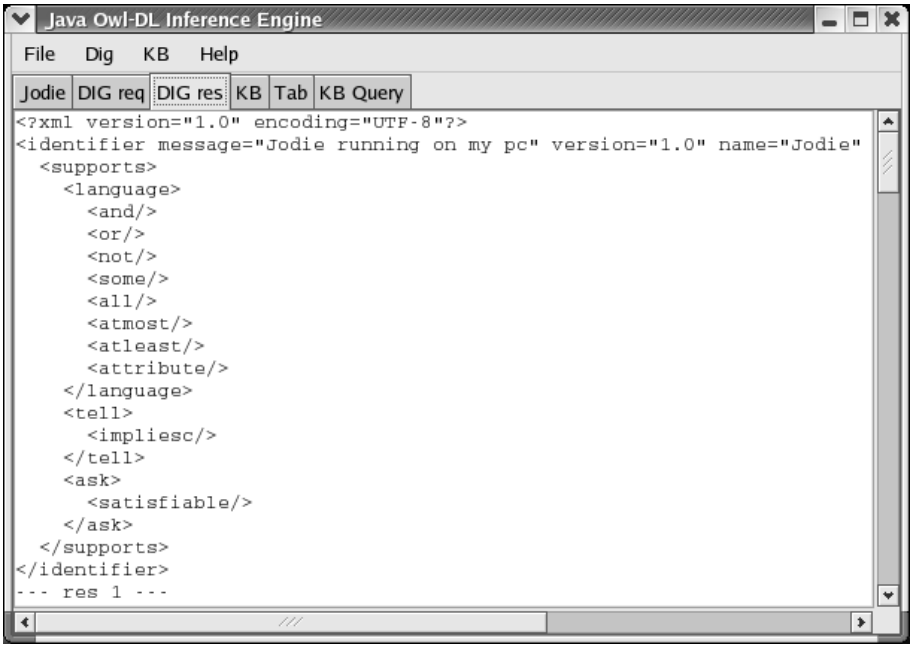

Figura 3.14: La scheda **DIG res** *contiene tutte le risposte DIG che il reasoner invia ai client.*

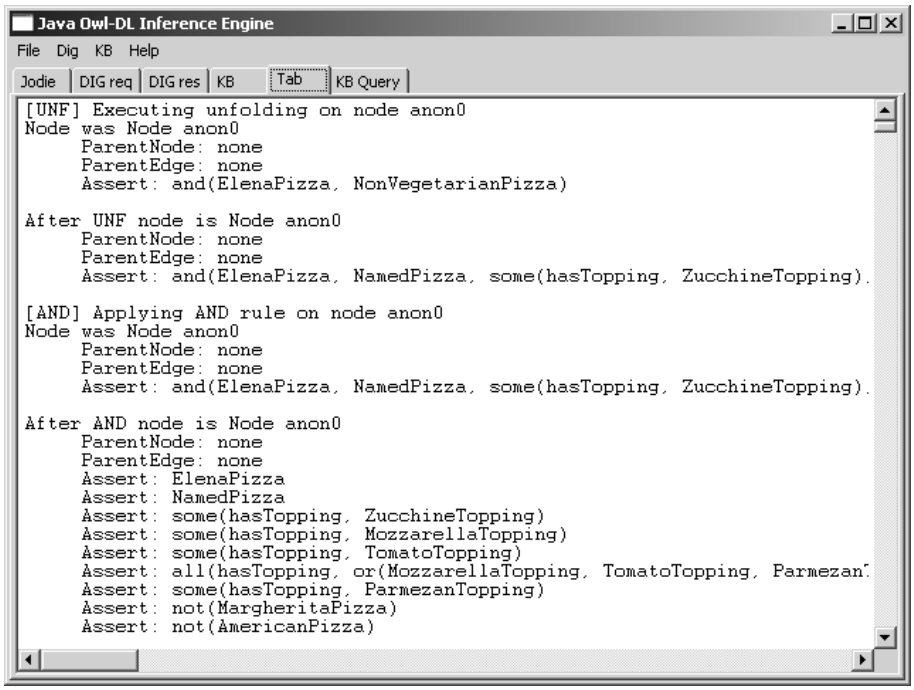

Figura 3.15: Un Log chiamato **Tab** *segue l'applicazione delle regole dell'algoritmo di tableaux.*

comunicazioni DIG fino ad arrivare al salvataggio dei dati nella base di conoscenze, alla loro semplificazione e al ragionamento tramite gli algoritmi di tableaux.

Ogni oggetto di tipo *Log* viene creato con un nome che lo identifica e una variabile che ne specifica la tipologia. A seconda del valore di questa variabile, i relativi messaggi compa:riranno nella GUI (in una scheda con lo stesso nome dell'oggetto), a terminale (preceduti dallo stesso nome) oppure saranno disattivati. Poiché la stessa interfaccia grafica può essere a sua volta abilitata o meno, vi sono diverse combinazioni di funzionamento in base alla tipologia di log: quando è OFF, il messaggio non viene visualizzato; se è CMD, il messaggio compare su terminale; se è GUI e l'interfaccia grafica è attiva, il testo viene visualizzato all'interno di una scheda nella finestra principale, in caso contrario viene ignorato.

La classe *Config*, come suggerito dal suo stesso nome, si occupa di gestire la configurazione dell'applicazione. Le impostazioni possono essere specificate in tre modi diversi:

- come parametri predefiniti, all'interno del codice sorgente di JODIE

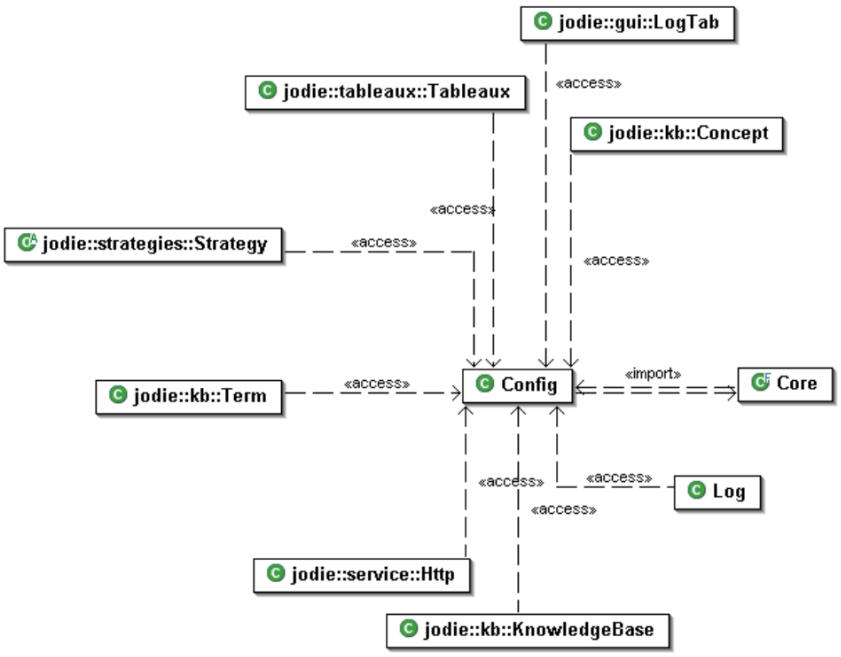

Figura 3.16: Diverse classi accedono a Config *per leggere i propri parametri di configurazione.*

- nel file di configurazione
- da linea di comando (al momento, tale possibilità viene offerta solo per il parametro `hasGUI`)

Il file di configurazione (Figura 3.17) è scritto in XML ed è diviso in sezioni diverse. Al primo livello di profondità compare la sezione `profile`, all'interno della quale vengono specificati le classi di parametri dipendenti da un particolare profilo. Questa tecnica viene utilizzata, insieme al rilevamento del sistema operativo in uso, per modificare automaticamente tutti quei parametri che cambiano in base al sistema utilizzato. In questa versione di JODIE, in particolare, il rilevamento automatico consente di distinguere un sistema Windows da uno UNIX e, a seconda dell'OS, vengono caricati i parametri del profilo `win` piuttosto che quelli di `nix`. Le impostazioni presenti in `common`, invece, sono quelle comuni a entrambi i sistemi e vengono quindi sempre prese in considerazione.

Fra le sezioni presenti al secondo livello (quelle, cioè, contenute all'interno di un `profile`) è possibile trovare:

- `base`, che contiene le impostazioni di base, come ad esempio l'attivazione della modalità debug, l'interfaccia grafica e la porta a cui deve rispondere il server HTTP.
- `log`, all'interno della quale vengono salvati i parametri di funzionamento dei log.
- `output`, che contiene i nomi dei file di output in cui vengono salvati i dati (in formato XML, DOT e immagine) relativi ai tableaux.
- `strategies`, con le corrispondenze fra espressività e strategie di ragionamento.
- `path`, contenente i percorsi delle directory di output e di GraphViz (per la rappresentazione grafica dei tableaux).

```
<config>
    <profile name="common">
        <base>
            <debug>true</debug>
            <hasGUI>true</hasGUI>
            <hasGV>true</hasGV>
            <httpPort>8080</httpPort>
        </base>
        <log><!-- 0=OFF, 1=CMD, 2=GUI -->
            <main>2</main>
            <dig>2</dig>
            <res>2</res>
            <kb>0</kb>
            <tab>1</tab>
        </log>
        <output>
            <tableauxXML>tableaux.xml</tableauxXML>
            <dotfilename>mala.dot</dotfilename>
            <imageformat>png</imageformat>
            <imgfilename>mala.png</imgfilename>
        </output>
        <strategies>
            <ALL>SH</ALL>
            <ALC>ALC</ALC>
            <ALCF>ALCf</ALCF>
        </strategies>
    </profile>
    <profile name="win">
        <path>
            <datapath>D:\incomin\poli\thesis\data\</datapath>
            <dothomedir>C:\Programmi\ATT\Graphviz\bin</dothomedir>
        </path>
    </profile>
    <profile name="nix">
        <path>
            <datapath>/home/eynard/data/</datapath>
            <dothomedir>/home/eynard/graphviz/bin</dothomedir>
        </path>
    </profile>
</config>
```

Figura 3.17: Il file di configurazione: un esempio.

Capitolo 4

Strategie di ragionamento

Come già descritto in dettaglio nel Capitolo 2, gli algoritmi di tableaux operano secondo regole che possono cambiare a seconda della complessità della logica a cui vengono applicati. Alcuni di essi sono definiti in modo incrementale rispetto ai precedenti, mentre altri comportano tali cambiamenti da dover essere quasi riscritti da zero. All'interno di questo capitolo viene descritto in dettaglio l'aspetto implementativo delle varie strategie di ragionamento, a partire dalla scelta della strategia da utilizzare fino ad arrivare, per ognuno degli algoritmi presi in considerazione, alla descrizione dei singoli componenti, delle somiglianze e delle novità che lo caratterizzano rispetto agli altri.

4.1 Strategie di ragionamento

Come conseguenza diretta dell'esistenza di diverse strategie di ragionamento, è possibile che un algoritmo sia più potente, cioè applicabile a logiche più espressive, ma allo stesso tempo più lento di un altro nato per essere utilizzato con logiche più semplici. Allora, per ottenere il massimo di prestazioni da un motore inferenziale, talvolta potrebbe risultare utile avere la possibilità di scegliere quale algoritmo usare in base all'espressività della logica utilizzata.

Per questo motivo JODIE usa una tecnica, già in parte sfruttata dal reasoner Pellet [Sirin and Parsia, 2004], che consiste nel calcolare l'espressività della KB in base alle operazioni che vengono effettuate su di essa e, a seconda del valore ottenuto, caricare la strategia di ragionamento preferita. In particolare:

1. La classe *Expressivity* viene creata come "contenitore di flag": essa,

cioè, non fa altro che accettare l'attivazione di una serie di parametri (che possono avere solo i valori vero o falso) e, nel momento in cui si chiama il metodo `toString`, compone una stringa contenente il nome della logica corrispondente ai flag che sono attivi in quel particolare momento (per maggiori informazioni sulle famiglie di logiche, è possibile consultare la Sezione 2.2).

2. I flag vengono attivati come segue:

   - una chiamata al metodo `setTransitive` della classe *KnowledgeBase* attiva il flag S (equivalente ad $\mathcal{ALC}_{R^+}$);
   - una chiamata al metodo `addImpliesR` della classe *KnowledgeBase* attiva il flag H (role hierarchy);
   - una chiamata al metodo `addInverseR` della classe *KnowledgeBase* attiva il flag I (inverse roles);
   - se il modulo *DIG* riceve un concetto del tipo **ATL** o **ATM** (per la sintassi di DIG si consulti [Bechhofer et al., 1999]), il flag N o Q viene attivato, a seconda che la restrizione sia qualificata o meno;
   - una chiamata al metodo `setFunctional` (che rende un ruolo funzionale) attiva il flag F (functional roles).

