Indice

Elenco delle tabelle

Elenco delle figure

INTRODUZIONE

Obiettivi della tesi

La ricerca e lo sviluppo nel campo della rappresentazione della conoscenza ha portato, recentemente, alla realizzazione di sistemi, basati su logiche descrittive, che supportano linguaggi altamente espressivi e che sono in grado di fornire potenti servizi inferenziali. Questi sistemi trovano applicazione in diverse aree: dal Web Semantico al Software Engineering, dal Electronic Business ai DataBase.

In particolare, nel campo delle applicazioni database, i servizi di ragionamento offerti da questi sistemi risultano di utile impiego nelle fasi di design, di management, ed d'integrazione delle informazioni.

Questa tesi è stata svolta presso il DB&KR Group - DataBase and Knowledge Representation Group (www.dbgroup.unimo.it), il gruppo di ricerca sulle Basi di Dati e la Rappresentazione della Conoscenza presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Modena e Reggio Emilia. L'attività di ricerca del DB&KR Group è rivolta principalmente ai Sistemi Intelligenti di Basi di Dati e all'Integrazione Intelligente delle Informazioni. Il gruppo è stato attivo nell'area dell'accoppiamento tra tecniche di intelligenza artificiale proposte nell'area delle Logiche Descrittive (DLs - Description Logics) - e Basi di Dati per lo sviluppo di Sistemi Intelligenti di Basi di Dati. In quest'area è stato sviluppato il sistema ODB-Tools [BBSV97a, BBSV97b] che esegue il controllo di consistenza degli schemi e l'ottimizzazione semantica delle interrogazioni nelle OODB - (Object Oriented DataBases).

Successivamente, l'attività di ricerca del DB&KR Group è stata estesa all'area della Integrazione Intelligente delle Informazioni (I3), ed in particolare alle architetture a due livelli wrapper/mediatore. I wrapper accedono a sorgenti di informazioni eterogenee e restituiscono queste informazioni nel linguaggio comune del mediatore. Il mediatore, sulla base di tecniche di inferenza delle Logiche Descrittive e di Clustering, è in grado di integrare e sintetizzare queste informazioni in una Vista Virtuale Globale che può essere interrogata dall'utente in maniera totalmente trasparente dalle sorgenti integrate, ottenendo una singola risposta unificata. I risultati dell'attività di ricerca hanno portato al progetto e allo sviluppo del sistema MOMIS - Mediator envirOnment for Multiple Information Sources - (www.dbgroup.unimo.it/Momis) per l'integrazione di informazioni provenienti da sorgenti eterogenee distribuite di dati strutturati (quali ad esempio database) e semi-strutturati (quali ad esempio pagine XML) [DB01, SBM01].

La motivazione della presente tesi nasce dalla seguente considerazione. Il sistema ODB-Tools è basato su una DL sviluppata ad hoc per rappresentare la semantica dei modelli di dati ad oggetti (OODMs - Object Oriented Data Models). Inoltre, la capacità espressiva di tale DL è stata opportunamente limitata in modo da ottenere algoritmi di ragionamento efficienti e rendere quindi le tecniche applicabili anche nel campo dell'ottimizzazione semantica delle interrogazioni. Di conseguenza la parte di linguaggio del sistema MOMIS sul quale effettuare attività di reasoning applicando le tecniche di inferenza tipiche delle Description Logics risulta essere limitato. L'esigenza di estendere la parte di linguaggio di MOMIS supportata da un ragionatore è soprattutto evidenziata nel contesto dell'evoluzione del progetto MOMIS nell'ambito del progetto SEWASIE (Semantic Webs and AgentS in Integrated Economies) [D2.ry].