3. Nel momento in cui la base di conoscenze deve rispondere a una query, essa associa all'espressività calcolata al momento la strategia migliore (in base alle impostazioni dell'utente) e chiede al proprio tableaux di applicare l'algoritmo corrispondente.

La corrispondenza fra espressività e strategie viene stabilita dall'utente, all'interno del file di configurazione di JODIE: nella sezione `<strategies>` è possibile assegnare una strategia a ogni logica, oppure specificarne solo alcune e lasciare che il programma associ le altre alla strategia predefinita, marcata dal tag `<ALL>`.

Per quanto riguarda l'implementazione dei singoli algoritmi, si è deciso di organizzare le relative classi in una gerarchia capace di sfruttare, allo stesso tempo, la suddivisione in regole tipica degli algoritmi di tableaux e il fatto che alcune di tali regole siano ricorrenti in diverse strategie. Tale gerarchia si compone di una classe *Strategy*, dalla quale le altre ereditano i metodi di base, e una serie di classi di primo livello (nella fattispecie, $\mathcal{ALC}$, $\mathcal{SH}$ ed $\mathcal{SI}$), dalle quali altre strategie ereditano le metodologie tipiche di ogni famiglia di logiche (v. Fig. 4.1).

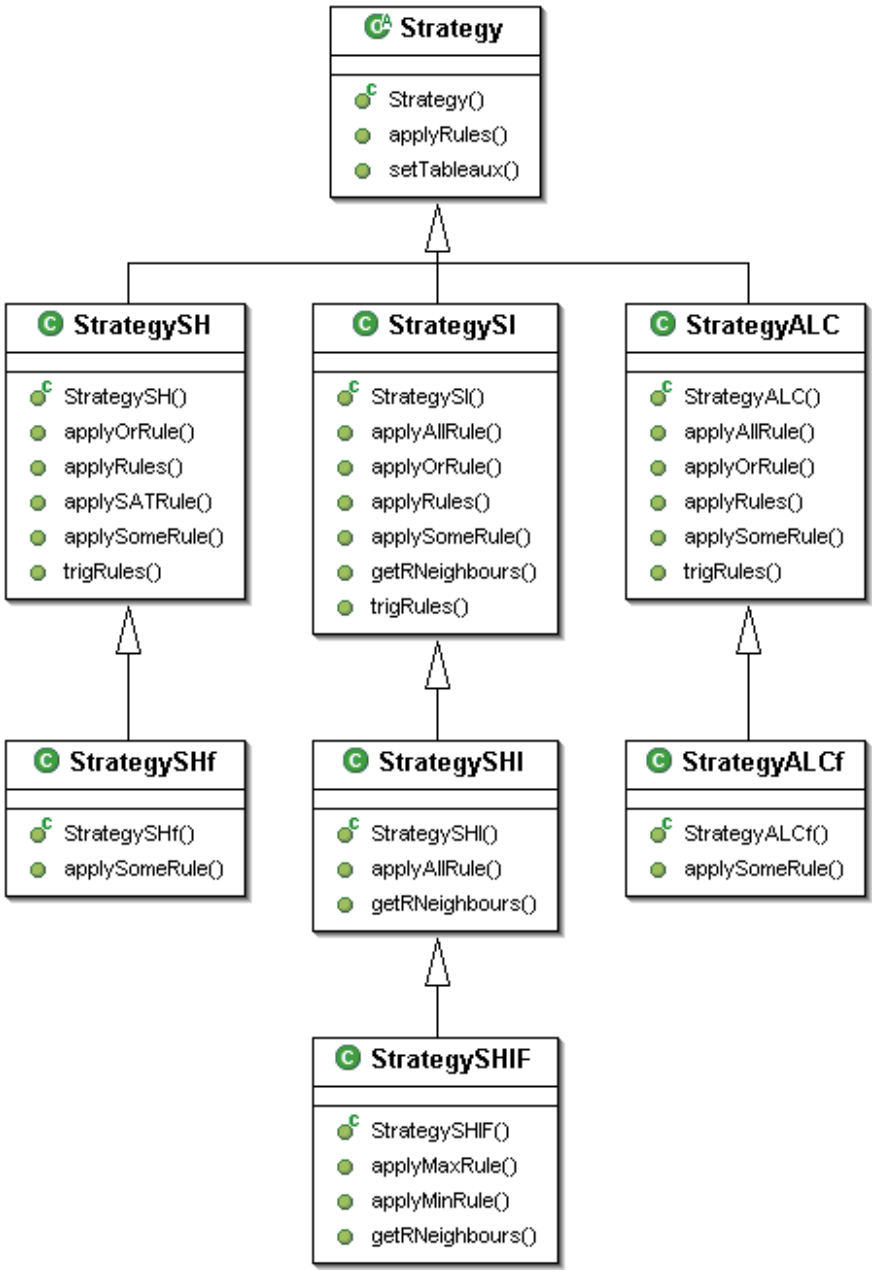

Figura 4.1: Un esempio di ereditarietà fra le classi di tipo Strategy.

La suddivisione in famiglie è risultata pressoché automatica, una volta riscontrate le notevoli differenze che ognuna di esse apportava alle metodologie di ragionamento: ad esempio, ogni famiglia utilizza un diverso modo per gestire l'espansione indeterministica delle disgiunzoni. Nonostante tale scel-

ta sia sembrata la migliore e la più elegante, essa tuttavia non è vincolante: a livello puramente logico, infatti, è possibile creare una qualsiasi gerarchia di classi, a patto che queste rispettino semplicemente un particolare formato sui nomi (nella fattispecie, tutte le strategie devono appartenere a una classe il cui nome inizi con "Strategy").

4.2 $\mathcal{ALC}$

La strategia per $\mathcal{ALC}$ è stata la prima ad essere sviluppata e segue quasi alla lettera le definizioni date all'interno della descrizione classica dell'algoritmo: l'oggetto *Tableaux* contiene una pila di *Node*, che viene inizializzata con il solo nodo radice e alla quale vengono aggiunti tutti i nodi che sono stati modificati o creati ex novo; il metodo `applyRules()` si occupa di estrarre, finché è possibile, un nuovo nodo dalla cima della pila, e applicare ad esso, in sequenza, tutte le regole fino alla completa espansione o alla generazione di un clash.

Delle regole utilizzate da $\mathcal{ALC}$, alcune fanno parte del corredo di tutte le altre logiche e per tale motivo sono state inserite all'interno della classe *Strategy*, dalla quale la maggior parte delle strategie eredita i metodi principali. In particolare fanno parte di *Strategy*:

- `applyUnfoldRule(Node n)`, che si occupa di ricevere un nodo n ed eseguire su di esso la procedura di *unfolding* (o, per meglio dire, di *lazy unfolding*: si veda, a proposito, il Capitolo 5);
- `applyBCP(Node n)`, che riceve un nodo n e vi applica la procedura di Boolean constraint propagation (BCP) per minimizzare il numero di espansioni non deterministiche [Freeman, 1995];
- `applyAndRule(Node n)`, che riceve un nodo n in ingresso e vi applica la "regola $\sqcap$", creando una nuova *assertion*, all'interno del nodo, per ognuno dei termini dell'AND.

I metodi che, invece, sono implementati all'interno di *StrategyALC* sono:

- `applySomeRule(Node n)`, che verifica nel nodo n la presenza di enunciati del tipo $\exists R.C$ e, quando ne trova uno, crea un nuovo nodo dotato di un solo enunciato C e lo collega a n tramite un arco etichettato con il nome R;
- `applyAllRule(Node n)`, che verifica nel nodo n la presenza di enunciati del tipo $\forall R.C$ e, quando ne trova uno, aggiunge in tutti i nodi

collegati a n tramite un arco etichettato R il concetto C (qualora esso non compaia già fra gli enunciati);

- `applyOrRule(Node n)`, che si occupa di gestire l'espansione non deterministica degli OR.

Fra i metodi descritti, quello dedicato all'espansione degli OR merita un approfondimento. Infatti in letteratura ([Horrocks, 1997]) tale procedura viene descritta nel modo seguente:

1. *salva* il tableaux **T**
2. dato il nodo x e il concetto $(C_1 \sqcup C_2) \in \mathcal{L}(x)$, prova ad aggiungere C_1 a $\mathcal{L}(x)$. Se quest'operazione conduce a un clash allora *ripristina* **T** e
3. prova ad aggiungere C_2 a $\mathcal{L}(x)$

Per poter portare a termine tale operazione, si è reso necessario produrre del codice in grado di salvare e ripristinare il tableaux ogniqualvolta questo fosse necessario: sono stati quindi implementati dei metodi `clone()`, in grado di fare una *deep copy* del tableaux e dei nodi in esso contenuti, e all'interno della regola è stato creato un sistema per lavorare su copie del tableaux principale, che vengono estese di volta in volta con i diversi termini dell'OR.

Questa scelta implementativa, anche se molto intuitiva e fedele alla descrizione che ne viene fatta in letteratura, non è la migliore dal punto di vista dell'efficienza: il salvataggio del tableaux, infatti, potrebbe risultare oneroso per basi di conoscenze particolarmente grosse o semplicemente per logiche più espressive. Tale scelta, quindi, è rimasta limitata alle logiche $\mathcal{ALC}$ e $\mathcal{ALCF}$, mentre per le successive sono state adottate soluzioni differenti.

4.3 $\mathcal{ALCF}$

La strategia per la logica $\mathcal{ALCF}$ deriva la quasi totalità delle sue regole da quella creata per $\mathcal{ALC}$: l'unica differenza sta nella regola relativa al quantificatore esistenziale, che deve tener conto dell'esistenza dei ruoli funzionali durante la fase di creazione di nuovi nodi.

Questo legame fra le due logiche si traduce, a livello implementativo, in ereditarietà fra classi: infatti, le regole per la strategia $\mathcal{ALCF}$ sono salvate all'interno di una classe *StrategyALCf*, che eredita buona parte dei metodi di ALC e ne ridefinisce uno soltanto. Il nuovo metodo `applySomeRule` riceve in ingresso un nodo n ed esegue le seguenti operazioni:

1. se constata la presenza di un concetto del tipo $\exists A.C$ (dove A è un nome di ruolo funzionale e C un nome di concetto), verifica l'esistenza di nodi successori, connessi a n tramite archi con etichetta A;

2. se non ne esistono, allora crea un nuovo nodo contenente il solo concetto C;

3. in caso contrario, *aggiunge* tale concetto (qualora esso non sia già presente) all'interno dell'unico nodo A-successore;

4. al termine delle operazioni, il nodo appena creato (o quello modificato) viene accodato per un'eventuale espansione.

4.4 $\mathcal{ALCH}_{R^+}$ ($\mathcal{SH}$)

L'algoritmo utilizzato in letteratura ([Horrocks, 1997]) per $\mathcal{SH}$ e $\mathcal{SHF}$ utilizza un metodo completamente diverso per gestire l'espansione delle disgiunzioni: anzichè salvare e ripristinare il tableaux, esso introduce una nuova tipologia di archi (gli archi anonimi, utilizzati per ricercare le diverse scelte messe a disposizione da un OR), delle nuove definizioni e una nuova regola chiamata **SAT**.

In particolare, per quanto riguarda le definizioni sono stati creati i seguenti metodi all'interno della classe *Node*:

- `preTableaux()` verifica che il nodo corrente sia un *pre-tableaux*, controllando che non si verifichi un clash al suo interno (tramite il metodo `clashes`) e che non contenga congiunzioni o disgiunzioni ancora non espanse;

- `getAncestors()` restituisce l'elenco di nodi che sono antenati del nodo corrente. Per eseguire questa operazione viene creata una *NodeList* all'interno della quale viene salvato, innanzi tutto, il nodo corrente (la relazione è riflessiva), quindi ricorsivamente vengono letti i nodi genitore (salvati in un attributo) fino ad arrivare al nodo radice;

- `getChildren(String EdgeName)` restituisce l'elenco di nodi che sono R-successori. Per eseguire questa operazione, il metodo agisce da filtro sull'elenco di archi che si dipartono dal nodo (salvato in un attributo) e accoda all'interno di una *NodeList* solo quelli connessi tramite un arco che ha il nome specificato in `EdgeName`;

- `getUSuccessors()` restituisce l'elenco dei nodi che sono $\sqcup$-successori. Per eseguire questa operazione viene creata una *NodeList* all'interno

della quale viene salvato, innanzi tutto, il nodo corrente (la relazione è riflessiva). Ad esso vengono aggiunti, in seguito, tutti gli R-successori per i quali R è uguale a un valore convenzionale usato solo per gli archi anonimi (nella fattispecie, è stato usato il valore `::OR::`). Quest'ultima operazione viene eseguita ricorsivamente su tutti i nodi ottenuti.

Per quanto riguarda le regole, invece, seguono i metodi che sono stati ridefiniti o creati *ex novo*:

- `apllySomeRule` incorpora le funzioni precedentemente delegate a entrambi i metodi `applySomeRule` e `applyAllRule`. Inoltre, implementa una strategia di *blocking*, in grado di garantire la terminazione nel caso in cui il concetto da espandere sia del tipo $D = \exists R.C \sqcap \forall R.(\exists R.C)$ e $R \in \mathbf{R}_+$;

- `applyOrRule` applica l'espansione delle disgiunzioni senza più creare una copia del tableaux, bensì costruendo, per ogni termine dell'OR, un nuovo nodo. Ognuno di questi nodi è connesso al padre tramite un arco anonimo e contiene, come etichette, tutte quelle del padre più quella corrispondente al termine che l'ha generato;

- `applySATRule` applica le regole di espansione al nodo in esame e ne verifica, ricorsivamente, la soddisfacibilità. Questa regola costituisce una delle principali novità dal punto di vista implementativo della strategia, gestendo allo stesso tempo l'ordine in cui vengono eseguite le regole (operazione in precedenza delegata ad `applyRules`), la verifica di soddisfacibilità di un nodo e la ricerca della soluzione fra i diversi rami generati non deterministicamente all'interno dell'albero.

4.5 $\mathcal{SHF}$

Come avviene per la logica $\mathcal{ALCF}$ rispetto ad $\mathcal{ALC}$, anche $\mathcal{SHF}$ eredita buona parte delle sue regole dalla logica $\mathcal{SH}$. La differenza fra le due, infatti, riguarda esclusivamente la regola $\exists$ e la classe *StrategySHf*, di conseguenza, eredita i suoi metodi da *StrategySH* ridefinendone uno soltanto.

Il metodo `applySomeRule`, infatti, è ora diviso (non solo a livello concettuale, come descritto nella Tabella 2.5, ma anche dal punto di vista del codice) in due parti distinte: la prima è assolutamente identica a quella presente nella regola di $\mathcal{SH}$, mentre la seconda è stata progettata per gestire la presenza di ruoli funzionali.

Oltre alla semplice modifica del codice della regola $\exists$, è stato necessario creare alcuni metodi di supporto. In particolare:

- poiché, a causa dell'interazione fra gerarchie di ruoli e ruoli funzionali, può capitare che alcuni nodi vengano raggruppati e che agli archi vengano attribuiti *insiemi* di etichette anziché etichette singole, è innanzi tutto necessario che la strategia $\mathcal{SHF}$ sia in grado di gestire questi insiemi: per questo motivo, è stato creato appositamente un metodo `getNameLabels` all'interno della classe *Edge* e, in fase di creazione di A-successori, si marca l'arco che unisce il nodo corrente ai nuovi figli con un insieme di etichette anziché con un'etichetta singola;
- il metodo `getConstraints`, una volta ricevuto un attributo a e un nodo n, restituisce l'insieme di attributi che sono *vincolati* da a in $\mathcal{L}(n)$. Analogamente, è stato definito un metodo per stabilire se un attributo è *direttamente vincolato* da un altro.

4.6 $\mathcal{SI}$

Nonostante la logica $\mathcal{SI}$ sia dotata, rispetto alle ultime descritte, di un numero inferiore di possibilità a livello espressivo, il suo algoritmo tende a complicarsi rispetto ai precedenti: l'introduzione dei ruoli inversi, infatti, rende impossibile l'utilizzo degli archi anonimi usati per $\mathcal{SH}$ e $\mathcal{SHF}$; d'altra parte, l'aumento di espressività rispetto ad $\mathcal{ALC}$ rende l'operazione di copia e ripristino del tableaux troppo onerosa. Si rende necessario, quindi, l'utilizzo di un nuovo metodo per la gestione dell'espansione indeterministica delle disgiunzioni.

Il metodo studiato per consentire delle rapide operazioni di salvataggio e di rollback si basa fortemente sull'idea della *scelta* di un singolo termine rispetto agli altri contenuti all'interno di una disgiunzione: tramite delle etichette, infatti, vengono marcati tutti i nodi, gli archi e i concetti che sono stati creati o modificati in seguito a una di queste scelte; in questo modo, quando l'algoritmo constata che essa porta a un clash ha la possibilità di rintracciare tutti i cambiamenti fatti e annullarli, in modo da ripristinare lo stato del tableaux al momento precedente alla scelta.

Pur essendo più oneroso, a livello implementativo, rispetto a una semplice copia del tableaux, il metodo delle etichette è decisamente più economico in termini di spazio e sicuramente più rapido nella fase di esplorazione iniziale: supponendo, infatti, che il primo clash nel tableaux venga individuato solo dopo l'espansione in catena di cinque diversi OR, con il vecchio algoritmo usato per $\mathcal{ALC}$ si sarebbero dovute effettuare cinque copie del tableaux,

mentre in questo caso l'algoritmo non fa altro che aggiungere delle etichette a degli elementi dell'albero.

Esiste, al momento, una soluzione intermedia fra la copia completa di un tableaux e l'uso capillare di etichette: essa consiste nel salvataggio e ripristino di interi nodi, effettuato pero' in modo *lazy*, cioè solo quando il nodo viene effettivamente modificato. Tale soluzione è adottata in FaCT ed è descritta in [Tsarkov and Horrocks, 2005] e attualmente si sta cercando di valutare l'efficienza, in termini di tempo e di spazio, del metodo delle etichette rispetto ad essa.

Oltre a ridefinire completamente la gestione delle disgiunzioni (operazione che, di per sè, ha dato origine a una nuova famiglia di strategie), all'interno di *StrategySI* sono stati definiti dei nuovi metodi di supporto alle regole della strategia:

- `getRNeighbours` restituisce, all'interno di una *NodeList*, l'elenco dei nodi che sono R-vicini del nodo n, specificato come argomento. Seguendo la definizione presente nella Sezione 2.7, vengono considerati R-vicini tutti i nodi che sono R-successori o $\mathsf{Inv}(R)$-predecessori di n;

- `isBlocked` restituisce `true` se il nodo è *bloccato*, `false` se non lo è. Si ricorda che un nodo x è bloccato se, per qualche antenato y, y è bloccato o $\mathcal{L}(x) = \mathcal{L}(y)$;

- `isIndirectlyBlocked` restituisce `true` se il nodo è *indirettamente bloccato*, `false` se non lo è. Si ricorda che un nodo x è indirettamente bloccato se il suo predecessore è bloccato, altrimenti esso è *direttamente bloccato*.

Ai metodi appena descritti si aggiungono, all'interno della classe *Role*, `getInverse` e `isTransitive`, che consentono, rispettivamente, di ottenere il ruolo inverso a quello specificato e di stabilire se un ruolo è transitivo oppure no. Inoltre, i seguenti metodi sono stati ridefiniti:

- `applyAndRule` è praticamente identico al suo omonimo in *Strategy*, ma prima di applicare l'espansione verifica che il nodo n non sia indirettamente bloccato;

- `applyOrRule` verifica innanzitutto che il nodo n non sia indirettamente bloccato, quindi esegue l'espansione degli OR come descritto in precedenza;

- `applySomeRule` verifica che il nodo n non sia bloccato e che non esistano R-successori y per i quali $C \in \mathcal{L}(y)$. Se queste condizioni sono

rispettate, crea un nuovo nodo contenente il concetto C e collegato al nodo corrente tramite un arco chiamato R;

- `applyAllRule` verifica innanzitutto che il nodo n non sia indirettamente bloccato. Quindi, applica entrambe le regole $\forall$ e $\forall_+$ descritte in teoria, aggiungendo rispettivamente C e $\forall R.C$ (se R è transitivo) a tutti gli R-vicini che non li contengano già.

4.7 $\mathcal{SHI}$

La strategia per la logica $\mathcal{SHI}$ viene costruita in modo incrementale rispetto ad $\mathcal{SI}$: come è possibile intuire dalla Sezione 2.8, infatti, l'aggiunta del supporto per le gerarchie di ruoli non richiede particolari modifiche all'algoritmo, se non in alcune definizioni e nella gestione del quantificatore universale.

A livello implementativo, la classe *StrategySHI* eredita i suoi metodi da *StrategySI* e ridefinisce i seguenti:

- `applyAllRule` tiene conto della presenza della gerarchia di ruoli e applica i concetti associati al quantificatore universale a tutti gli R-vicini del nodo specificato;
- `getRNeighbours` ridefinisce il concetto di R-vicino, tenendo conto della gerarchia di ruoli e della sua chiusura riflessiva e transitiva.

Entrambi i metodi si appoggiano a un nuovo metodo, introdotto all'interno della classe *Role*: il suo nome è `impliesTr` e, grazie ad esso, è possibile ottenere i super-ruoli di un ruolo R, tenendo conto della chiusura transitivo-riflessiva di $\sqsubseteq^*$ rispetto al normale operatore $\sqsubseteq$.

4.8 $\mathcal{SHIF}$

Come la strategia per $\mathcal{SHI}$ è stata costruita basandosi su $\mathcal{SH}$, a sua volta quella per $\mathcal{SHIF}$ eredita buona parte dei suoi metodi da *StrategySHI*. Tuttavia, l'interazione fra ruoli funzionali e gerarchie di ruoli richiede la modifica di alcuni metodi, insieme alla creazione di quelli specifici per la gestione degli attributi. In particolare:

- se un nodo ha più R-vicini di quanti la restrizione funzionale ($\leqslant 1 R$) non consenta, è necessario fondere assieme questi nodi all'interno di uno solo. Tuttavia, poiché essi possono anche essere vicini rispetto

ad altri ruoli S, S' che non sono confrontabili tramite l'operatore $\sqsubseteq^*$, allora il nodo che si viene a creare può essere non solo un R-vicino, ma anche un S- e un S'- vicino. Per gestire questo caso, gli archi vengono etichettati con *insiemi* di ruoli e non con ruoli singoli;

- a causa della modifica appena descritta, anche le definizioni di vicino e successore devono essere cambiate;
- i metodi `isBlocked` e `isIndirectlyBlocked` sono cambiati, per riflettere la diversa strategia di blocking (v. Sezione 2.9);
- sono state aggiunte due nuove regole di espansione per le restrizioni funzionali, e il metodo `applySomeRule` è stato modificato per gestire gli archi etichettati con insiemi di ruoli;
- la definizione di *clash* è stata estesa per gestire quei casi in cui ci sono delle restrizioni funzionali in conflitto fra loro.

Capitolo 5

Ottimizzazioni

Gli algoritmi di tableaux, pur fornendo risultati soddisfacenti in pratica, hanno pur sempre una complessità esponenziale: per questo motivo, l'utilizzo di procedure di ottimizzazione è talvolta non solo auspicabile, ma necessario per ottenere risposte in tempi accettabili. Basti pensare che, per alcune ontologie, il semplice cambiamento dell'ordine in cui vengono applicate le regole dell'algoritmo di tableaux puo' portare a ridurre il tempo di esecuzione a un decimo rispetto al caso pessimo (v. Tabella 5.7); in diversi casi, addirittura, l'applicazione delle procedure di ottimizzazione ha permesso ad alcuni problemi, che erano rimasti irrisolti dopo ore di elaborazione, di trovare una soluzione nel giro di pochi millisecondi ([Horrocks and Patel-Schneider, 1999], [Massacci, 1999]).

JODIE prevede l'utilizzo di diverse procedure di ottimizzazione, in supporto alle operazioni di ragionamento. Non tutte le ottimizzazioni descritte in letteratura sono ancora state implementate, tuttavia buona parte di esse è già pronta e, grazie alla modularità dell'applicazione, è possibile aggiungerne di nuove in modo piuttosto semplice.

All'interno di questo capitolo sono descritte le diverse tecniche di ottimizzazione implementate all'interno di JODIE. Nella prima sezione esse vengono suddivise per tipologie, quindi nelle successive compare la loro descrizione in dettaglio.

5.1 Tipologie di ottimizzazioni

Le tecniche di ottimizzazione utilizzate all'interno di JODIE possono essere suddivise per tipologia, in base a diverse distinzioni. Una prima considerazione che si può fare è che spesso i reasoner non si limitano a prova-

re la soddisfacibilità di un singolo concetto, ma usano i propri algoritmi di ragionamento per calcolare una gerarchia di concetti, cioè un loro ordinamento parziale in base alla relazione di sussunzione. Tale operazione (chiamata *classificazione*) è, naturalmente, passibile a sua volta di ottimizzazione. La prima distinzione, quindi, può essere effettuata fra le tecniche destinate a velocizzare il singolo test di soddisfacibilità e quelle destinate a minimizzare il numero di test necessari per classificare un'ontologia ([Baader et al., 1993], [Horrocks, 1997]). Allo stato attuale, all'interno di JODIE sono state implementate solo tecniche del primo tipo.