L'obiettivo della tesi è quindi quello di estendere la parte del linguaggio del sistema MOMIS sul quale risultino applicabili le tecniche di inferenza delle Logiche Descrittive. Il punto fondamentale per svolgere tale attività è stata la constatazione che attualmente sono disponibili sistemi di rappresentazione della Conoscenza (Knowledge Base Representation Systems KBRS) basati su Description Logics, anche molto espressive, e pertanto ho ritenuto opportuno e più conveniente effettuare una traduzione dal linguaggio

del sistema MOMIS in un KBRS esistente, invece di aumentare la capacità espressiva del sistema ODB-Tools. Il KBRS scelto per svolgere tale attività è stato RACER, che supporta una DL appartenente alla famiglia delle logiche $\mathcal{AL}$ ed è in grado di gestire la conoscenza intensionale ed estensionale separando le funzionalià per entrambe le componenti. RACER fornisce inoltre un supporto per il ragionamento algebrico includendo i domini concreti, tramite i quali è in grado di trattare diverse tipologie di problemi legate ai numeri interi, ai numeri reali e alle stringhe.

La Logica Descrittiva $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ ed il sistema RACER

Il linguaggio $\mathcal{AL}$ (attributive language), introdotto per la prima volta in [SSS91] come linguaggio minimale di interesse pratico, è il capostipite di una numerosa famiglia, largamente utilizzata nei sistemi basati sulle logiche descrittive. Elemento caratterizzante di ciascun linguaggio descrittivo sono i costruttori che fornisce, infatti è grazie ad essi che, partendo dai concetti atomici e dai ruoli atomici, elementi fondamentali del linguaggio concettuale, è possibile creare descrizioni complesse. Si può ottenere una maggior espressività da un linguaggio aumentandone il numero di costruttori. Come vedremo in sezione 2.4, l'estensione del linguaggio $\mathcal{AL}$ mediante sott'insiemi di costruttori genera dei particolari linguaggi denotati come linguaggi della famiglia $\mathcal{AL}$. In questa tesi verrà considerato un particolare linguaggio $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, che è quello effettivamente supportato dal sistema RACER.

Il sistema ODB-Tools

Il sistema ODB-Tools è basato su una DL denominata *ODL* (Object Description Language) [BN94a, D.B94] che in particolare estende le DL "classiche" in quanto consente la modellazione di valori complessi e dell'ereditarietà multipla. Successivamente, *ODL* è stato esteso come segue:

- $\mathcal{ODL}_{\text{RE}}$ = $\mathcal{ODL}$ + *regole di integrità* + *path expression*;

- *regole di integrità*: regole *if then* corrispondenti ad assiomi di inclusione;
- *path expression*: sequenze di attributi che rappresentano l'ingrediente fondamentale del linguaggio d'interogazione Object-Oriented Query Languages (OQL) per navigare attraverso le gerarchie di aggregazione delle classi;

- $\mathcal{OLCD}$ = $\mathcal{ODL}$ + *unione* + *complemento*;
  - $\mathcal{OLCD}$ è un sovrainsieme di $\mathcal{ODL}_{\mathrm{RE}}$: le regole di integrità vengono espresse in $\mathcal{OLCD}$ tramite *unione* + *complemento*.

Attualmente il sistema ODB-Tools ha le seguenti caratteristiche rispetto ai linguaggi discussi in precedenza:

1. supporta pienamente il linguaggio $\mathcal{ODL}$, fornendone algoritmi di sussunzione e di soddisfacibilità corretti, completi e polinomiali;
2. supporta il linguaggio $\mathcal{ODL}_{\mathrm{RE}}$ fornendone algoritmi di sussunzione e di soddisfacibilità corretti ma non completi. Le motivazioni di tale scelta derivano dal contesto specifico nel quale le tecniche di ragionamento sono state applicate all'ottimizzazione semantica delle interrogazioni [D.B].

In questa tesi, essendo interessati ad un contesto generale, faremo riferimento al fatto che ODB-Tools supporta pienamente il linguaggio $\mathcal{ODL}$ ed indicheremo con **ODL**$_{I^3}$-ODL la parte di linguaggio di MOMIS supportata da ODB-Tools.