Una seconda suddivisione può essere effettuata in base alla necessità o meno di intervenire modificando, in qualche modo, i termini logici che compongono i concetti:

- le tecniche che modificano i termini sono, ad esempio, la *semplificazione* e la *normalizzazione*, il *lazy unfolding* e la *boolean constraint propagation*;

- le tecniche che preservano lo stato dei termini sono quelle che lavorano sull'ordine di esecuzione delle varie porzioni di codice: in particolare, la scelta di usare algoritmi *depth first* e gli studi sull'*ordine di applicazione delle regole*. Inoltre, le varie tecniche di *caching* (che possono essere applicate in diversi contesti) consentono un miglioramento delle performance senza modificare direttamente i termini.

Un'ultima distinzione può essere effettuata fra le tecniche di ottimizzazione che privilegiano la riduzione dello *spazio* occupato (a livello di strutture dati necessarie all'algoritmo) e quelle che, invece, mirano soprattutto a ridurre il *tempo* di esecuzione. Posto che comunque, per un algoritmo di tableaux, la velocità è un requisito essenziale e dev'essere l'obiettivo primario da perseguire in fase di ottimizzazione, l'occupazione di spazio può diventare un problema qualora si decida di lavorare su basi di conoscenze piuttosto grosse. Per questo motivo, sono state ideate alcune tecniche per ridurre lo spazio occupato dalle strutture dati dell'algoritmo: ad esempio, è possibile cancellare elementi del tableaux che non servono più (etichette di nodi già espansi o interi nodi già marcati come non soddisfacibili), ma tali operazioni diventano via via più complesse o, addirittura, non più praticabili quando si prendono in considerazione logiche più espressive. Per fortuna esistono altri metodi, come ad esempio la scelta di algoritmi *depth first* per l'espansione dell'albero, che riescono a dare buoni risultati in ogni occasione.

In base alla distinzione fra spazio e tempo, le tecniche di ottimizzazione

usate all'interno di JODIE e citate in precedenza possono essere suddivise come segue:

- naturalmente, l'uso di algoritmi *depth first* rientra nella categoria delle ottimizzazioni rivolte a diminuire lo spazio occupato;
- le principali tecniche di ottimizzazione usate per aumentare la velocità dell'algoritmo (e quindi diminuirne il *tempo* di esecuzione) sono *lazy unfolding*, *caching* e un'oculata scelta dell'*ordine di applicazione delle regole*;
- vi sono, infine, alcune tecniche come *BCP* e *semplificazione* che, eliminando termini ridondanti, danno buoni risultati sia dal punto di vista del tempo di esecuzione sia per lo spazio occupato.

5.2 Normalizzazione e semplificazione

Fra le diverse tecniche di ottimizzazione, quella che consiste nel semplificare i termini dagli elementi ridondanti è una delle operazioni più semplici da portare a termine, ma anche una di quelle che sortisce migliori risultati. Grazie alla semplificazione, infatti, è possibile individuare eventuali clash prima ancora di applicare una delle regole di espansione; inoltre, anche nel caso in cui in un concetto non vi siano contraddizioni, la sua versione semplificata sarà comunque allo stesso tempo più compatta e più veloce da usare.

Il presupposto su cui si basa l'operazione di normalizzazione è che gli stessi concetti possono essere espressi con formule fra di loro equivalenti. Ad esempio, *"tutte le mamme sono belle"* e *"non esiste una mamma che non sia bella"* sono due modi differenti per esprimere lo stesso concetto, così come *"non posso, allo stesso tempo, andare in vacanza e finire il lavoro"* può essere trasformato in *"o non vado in vacanza, o non finisco il lavoro"* conservando il medesimo significato. Per questo motivo, prima di semplificare un termine si è soliti *normalizzarlo*, in modo da poterlo ricondurre a una forma "standard" più semplice da analizzare.

In particolare, le trasformazioni che l'operazione di normalizzazione attua sui termini che costituiscono un concetto sono mostrate nella Tabella 5.1. Si noti che, nel caso in cui il concetto da normalizzare sia complesso, tale operazione viene ripetuta in modo ricorsivo su tutti i suoi sottoconcetti.

Una volta eseguita la normalizzazione, è possibile proseguire con la semplificazione: anche questa procedura è ricorsiva quando opera su elementi complessi, e consente non solo di ottenere forme più compatte dei medesimi concetti ma anche, in taluni casi, di individuare clash ovvi (come,

concetto semplice C	$\longrightarrow$	C
or(A, B, C)	$\longrightarrow$	not(and(not(A), not(B), not(C)))
some(R, C)	$\longrightarrow$	not(all(R, not(C)))
atm(n, R, C)	$\longrightarrow$	not(atl(n+1, R, C))

Tabella 5.1: Le operazioni compiute dal metodo normalize.

concetto semplice C	$\longrightarrow$	C
not(not(C))	$\longrightarrow$	C
and(A, B, and(C, D))	$\longrightarrow$	and(A, B, C, D)
and(A, A, B, C, C, C, D)	$\longrightarrow$	and(A, B, C, D)
and(A, B, not(B), C)	$\longrightarrow$	BOTTOM
and(A, B, C, TOP, D)	$\longrightarrow$	and(A, B, C, D)
and(A, B, C, BOTTOM, D)	$\longrightarrow$	BOTTOM
and(all(R, C), all(R, D))	$\longrightarrow$	and(all(R, and(C, D)))
and(C)	$\longrightarrow$	C
all(R, TOP)	$\longrightarrow$	TOP
atl(0, R, C)	$\longrightarrow$	TOP

Tabella 5.2: Le semplificazioni operate dal metodo simplify.

ad esempio, quello generato da un `and(C,not(C))`). Le regole usate dalla procedura di semplificazione sono specificate nella Tabella 5.2; all'interno della Figura 5.1, invece, è possibile seguire le operazioni di normalizzazione e semplificazione che vengono eseguite sul concetto complesso `and(all(R, or(D, not(C), not(E)), some(R, and(C, E, not(D)))`, seguendo questi passi:

1. il concetto viene spezzato nelle sue due componenti:
   `all(R, or(D, not(C), not(E))` e `some(R, and(C, E, not(D))`;

2. `some` e `or` vengono normalizzati come mostrato nella Figura 5.1;

3. il nuovo `all` viene semplificato, eliminando le doppie negazioni;

4. alla fine si giunge a una forma del tipo `and(not(F),F)`, dove `F` corrisponde a `all(R,not(and(not(D),C,E)))`. È stato individuato un clash e quindi la funzione restituisce `BOTTOM`.

Normalizzazione:
some(R, and(C, E, not(D)))
  ⟶ not(all(R, not(and(C, E, not(D)))))
all(R, or(D, not(C), not(E)))
  ⟶ all(R, not(and(not(D), not(not(C)), not(not(E)))))

Semplificazione:
all(R, not(and(not(D), not(not(C)), not(not(E)))))
  ⟶ all(R, not(and(not(D), C, E)))
and(not(F),F)
  ⟶ BOTTOM

Figura 5.1: Un esempio di semplificazione che porta all'individuazione di un clash.

5.3 Lazy unfolding

La procedura di *unfolding* è un'operazione di semplificazione che consiste nel sostituire, all'interno di una espressione, tutte le occorrenze di uno o più concetti non primitivi con le loro rispettive definizioni. Tale espansione viene solitamente ripetuta finché all'interno dell'espressione non compaiono solamente concetti primitivi: in questo modo, al termine dell'operazione sarà più semplice confrontare fra di loro i vari termini al fine di individuare dei clash.

Solitamente, durante la procedura di unfolding, si è soliti sostituire un concetto C, all'interno di un'espressione E, con un altro concetto D tale che $C \equiv D$. Tuttavia, è anche possibile eseguire la stessa operazione per i D tali che $C \sqsubseteq D$, a patto di ricordare che l'espressione risultante E' non è più equivalente ad E. Ad esempio, data l'ontologia

$$\begin{aligned} &\text{Organismo} \equiv \text{Animale} \sqcup \text{Vegetale} \\ &\text{Erba} \sqsubseteq \text{Vegetale} \\ &\text{Mucca} \sqsubseteq \text{Animale} \sqcap \forall \text{mangia.Erba} \end{aligned}$$

è possibile espandere la definizione di "Mucca", in modo da ottenere che è un animale che mangia dei vegetali. Questa definizione è ancora corretta, ma è meno specifica dell'originale (infatti, una mucca non mangia *qualsiasi* tipo di vegetale).

Allo stesso modo, data la seguente ontologia

$$\begin{aligned}
&\text{Organismo} \equiv \text{Animale} \sqcup \text{Vegetale}\\
&\text{Persona} \sqsubseteq \text{Animale}\\
&\text{Erba} \sqsubseteq \text{Vegetale}\\
&\text{Mucca} \sqsubseteq \text{Animale} \sqcap \forall\text{mangia.Erba}\\
&\text{Carnivoro} \equiv \text{Organismo} \sqcap \forall\text{mangia.Animale}\\
&\text{Rancher} \equiv \text{Persona} \sqcap \forall\text{mangia.Mucca} \sqcap \exists\text{possiede.Ranch}
\end{aligned}$$

è possibile eseguire l'unfold di "Rancher" fino ad ottenere che è un organismo che mangia solo animali:

$$\begin{aligned}
\text{Rancher} \equiv\ &\text{Persona'} \sqcap \text{Animale'} \sqcap \text{Organismo} \sqcap\\
&\forall\text{mangia.}(\text{Mucca'} \sqcap \text{Animale} \sqcap \forall\text{mangia.Erba}) \sqcap\\
&\exists\text{possiede.Ranch}
\end{aligned}$$

Da questa espansione si possono trarre alcune conclusioni:

- innanzitutto, è possibile notare che ora è molto semplice confrontare i concetti "Rancher" e "Carnivoro" e stabilire che Rancher $\sqsubseteq$ Carnivoro;
- inoltre, si può anche facilmente dimostrare che non è vero il contrario, cioè Carnivoro $\not\sqsubseteq$ Rancher;
- durante il processo di espansione si è deciso di aggiungere, anziché sostituire, i nuovi concetti ai vecchi, marcando questi ultimi con un apice. Quest'operazione consente di effettuare contemporaneamente più confronti su diversi livelli dell'ontologia e, di conseguenza, permette di individuare più facilmente eventuali clash (come già notato in [Baader et al., 1993], [Horrocks, 1997]).

Tutto quanto descritto finora è stato implementato in JODIE, all'interno del metodo `lazyUnfold` della classe *Term*, che si occupa di suddividere espressioni complesse in concetti semplici, e nel metodo `lazyExpand` della classe *Concept*, che recupera dalla base di conoscenze i concetti equivalenti a quello che si desidera espandere (o quelli da esso implicati).

Come suggerito dagli stessi nomi dei metodi creati all'interno di JODIE, la procedura di unfolding è *lazy*: questo significa che l'espansione del concetto non avviene immediatamente, ma solo nel momento in cui esso viene utilizzato dall'algoritmo. Ad esempio, cercando di verificare la soddisfacibilità dell'espressione $\exists R.\mathsf{CN}$ (dove CN è un nome di concetto), l'unfolding di CN può essere ritardato fino al momento in cui la regola $\exists$ abbia creato un R-successore y con $\mathcal{L}(y) = \{\mathsf{CN}\}$: a questo punto è possibile sostituire, all'interno di $\mathcal{L}(y)$, il nome del concetto con la forma normale negata della sua definizione.

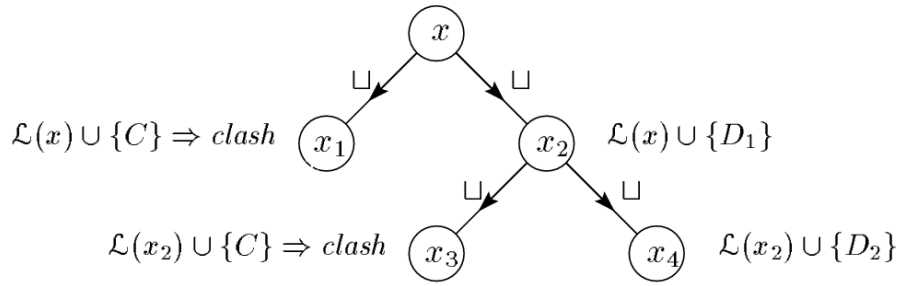

Figura 5.2: Nell'espansione del nodo x, con $\{(C \sqcup D_1), (C \sqcup D_2)\} \subseteq \mathcal{L}(x)$ e C non soddisfacibile, il syntactic branching opera in modo inefficiente.

Infine, i metodi per l'unfolding sono stati adattati in modo da poter gestire anche *terminologie cicliche*. Se da una parte, infatti, è vero che la procedura di espansione viene eseguita ricorsivamente, dall'altra il metodo, ad ogni sua chiamata, verifica che il concetto da espandere non sia ancora stato "incontrato": se esso è nuovo, allora l'algoritmo prosegue; in caso contrario viene restituito il termine inespanso. Questo garantisce la terminazione della procedura di unfolding anche nel caso in cui le terminologie adottate non siano acicliche.

5.4 Boolean Constraint Propagation

Gli algoritmi di tableaux, nella loro implementazione standard, soffrono di un'inefficienza intrinseca dovuta all'utilizzo del cosiddetto *syntactic branching* (v. Figura 5.2). Questa tecnica consiste nello scegliere una disgiunzione non ancora espansa ed effettuare una ricerca all'interno dei diversi modelli che si possono ottenere valutando, uno per volta, i suoi termini (si consulti, ad esempio, [Giunchiglia and Sebastiani, 1996]).

Per fare fronte a questa inefficienza sono state proposte diverse soluzioni: alcune di esse, come ad esempio il *semantic branching* (descritto in dettaglio all'interno di [Horrocks, 1997], [Horrocks et al., 2000]) propongono una modifica al metodo stesso adottato per effettuare la ricerca, mentre altre operano a monte, cercando di diminuire il numero di alternative da valutare in fase di ricerca.

La BCP (Booean Constraint Propagation, [Freeman, 1995]) è una tecnica di ottimizzazione che fa parte di questa seconda categoria. Poiché essa viene applicata prima ancora che un qualsiasi algoritmo di ricerca venga av-

viato, può essere utilizzata sia con il syntactic che con il semantic branching e, in entrambi i casi, è in grado di velocizzare notevolmente l'esecuzione dell'algoritmo.

La BCP viene usata per massimizzare l'espansione deterministica (e, quindi, la semplificazione dell'albero di ricerca attraverso l'individuazione di clash) prima di effettuare quella non deterministica. Prima che la regola $\sqcup$ venga applicata all'interno di un nodo x, la BCP espande in modo deterministico le disgiunzioni all'interno di $\mathcal{L}(x)$ che presentano solo una possibilità di espansione, e individua un clash se una disgiunzione all'interno di $\mathcal{L}(x)$ non ha possibilità di espansione. In pratica, la BCP applica la regola d'inferenza

$$\frac{\neg C, C \sqcup D}{D}$$

ad $\mathcal{L}(x)$. Ad esempio, dato un nodo x tale che

$$\{(C \sqcup (D_1 \sqcap D_2)), (\neg D_1 \sqcup \neg D_2), \neg C\} \subseteq \mathcal{L}(x)$$

la BCP espande deterministicamente la disgiunzione $(C \sqcup (D_1 \sqcap D_2))$, poiché $\neg C \in \mathcal{L}(x)$:

$$\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{(D_1 \sqcap D_2)\}$$

Dopo che $(D_1 \sqcap D_2)$ è stato espanso dalla regola $\sqcap$, in modo da ottenere $\mathcal{L}(x) \longrightarrow \mathcal{L}(x) \cup \{D_1, D_2\}$, la BCP considera $(\neg D_1 \sqcup \neg D_2)$ come clash, poichè $D_1 \in \mathcal{L}(x)$ e $D_2 \in \mathcal{L}(x)$.

All'interno di JODIE, la tecnica di BCP viene implementata nel modo seguente:

1. il metodo `applyBCP` è definito all'interno delle classi di tipo strategia e riceve un nodo come parametro di ingresso;
2. una per volta, vengono analizzate tutte le disgiunzioni che compaiono all'interno del nodo;
3. se la disgiunzione contiene un termine che va in clash con le altre asserzioni all'interno del nodo, tale termine viene eliminato;
4. se la disgiunzione contiene un termine complesso `AND`, che a sua volta contiene un sotto-termine che va in clash con le altre asserzioni all'interno del nodo, tutto l'`AND` viene eliminato;
5. se, al termine delle semplificazioni, l'`OR` non contiene più termini, questo significa che ognuno di loro avrebbe generato un clash: di conseguenza, l'intera disgiunzione viene sostituita da un `BOTTOM`;

6. se al termine delle semplificazioni rimane un solo termine (cioè la disgiunzione assume la forma `OR(C)`), questo viene sostituito all'`OR`. Tuttavia, se tale termine esiste già fra le asserzioni del nodo, l'intera disgiunzione viene semplicemente cancellata.

5.5 Caching

L'operazione di *caching* è una delle ottimizzazioni "trasparenti" per l'applicazione: essa, cioè, può essere applicata in ausilio ad algoritmi già esistenti, senza stravolgerli e senza modificare i dati restituiti. A parità di risultato, tuttavia, il tempo di elaborazione è solitamente inferiore.

Il caching, all'atto pratico, non è altro che un salvataggio in memoria del risultato di alcune operazioni che si reputa verranno eseguite nuovamente entro breve tempo. Esso può essere effettuato in diversi momenti all'interno del programma: durante l'esecuzione di un particolare algoritmo (ad esempio, l'espansione di un termine) o fra iterazioni differenti dello stesso (ad esempio, durante la classificazione di un'ontologia, fra un test di sussunzione e l'altro). In ogni caso, il caching consente di ottenere una maggiore velocità di esecuzione al prezzo di un maggiore impiego di memoria: di volta in volta, quindi, è necessario valutare l'opportunità di adottare tale ottimizzazione in base alle dimensioni e alla complessità dell'ontologia in uso.

Allo stato attuale, all'interno di JODIE l'operazione di caching viene effettuata solamente durante la procedura di unfolding dei termini: ad ogni concetto viene associata la sua forma espansa in modo che, a una successiva richiesta di unfolding, sia sufficiente andare a recuperare il valore richiesto in cache anziché doverlo ricalcolare. L'operazione di recupero è rapida, in quanto l'intera struttura dati è salvata all'interno di un hash; inoltre, si è cercato di occupare meno spazio possibile in memoria, salvando all'interno dell'hash gli oggetti più semplici, di tipo *Term*. All'atto pratico, il tutto si traduce nelle seguenti operazioni:

- all'interno del metodo `lazyUnfold` di *Term*, subito prima che venga eseguita l'espansione del concetto corrispondente al termine specificato, viene effettuato un controllo all'interno della cache: se l'hash `exp` (attributo della TBox) contiene già, fra le sue chiavi, tale termine (o, meglio, la sua rappresentazione in formato stringa), allora esso non viene espanso ma viene restituito il valore corrispondente salvato all'interno dell'hash;

- se un termine non è presente all'interno della cache, esso viene espanso:

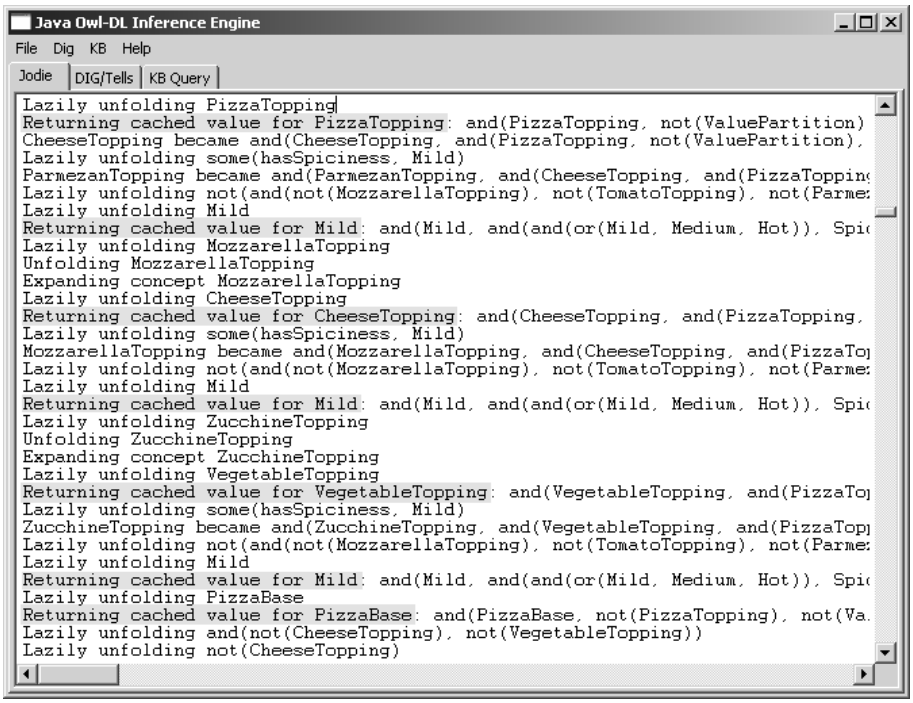

Figura 5.3: Un log di esecuzione del processo di unfolding, con evidenziati gli accessi alla cache.

subito dopo tale operazione, il risultato dell'espansione viene salvato nella cache in modo da poter essere riutilizzato in seguito.

5.6 Depth first search

L'utilizzo di algoritmi *depth first* in fase di ricerca all'interno dell'albero generato dall'algoritmo di tableaux è un'ottimizzazione che consente di diminuire l'utilizzo di memoria necessaria all'applicazione. Poiché, infatti, l'operazione di ricerca avviene in seguito all'espansione non deterministica delle disgiunzioni, è possibile, con un algoritmo depth first, analizzare in profondità un singolo ramo all'interno dell'albero, prima ancora di crearne degli altri.

Questa scelta porta a due conclusioni distinte:

- se il ramo, una volta espanso, non presenta clash, allora costituisce parte del modello che si desidera costruire e non è più necessario espandere gli altri rami;

- in caso contrario, sicuramente il ramo non farà parte del modello: è quindi possibile eliminarlo insieme ad eventuali figli.

Come si può notare, in entrambi i casi l'algoritmo a cui viene applicata la tecnica depth first trae notevoli vantaggi in termini di spazio e, talvolta, anche di tempo di esecuzione. All'interno di JODIE, l'applicazione di tale tecnica si traduce in un particolare ordinamento nella chiamata dei metodi ricorsivi, differente a seconda della famiglia a cui appartiene la strategia in uso. Poiché, infatti, il metodo di gestione dell'espansione delle disgiunzioni cambia per ognuna di queste famiglie, cambia di conseguenza anche l'implementazione *ad hoc* per la tecnica depth first:

- per le strategie della famiglia di $\mathcal{ALC}$, quando viene individuato un `OR` il programma crea un clone del tableaux, che viene valutato in seguito all'inserimento del primo termine della disgiunzione. I termini successivi vengono eventualmente valutati solo dopo aver verificato che il primo causa un clash;

- per le strategie della famiglia di $\mathcal{SH}$, la tecnica di depth first viene sfruttata direttamente all'interno del metodo `applySATRule`, che viene chiamato ricorsivamente per valutare la soddisfacibilità di un nodo (e, più in generale, del tableaux). Un nodo è considerato soddisfacibile se lo sono tutti i suoi R-successori e, quindi, la ricerca eseguita in modo ricorsivo tende a spostarsi verso le foglie. Inoltre, un nodo è soddisfacibile se lo è almeno uno dei suoi $\sqcup$-successori: quindi non appena, fra gli $\sqcup$-successori di un nodo n, ne viene individuato uno soddisfacibile, n viene considerato tale e tutti gli altri $\sqcup$-successori vengono ignorati;

- per le strategie della famiglia di $\mathcal{SI}$, si opera in modo simile alla famiglia di $\mathcal{ALC}$: quando si trova una disgiunzione, se ne valuta il primo termine all'interno di un ambiente che è possibile ripristinare a uno stato precedente, e si passa ai successivi solo se il primo genera un clash. L'unica differenza sta nel modo in cui le operazioni di salvataggio e ripristino sono effettuate, ma questo non influisce sull'applicazione della tecnica depth first.

5.7 Ordine di applicazione delle regole

Un'ultima tecnica di ottimizzazione implementata all'interno di JODIE è quella che consiste nell'adottare un particolare ordine di applicazione delle

	DOLCE		WineFood		GALEN	
	SAT	SUB	SAT	SUB	SAT	SUB
a	0.74	0.74	0.22	2.44	99.44	1678.11
aO	0.64	0.68	**0.14**	**1.64**	29.80	569.64
aEO	**0.58**	**0.57**	0.15	1.67	**9.88**	**173.79**
aE	0.60	0.58	0.27	2.87	13.35	205.32
aOE	0.61	0.59	0.27	2.93	13.22	201.40

Tabella 5.3: I risultati dei test sull'ordine di applicazione delle regole.

regole di espansione. Come descritto in [Tsarkov and Horrocks, 2005], infatti, la corretta scelta di tale ordine può velocizzare notevolmente l'algoritmo di verifica della soddisfacibilità: a dimostrazione di ciò, nella Tabella 5.7 vengono riportati i risultati dei test eseguiti da Horrocks.

Nella prima colonna della tabella viene specificato l'ordine di applicazione delle regole: "O" corrisponde alla regola $\sqcup$, "E" alla regola $\exists$, mentre "a" a qualsiasi altra regola. Ad esempio, "aEO" corrisponde a una strategia in cui la regola $\sqcup$ viene eseguita per ultima ed è preceduta dalla regola $\exists$ che, a sua volta, ha priorità inferiore a tutte le rimanenti.

I valori in tabella sono in parte giustificabili con considerazioni abbastanza semplici: ad esempio, è ovvio che la gestione delle disgiunzioni e la ricerca di una soluzione all'interno dell'albero siano operazioni onerose e, come tali, è bene posticiparle il più possibile. In generale, inoltre, conviene cercare un clash all'interno di un nodo prima ancora di espanderlo: in questo modo, in caso il clash sia effettivamente presente, l'algoritmo risparmierà tutto il tempo che sarebbe altrimenti stato necessario per l'espansione. Per questo motivo, anche la gestione dei quantificatori esistenziali viene posticipata rispetto alle altre regole di espansione.

All'interno di JODIE si è cercato di applicare questi criteri di precedenza, non solo per quanto riguarda le classiche regole di espansione ma anche per i vari metodi di ottimizzazione che ormai sono parte integrante dell'algoritmo di tableaux. In linea di massima, sono state seguite queste linee guida:

1. dato un nodo, viene eseguito un controllo preliminare su di esso in modo da individuare dei clash banali. Qualora essi siano effettivamente presenti, nessun'altra operazione di espansione viene eseguita sul nodo;

2. in seguito si procede con il metodo `applyUnfoldRule`, che si occupa di normalizzare, semplificare ed eseguire l'unfolding dei concetti presenti all'interno del nodo; i clash possono essere individuati durante la proce-

dura di semplificazione oppure in seguito all'unfolding, interrompendo così ogni eventuale espansione successiva;

3. a questo punto segue la regola $\sqcap$ e un nuovo controllo per la presenza di clash;

4. in seguito, viene il turno della regola $\exists$ e della regola $\forall$: poiché esse operano su nuovi nodi, non è necessario avviare, dopo la loro esecuzione, un controllo di clash all'interno del nodo corrente;

5. prima della regola $\sqcup$ viene eseguito il metodo `applyBCP`, in modo da eliminare eventuali termini superflui della disgiunzione;

6. alla fine, con livello di priorità minimo, viene eseguita la regola $\sqcup$.

Si noti che, sebbene quest'elenco di operazioni rispecchi la forma ideale di esecuzione delle regole di espansione, esso non viene sempre rispettato in fase di implementazione: questo è dovuto al fatto che, in alcuni algoritmi, è necessario anteporre particolari operazioni ad altre e quindi venir meno alle indicazioni specificate. Ad esempio, le strategie della famiglia di $\mathcal{SH}$ richiedono, per il test di soddisfacibilità, la verifica della condizione di *pre-tableaux*; questa verifica può avvenire solo dopo l'espansione delle disgiunzioni, quindi la regola $\sqcup$ viene in questo caso anteposta alla $\exists$. Quando possibile, tuttavia, l'ordine specificato viene mantenuto, sia per quanto riguarda l'applicazione delle regole sia, in particolare, per le procedure di ottimizzazione.

Capitolo 6

Risultati sperimentali

Anche se per certi versi si è ispirata a programmi già esistenti, JODIE è un'applicazione progettata e sviluppata da zero, contenente diverse caratteristiche originali che la rendono difficilmente paragonabile ad altri software dello stesso tipo.

Tuttavia, nel tentativo di valutare la validità del progetto, si è ugualmente cercato di individuare una metrica, tramite la quale misurare le prestazioni del programma ed eventualmente paragonarle a quelle degli altri software. Inoltre, tramite l'ausilio di diversi strumenti, è stato verificato il corretto funzionamento del programma e sono stati valutati i vantaggi ottenuti dalle varie procedure di ottimizzazione citate all'interno del precedente capitolo.

All'interno di questo capitolo vengono descritte le diverse tipologie di test eseguiti su JODIE, le motivazioni che hanno portato a sceglierli, gli strumenti a cui si sono appoggiati e i risultati a cui si è pervenuti.

6.1 Strumenti utilizzati

Gli strumenti utilizzati per eseguire i test su JODIE appartengono a due categorie differenti:

- strumenti **esterni**: sono applicazioni indipendenti da JODIE, non necessariamente scritte usando lo stesso linguaggio di programmazione ed eseguibili anche per scopi diversi dal semplice testing;
- strumenti **interni**: sono parte integrante del codice di JODIE, sono scritti in Java e almeno una parte di essi è stata scritta con la sola finalità di eseguire dei test sul programma.

Gli strumenti esterni, in particolare, sono

- gli editor di ontologie **Protégé**, Versione 3.0, Build 141 (con installato il plugin per OWL V1.3, Build 225.4), e **Oiled**, Versione 3.5.7. Essi sono stati usati non solo per creare le ontologie usate nei test, ma anche per verificare il corretto scambio di informazioni attraverso il protocollo DIG;

- il reasoner **RACER**, Versione 1.7.19, usato come metro di paragone per i test sul funzionamento di JODIE;

- l'applicazione **GraphViz**, Versione 2.2.1, grazie alla quale è stato possibile verificare graficamente la correttezza dei tableaux costruiti.

Gli strumenti implementati all'interno di JODIE, invece, sono

- i **timer**, il cui codice è stato integralmente recuperato dal reasoner Pellet. È stato sufficiente, infatti, importare la classe *Timers*, distribuita insieme al codice sorgente di Pellet, per ottenere una serie di cronometri pronti da utilizzare all'interno di JODIE. Ognuno di questi cronometri è stato associato a una particolare azione, in modo da poter misurare non solo il tempo impiegato dal programma per portare a termine un'operazione di ragionamento, ma anche i tempi di esecuzione delle singole funzioni;

- il metodo `toDOT`, che consente di esportare lo stato di un tableaux nel formato DOT. I file in questo formato vengono in seguito passati a GraphViz, per ottenere la rappresentazione grafica dei tableaux;

- il sistema avanzato di **log** adottato all'interno di JODIE permette di seguire in ogni momento il funzionamento del reasoner e di capire quali operazioni vengono eseguite in fase di comunicazione con l'editor di ontologie, durante la creazione della base di conoscenze e infine durante l'intero processo di ragionamento.

Agli strumenti adottati per i test si aggiunge la scelta di alcune **basi di conoscenze**, caratterizzate da diverse espressività e dimensioni, che consentono di verificare il funzionamento di JODIE in situazioni differenti. Alcuni buoni punti di partenza per trovare ontologie sono, ad esempio, i siti `http://www.co-ode.org` (sede, tra l'altro, dell'utilissimo *Protégé OWL tutorial*) e `http://protege.stanford.edu/plugins/owl/owl-library`. Le

ontologie usate per eseguire i test su JODIE, invece, sono una versione personalizzata di `pizza.owl`, il cui file originale si trova su CO-ODE, e una ricostruzione parziale in owl della famosa OpenGALEN, che può essere scaricata da `http://www.cs.man.ac.uk/~rector/ontologies/simple-top-bio`. All'occasione, infine, sono state create delle mini-ontologie *ad hoc*, per verificare il comportamento del reasoner in casi veramente particolari.

6.2 Tipologie di test

I test effettuati su JODIE sono stati suddivisi, per semplicità, in tre categorie: *funzionamento*, *velocità* e *ottimizzazione*.

Funzionamento

Il primo gruppo di test ha lo scopo di verificare il corretto funzionamento dei vari componenti di JODIE e, in particolare, degli algoritmi di ragionamento. Per eseguire tali verifiche sono stati utilizzati sia applicazioni esterne sia strumenti interni, e in particolare:

- usando Protégé e Oiled è stata controllata in modo empirico la correttezza delle comunicazioni effettuate tramite il protocollo DIG: sia in fase di identificazione del reasoner (attraverso il *Reasoner Inspector*, come mostrato nella Figura 6.1), sia per quanto riguarda le risposte alle *tells* e alle *asks*, l'editor di ontologie ha sempre accettato i messaggi ricevuti; in appoggio agli editor di ontologie , i log si sono rivelati uno strumento prezioso per verificare che il formato con cui i dati venivano inviati fosse corretto (v. Figura 6.2);
- la correttezza dei test di soddisfacibilità è stata verificata empiricamente, provando ad eseguire query dal risultato noto attraverso l'interfaccia grafica; in casi più complessi, si sono confrontati i risultati forniti da JODIE con quelli generati da RACER.

Velocità

Per quanto riguarda i test sulla velocità, sono necessari alcuni chiarimenti. Innanzi tutto, in alcuni casi non è significativo o, addirittura, non è possibile effettuare un test comparativo fra JODIE e altri reasoner: ad esempio, mentre si possono paragonare i tempi di esecuzione necessari al singolo test di soddisfacibilità, non ha senso fare lo stesso per la classificazione di un'ontologia. Infatti, come già descritto nel Capitolo 5, dedicato alle ottimizzazioni,

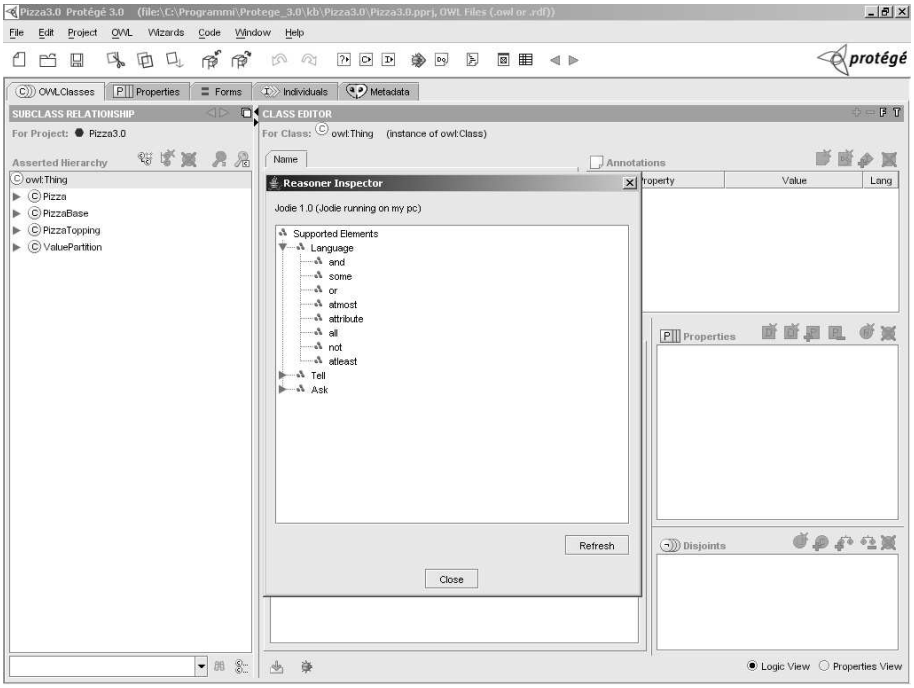

Figura 6.1: Il reasoner inspector di Protégé mostra i dati identificativi inviati da JODIE.

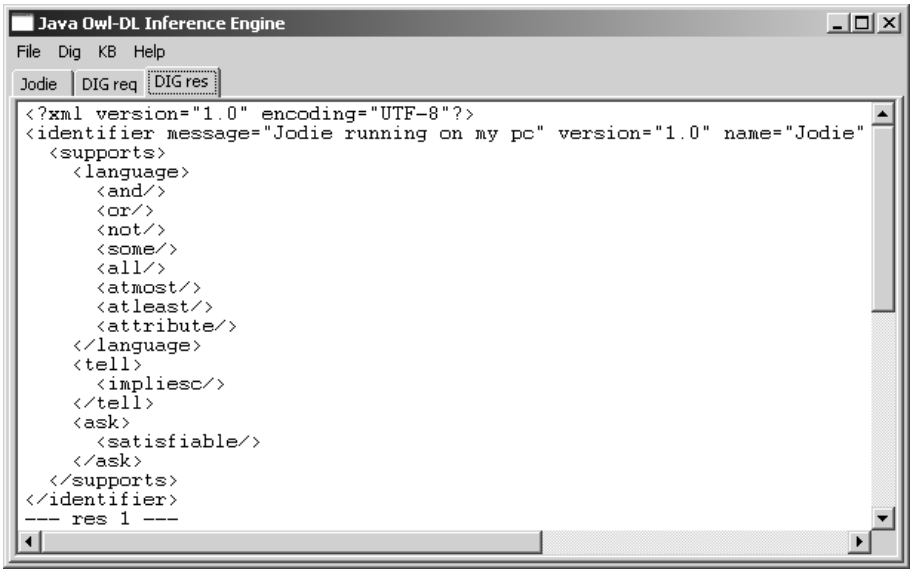

Figura 6.2: Il messaggio di identificazione inviato da JODIE a Protégé, visualizzato all'interno dei log.

```
Time         | Count | Avg                  | Total
=============================================
main         | 1     | 25813.0              | 25813
Dump         | 1     | 1532.0               | 1532
Rulez        | 1     | 578.0                | 578
[TAB][UNF]   | 177   | 2.378531073446328    | 421
DIG Parsing | 1     | 297.0                | 297
[TAB][AND]   | 167   | 0.19161676646706588 | 32
[TAB][BCP]   | 167   | 0.18562874251497005 | 31
[TAB][OR]    | 177   | 0.0                  | 0
[TAB][SOME]  | 10    | 0.0                  | 0
```

Figura 6.3: Un esempio di output generato dai cronometri, attivati su tutte le funzioni dell'algoritmo di tableaux, parsing DIG e dump dei log.

il caching in fase di classificazione ancora non è stato implementato e quindi l'algoritmo non è neanche paragonabile, in termini di efficienza, a quelli degli altri reasoner.

Inoltre, constatata la versatilità dei timer interni a JODIE, che consentono di cronometrare le singole funzioni del programma, si è deciso di utilizzare i test di velocità come strumento per valutare e motivare le diverse scelte implementative (v. Figura 6.3). All'atto pratico, questo si è tradotto nei seguenti test:

- tramite l'utilizzo dei timer, si è calcolata la velocità di esecuzione delle singole query e si è cercato di individuare dove si trovassero eventuali colli di bottiglia;
- sono state proposte delle scelte alternative per i componenti critici: per ognuna di esse, sono state effettuate nuove misurazioni e in base alle loro performance è stata scelta la migliore.

Infine, si tenga presente che tutti i tempi citati in seguito sono stati registrati su un Athlon XP 2400 con 1GB di RAM: una variazione di CPU o, soprattutto, di memoria può cambiare notevolmente i tempi di esecuzione.

Ottimizzazione

Per i test dedicati a valutare l'efficacia delle diverse tecniche di ottimizzazione, si è deciso di adottare dei criteri simili a quelli utilizzati per la categoria precedente: anche questa volta sono stati usati i timer per valutare la velocità delle query; per ogni richiesta inviata al reasoner, inoltre, è stata

calcolata prima la velocità di esecuzione in assenza di ottimizzazioni, quindi quella in presenza di alcune di esse, infine quella del sistema completamente ottimizzato. Questo ha permesso di valutare non solo la bontà delle singole procedure adottate, ma anche la conseguenza della loro mutua interazione.

6.3 Risultati

All'interno di questa sezione vengono descritti e commentati i risultati dei test eseguiti su JODIE, raggruppati in base alla categoria di appartenenza.

Funzionamento

I test eseguiti per verificare il corretto funzionamento di JODIE hanno avuto esito positivo: gli editor di ontologie comprendono i messaggi inviati da JODIE e, naturalmente, JODIE è in grado di capire ciò che le viene comunicato tramite il protocollo DIG (v. Fig. 6.4); inoltre, tutti i test di soddisfacibilità eseguiti hanno avuto l'esito previsto, e le immagini generate fda GraphViz hanno confermato che l'evoluzione del tableaux è stata quella desiderata (v. Fig 6.5).

Velocità

Per quanto riguarda il calcolo della velocità, i test eseguiti hanno permesso di individuare i colli di bottiglia dell'applicazione e di studiare delle alternative per permettere al reasoner di lavorare nel modo più efficiente possibile.

Il primo problema affrontato è stato quello di decidere se utilizzare o meno i JavaBean per il parsing dei messaggi DIG. La velocità, in questo caso, non era l'unico fattore determinante, ma ha sicuramente aiutato nella scelta: com'è stato possibile constatare dai dati sperimentali, infatti, il parsing DIG tramite JavaBean è sempre più oneroso rispetto a quello eseguito con il parser DOM. In particolare, il rallentamento è sensibile in presenza di piccole ontologie: basti pensare che, mentre nell'ontologia GALEN si passa da un tempo di 1047ms per i beans, a uno di 219ms per DOM (quasi cinque volte piu' lento), in quella dedicata alle pizze si passa da un tempo di 843ms a uno di 63ms (più di dieci volte più lento). Se i tempi sembrano comunque troppo piccoli per essere degni di nota, si tenga presente che, come accennato in precedenza, dipendono strettamente dal tipo di CPU e di RAM installate: su macchine più antiquate, essi possono addirittura cambiare di ordine di grandezza.

Gli altri colli di bottiglia individuati sono, com'era facile aspettarsi, tutti quelli relativi all'output dei dati: in particolare a fronte di ontologie medio-

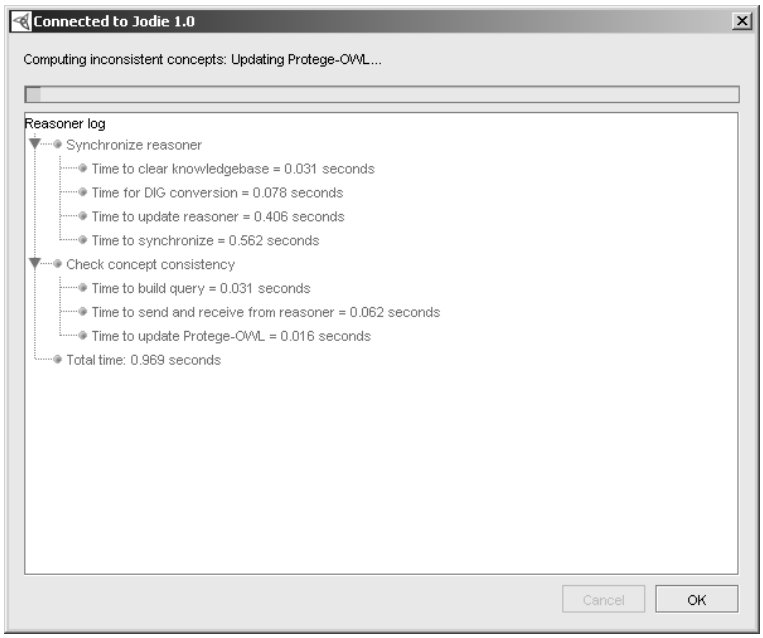

Figura 6.4: Il log di Protégé mostra i dettagli della comunicazione avvenuta con JODIE per effettuare un controllo di consistenza.

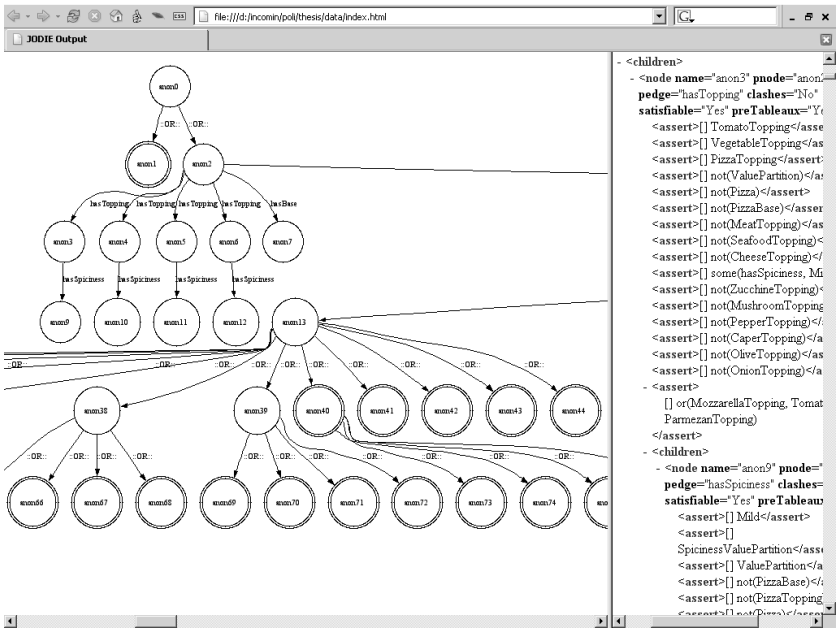

Figura 6.5: Un tableaux particolarmente complesso viene visualizzato all'interno del browser.

piccole, per le quali il tempo di elaborazione non è particolarmente lungo, la grande quantità di log generati e il disegno dell'albero del tableaux possono rallentare notevolmente (da pochi millisecondi fino a qualche secondo, su macchine più lente) l'esecuzione del reasoner. Questo è uno dei motivi per cui si è scelto di consentire all'utente, e non solo al programmatore, di attivare o disattivare i log a piacimento.

Ottimizzazione

Per i test sulle procedure di ottimizzazione è stata scelta una strategia (*StrategySH*), una base di conoscenze (`pizza.owl`) e tre query di complessità crescente, numerate da 1 a 3. Le misure sono state effettuate prima senza ottimizzazioni, quindi riattivando man mano le singole procedure e verificando eventuali interazioni fra di loro.

Naturalmente, non è stato possibile disattivare *tutte* le tecniche di ottimizzazione: infatti alcune di esse, come la ricerca depth first all'interno dell'albero, influiscono direttamente sugli algoritmi e non possono essere modificate in modo semplice. Per questo motivo si è deciso di operare solo su lazy unfolding, semplificazione, caching e BCP.

I risultati dei test sono mostrati all'interno delle tabelle 6.1, 6.2 e 6.3. Le misurazioni sono riferite al tempo di esecuzione (in ms) delle regole dell'algoritmo di tableaux e del dump dei dati su disco: in questo modo è stato possibile farsi un'idea non solo della velocità degli algoritmi ma anche dello spazio in memoria occupato.

Per la query più semplice e per quella più complicata è stato eseguito un ulteriore test: la voce (+Simplify), infatti, corrisponde all'inserimento delle regole di semplificazione *in sostituzione*, e non solo in aggiunta, alla procedura di lazy unfolding. Le conclusioni tratte dall'esito di questi test sono le seguenti:

- l'utilizzo di simplify, specialmente nel caso di query semplici, può risultare oneroso dal punto di vista della CPU, ma sicuramente è vantaggioso dal punto di vista dell'occupazione di spazio in memoria; quando le query aumentano di complessità, il peso dell'operazione di semplificazione tende a farsi sentire di meno mentre il guadagno in termini di spazio rimane sempre piuttosto alto;

- nei casi più complessi, la totale assenza di ottimizzazioni può causare un aumento esponenziale dei tempi di elaborazione: nel terzo test, i risultati sono stati fissati a ∞ poiché completamente fuori scala;

Ottimizzazione	Regole	Dump
Nessuna	93	31
+Lazy unfold	47	16
(+Simplify)	94	15
+Simplify	62	15
+Caching	62	15
+BCP	47	16

Tabella 6.1: I risultati dei test sulla query numero 1.

Ottimizzazione	Regole	Dump
Nessuna	1000	2750
+Lazy unfold	656	2422
+Simplify	672	1985
+Caching	578	1922
+BCP	297	641

Tabella 6.2: I risultati dei test sulla query numero 2.

Ottimizzazione	Regole	Dump
Nessuna	∞	∞
+Lazy unfold	2688	9953
(+Simplify)	4453	8828
+Simplify	2953	7532
+Caching	2703	7531
+BCP	1109	2938

Tabella 6.3: I risultati dei test sulla query numero 3.

in questo caso è stato tuttavia sufficiente applicare anche una sola procedura di ottimizzazione per ottenere dei risultati accettabili;

- lazy unfolding e BCP sembrano essere le due procedure in grado di fornire i risultati migliori: mentre la prima appare più efficace in termini di tempo (in effetti, non fa altro che posticipare alcune operazioni fino al momento in cui esse non siano veramente necessarie), la seconda offre notevoli risparmi sia dal punto di vista del tempo di esecuzione che per quanto riguarda lo spazio occupato;

- la procedura di caching in fase di unfolding consente, effettivamente, di ottenere qualche vantaggio in termini di tempo, ma il più delle volte questi sono trascurabili: essa potrebbe essere addirittura rimossa, consentendo al programma di risparmiare lo spazio occupato dalla cache, in favore di qualche altra procedura di ottimizzazione.

Capitolo 7

Conclusioni

All'interno del Capitolo 1 erano stati stabiliti alcuni requisiti ai quali il progetto si sarebbe dovuto conformare. Allo stato attuale, JODIE è in grado di soddisfare una buona parte di essi:

- il supporto per OWL Lite, corrispondente alla logica $\mathcal{SHF}$. In realtà il reasoner è in grado di supportare anche logiche più espressive, fino a $\mathcal{SHIF}$, tuttavia questo non è ancora sufficiente per arrivare allo stesso livello di espressività di OWL DL;
- la licenza e il linguaggio di programmazione sono quelli desiderati: JODIE è scritto in Java e verrà distribuito con licenza GPL (fatta eccezione per le classi relative ai timer, importate da Pellet, che hanno licenza MIT);
- le operazioni di ragionamento sono state eseguite utilizzando gli algoritmi di tableaux, accompagnati da diverse procedure di ottimizzazione;
- JODIE prevede un supporto per lo standard DIG, che gli permette di ricevere i dati relativi a una base di conoscenze da un'applicazione esterna come, ad esempio, un editor di ontologie.

Grazie ai test effettuati è stato possibile dimostrare empiricamente il corretto funzionamento del reasoner e del parser DIG, inoltre è stata misurata l'efficienza delle operazioni di ragionamento e delle procedure di ottimizzazione che sono state applicate ad esse. I risultati dei test hanno permesso di individuare i componenti più onerosi e le ottimizzazioni meno efficaci e, quindi, di intervenire per eliminarli o sostituirli con parti di codice più efficienti.

7.1 Valutazioni finali

Il progetto, nel suo insieme, è stato interessante e nel corso del suo sviluppo si è notata un'attenzione sempre crescente verso l'argomento: le possibilità di applicazione sono aumentate e sono stati pubblicati, anche di recente, paper dedicati agli algoritmi di tableaux applicati a logiche ancora più espressive (si consulti, ad esempio, [Horrocks and Sattler, 2005]).

Il parsing dei messaggi DIG funziona ed è discretamente veloce, grazie anche alla scelta di adoperare il parser DOM al posto dei JavaBean. Al momento attuale esso consente di inserire una base di conoscenze direttamente dall'editor di ontologie: grazie alle funzioni messe a disposizione dall'interfaccia grafica, è in seguito possibile eseguire manualmente diversi test di soddisfacibilità. Il parser, infine, è stato sviluppato in modo da essere sufficientemente semplice e modulare da poter essere incluso senza troppa fatica anche all'interno di altri progetti (come, ad esempio, il reasoner Pellet).

Per quanto riguarda il ragionamento, JODIE esegue correttamente le operazioni richieste e, anche se al suo interno non sono state ancora implementate tutte le procedure di ottimizzazione attualmente descritte in letteratura, quelle presenti hanno comunque una discreta efficacia: fra queste, una sola si è dimostrata poco utile e può essere disattivata senza grandi cali di performance, mentre tutte le altre hanno mostrato degli evidenti miglioramenti in termini di tempo di esecuzione o di memoria occupata.

Nel complesso, JODIE è un'applicazione modulare, di facile espansione e con diverse possibilità di personalizzazione: il programmatore ha la possibilità di prendere porzioni di codice da riutilizzare altrove, oppure aggiungere nuove funzioni all'interfaccia grafica o alle strategie di ragionamento; l'utente finale, invece, può modificare una grande quantità di parametri e passare, a scelta, da un reasoner "silenzioso" e performante a uno più lento ma molto più dettagliato nella descrizione delle sue operazioni.

A proposito di questo, il progetto introduce una discreta quantità di contenuti originali: nella sua versione più completa JODIE può mostrare (a scelta, all'interno della sua interfaccia grafica o da linea di comando) le comunicazioni DIG con l'editor di ontologie, la costruzione della base di conoscenze, le espansioni e le ottimizzazioni eseguite dall'algoritmo di tableaux e il risultato finale in formato grafico.

Naturalmente, il progetto può essere ancora migliorato, non tanto dal punto di vista della struttura quanto per le funzioni di ragionamento e di ottimizzazione. Al momento, infatti, esse hanno ancora un'impostazione fortemente didattica e, con qualche accorgimento non esattamente in linea con la loro descrizione formale, è sicuramente possibile ottenere dei risultati

migliori dal punto di vista delle performance. Inoltre, anche se lo scheletro dell'applicazione è ormai completo, esistono diverse funzioni accessorie che sarebbe interessante implementare, sia dal punto di vista degli strumenti (GUI, log, gestione della configurazione) sia da quello del ragionamento (algoritmi di tableaux, ottimizzazioni). Di questo, tuttavia, si parlerà più in dettaglio nella prossima sezione.

7.2 Sviluppi futuri

JODIE è passibile di numerosi miglioramenti, parte dei quali possono anche risultare piuttosto interessanti come futuri sviluppi del progetto:

- innanzi tutto, si potrebbero implementare le ottimizzazioni mancanti, così da poter eseguire in modo più efficiente non solo i normali controlli di soddisfacibilità (tramite semantic branching search, backjumping, GCI absorption e così via), ma anche l'intero processo di classificazione di un'ontologia (con un'implementazione del caching fra una query e l'altra);

- dal punto di vista della versatilità del reasoner, si potrebbe aggiungere il supporto per ABox e logiche più espressive, in modo da poter supportare restrizioni qualificate o meno (rispettivamente, $\mathcal{Q}$ e $\mathcal{N}$) e il ragionamento con individui, insieme alla possibilità di definire termini per enumerazione ($\mathcal{O}$); inoltre, potrebbe essere utile implementare l'applicazione delle regole di espansione tramite sistemi di precedenze e gestione di code, come descritto in [Tsarkov and Horrocks, 2005], in modo da poter modificare in modo semplice l'ordine di applicazione delle regole;

- infine, un'ultima integrazione al progetto potrebbe riguardare gli strumenti esterni al motore in sè: un miglioramento/potenziamento della GUI, il supporto per le *asks* o per eventuali nuovi comandi del protocollo DIG (che, è bene ricordare, appare ancora in evoluzione), una gestione della configurazione (eventualmente tramite la stessa interfaccia grafica) che stia al passo con le nuove funzioni inserite all'interno del programma.

Bibliografia

[A. Borgida, 2003] A. Borgida, M. Lenzerini, R. R. (2003). Description logics for databases. In Baader, F. and D. Calvanese, D. McGuinness, D. N. P. P.-S., editors, *The Description Logic Handbook*, pages 472–494. Cambridge University Press.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P. (2003). *The Description Logics Handbook: Theory, Implementations, and Applications*. Cambridge University Press.

[Baader et al., 1993] Baader, F., Hollunder, B., Nebel, B., Profitlich, H.-J., and Franconi, E. (1993). An empirical analysis of optimization techniques for terminological representation systems. Research Report RR-93-03, Deutsches Forschungszentrum für Künstliche Intelligenz, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH Erwin-Schrödinger Strasse Postfach 2080 67608 Kaiserslautern Germany.

[Baader et al., 2002] Baader, F., Horrocks, I., and Sattler, U. (2002). Description logics for the semantic web. *KI – Künstliche Intelligenz*, 4.

[Baader et al., 2005] Baader, F., Lutz, C., Milicic, M., Sattler, U., and Wolter, F. (2005). A description logic based approach to reasoning about web services. In *Proceedings of the WWW 2005 Workshop on Web Service Semantics (WSS2005)*, Chiba City, Japan.

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

[Bechhofer et al., 1999] Bechhofer, S., Horrocks, I., Patel-Schneider, P. F., and Tessaris, S. (1999). A proposal for a description logic interface. In *Description Logics*.

[Calvanese et al., 1998a] Calvanese, D., De Giacomo, G., and Lenzerini, M. (1998a). On the decidability of query containment under constraints.

In *Proceedings of the Seventeenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS-98)*.

[Calvanese et al., 2001a] Calvanese, D., De Giacomo, G., Lenzerini, M., and Nardi, D. (2001a). Reasoning in expressive description logics. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, pages 1581–1634. Elsevier Science Publishers.

[Calvanese et al., 2001b] Calvanese, D., Giacomo, G. D., Lenzerini, M., and Nardi, D. (2001b). Reasoning in expressive description logics. In *Handbook of Automated Reasoning*, volume 2, pages 1581–1634. MIT Press.

[Calvanese et al., 1998b] Calvanese, D., Lenzerini, M., and Nardi, D. (1998b). Description logics for conceptual data modeling. In Chomicki, J. and Saake, G., editors, *Logics for Databases and Information Systems*, pages 229–264. Kluwer Academic Publisher.

[Dickinson, 2004] Dickinson, I. (2004). Implementation experience with the dig 1.1 specification. Technical report, HP Digital Media Systems Laboratory.

[Freeman, 1995] Freeman, J. W. (1995). *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania.

[Giunchiglia and Sebastiani, 1996] Giunchiglia, F. and Sebastiani, R. (1996). Building decision procedures for modal logics from propositional decision procedure - the case study of modal k. In *Conference on Automated Deduction*, pages 583–597.

[Haarslev and Möller, 2001] Haarslev, V. and Möller, R. (2001). RACER system description. *Lecture Notes in Computer Science*, 2083:701–??

[Haarslev and Möller, 2003a] Haarslev, V. and Möller, R. (2003a). Description logic systems. In Baader, F. and D. Calvanese, D. McGuinness, D. N. P. P.-S., editors, *The Description Logic Handbook*, chapter 8, pages 282–305. Cambridge University Press.

[Haarslev and Möller, 2003b] Haarslev, V. and Möller, R. (2003b). Description logic systems with concrete domains: Applications for the semantic web. In *Proceedings of the International Workshop on Knowledge Representation meets Databases (KRDB-2003), Hamburg, Germany, September 15-1*.

[Haarslev and Möller, 2003c] Haarslev, V. and Möller, R. (2003c). The dig description logic interface. In *Proceedings of the International Workshop on Description Logics (DL-2003), Rome, Italy, September 5-7.*

[Haarslev and Möller, 2003d] Haarslev, V. and Möller, R. (2003d). Racer: A core inference engine for the semantic web. In *Proceedings of the 2nd International Workshop on Evaluation of Ontology-based Tools (EON2003)*, pages 27–36.

[Hladik and Model, 2004] Hladik, J. and Model, J. (2004). Tableau systems for SHIO and SHIQ. In Haarslev, V. and Möller, R., editors, *Proceedings of the 2004 International Workshop on Description Logics (DL 2004).* CEUR. Available from `ceur-ws.org`.

[Horrocks, 1997] Horrocks, I. (1997). *Optimising Tableaux Decision Procedures for Description Logics.* PhD thesis, University of Manchester.

[Horrocks, 1998a] Horrocks, I. (1998a). The FaCT system. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, pages 307–312.

[Horrocks, 1998b] Horrocks, I. (1998b). Using an expressive description logic: FaCT or fiction? In *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 636–647.

[Horrocks, 2002] Horrocks, I. (2002). Reasoning with expressive description logics: Theory and practice. In Voronkov, A., editor, *Proc. of the 19th Int. Conf. on Automated Deduction (CADE 2002)*, number 2392 in Lecture Notes in Artificial Intelligence, pages 1–15. Springer.

[Horrocks, 2005] Horrocks, I. (2005). Applications of description logics: State of the art and research challenges. In *Proc. of the 13th Int. Conf. on Conceptual Structures (ICCS'05).* To appear.

[Horrocks and Patel-Schneider, 2004a] Horrocks, I. and Patel-Schneider, P. (2004a). Reducing OWL entailment to description logic satisfiability. *J. of Web Semantics*, 1(4):345–357.

[Horrocks and Patel-Schneider, 1999] Horrocks, I. and Patel-Schneider, P. F. (1999). Optimising description logic subsumption. *Journal of Logic and Computation*, 9(3):267–293.

[Horrocks and Patel-Schneider, 2004b] Horrocks, I. and Patel-Schneider, P. F. (2004b). A proposal for an OWL rules language. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 723–731. ACM.

[Horrocks et al., 2003] Horrocks, I., Patel-Schneider, P. F., and van Harmelen, F. (2003). From $\mathcal{SHIQ}$ and RDF to OWL: The making of a web ontology language. volume 1, pages 7–26.

[Horrocks and Sattler, 1999] Horrocks, I. and Sattler, U. (1999). A description logic with transitive and inverse roles and role hierarchies. *Journal of Logic and Computation*, 9(3):385–410.

[Horrocks and Sattler, 2005] Horrocks, I. and Sattler, U. (2005). A tableaux decision procedure for SHOIQ. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. To appear.

[Horrocks et al., 1999a] Horrocks, I., Sattler, U., and Tobies, S. (1999a). Practical reasoning for description logics with functional restrictions, inverse and transitive roles, and role hierarchies.

[Horrocks et al., 1999b] Horrocks, I., Sattler, U., and Tobies, S. (1999b). Practical reasoning for expressive description logics. In Ganzinger, H., McAllester, D., and Voronkov, A., editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705 in Lecture Notes in Artificial Intelligence, pages 161–180. Springer-Verlag.

[Horrocks et al., 2000] Horrocks, I., Sattler, U., and Tobies, S. (2000). Practical reasoning for very expressive description logics. *Logic Journal of the IGPL*, 8(3):239–263.

[Massacci, 1999] Massacci, F. (1999). TANCS non classical system comparison. In *Proc. of TABLEAUX '99*.

[Sattler, 2003] Sattler, U. (2003). Description logics for ontologies. In *Proc. of the International Conference on Conceptual Structures (ICCS 2003)*, volume 2746 of *LNAI*. Springer Verlag.

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

[Sirin and Parsia, 2004] Sirin, E. and Parsia, B. (2004). Pellet: An owl dl reasoner. In *Description Logics*.

[Tsarkov and Horrocks, 2005] Tsarkov, D. and Horrocks, I. (2005). Ordering heuristics for description logic reasoning. In *Proc. of the 19th Int. Joint Conf. on Artificial Intelligence (IJCAI 2005)*. To appear.