Adesso siamo in grado di delineare in modo più specifico l'obiettivo della tesi, ovvero la progettazione e l'implementazione di un traduttore dal linguaggio **ODL**$_{I^3}$ a quello di Racer, più precisamente dal sottoinsieme del linguaggio **ODL**$_{I^3}$ che trova corrispondenza in $\mathcal{OLCD}$, indicato con **ODL**$_{I^3}$-$\mathcal{OLCD}$, al linguaggio d'interfaccia definito da Racer. In questo modo, all'interno del sistema MOMIS, le capacità di ragionamento verranno estese dal linguaggio **ODL**$_{I^3}$-ODL, supportato da ODB-Tools, al linguaggio, chiamato **ODL**$_{I^3}$-$\mathcal{OLCD}$, che costituisce un sovrainsieme significativo di **ODL**$_{I^3}$-ODL in quanto

introduce, tra l'altro, unione e negazione. L'implementazione del traduttore dal linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ a linguaggio di Racer, che chiameremo **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** , deve considerare lo stato attuale (ottobre 2004) dello sviluppo del sistema MOMIS e quindi, in particolare, che:

1. non esiste l'implementazione di un traduttore da **ODL**$_{I^3}$-$\mathcal{OLCD}$ a $\mathcal{OLCD}$, quindi il punto di partenza per l'implementazione di **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** deve essere necessariamente il linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$;
2. il linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ non è certamente minimale;
3. le regole sintattiche del linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ sono attualmente in una versione non definitiva.

Di conseguenza, in questa tesi, si è deciso di definire una versione "canonica" del linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ sul quale effettuare il traduttore **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER**; il termine "canonico" deriva dal fatto che si considera un solo livello di innestamento e che le strutture complesse (con più livelli di innestamento) vengono definite introducendo nomi di classi e di tipi.

Naturalmente la traduzione da **ODL**$_{I^3}$-$\mathcal{OLCD}$ ad RACER di una espressione non canonica potrà essere implementata in due passi:

1. trasformando in forma canonica l'espressione;
2. applicando il traduttore implementato in questa tesi.

Dal punto di vista dell'implementazione, per tenere in considerazione la situazione appena delineata, si è deciso di sviluppare prima un prototipo "stand-alone" del traduttore da **ODL**$_{I^3}$-$\mathcal{OLCD}$ a Racer, e procedere in una fase successiva nella sua integrazione nel progetto MOMIS/SEWASIE.

Il prototipo OOSWiR

Questo lavoro ha portato alla progettazione ed allo sviluppo di **OOSWiR**, il prototipo di un tool software, realizzato in Java, che partendo dalla descrizione in **ODL**$_{I^3}$-

$\mathcal{OLCD}$/**RACER** di uno schema OO, ed sfruttando la capacità di ragionamento di RACER, è in grado di:

- validare lo schema;
- computare le relazioni di sussunzione, implicite ed esplicite, tra le classi;
- computare le relazioni di equivalenza, implicite ed esplicite, tra le classi;
- computare le relazioni di disgiunzione, implicite ed esplicite, tra le classi;
- visualizzare graficamente la tassonomia delle classi;
- fornire la descrizione delle classi, riportando le informazioni riguardanti gli attributi esplicitamente dichiarati ed implicitamente ereditati;
- fornire supporto per una semplice rappresentazione della parte intensionale relativa allo schema, permettendo l'inserimento di oggetti appartenti alle classi base e vericandone la consistenza;
- visualizzare graficamente le relazioni tra gli oggetti inseriti.

Organizzazione dei capitoli

La prima parte della tesi è stata dedicata alla presentazione delle logiche descrittive "classiche", capitoli primo e secondo, e del KBRS RACER nel terzo capitolo. Il quarto capitolo riporta la logica descrittiva $\mathcal{OLCD}$, utilizzata per rappresentare la semantica dei modelli di dati ad oggetti complessi (CODMs), mentre il quinto capitolo descrive il sistema ODB-Tools. Nel sesto capitolo viene analizzata la corrispondenza tra le logiche descrittive classiche ed la logica $\mathcal{OLCD}$, in particolare viene introdotto il modello sul quale è basata la seguente tesi. Nel settimo capitolo viene trattato il problema della rappresentazione estensionale, ed offre alcuni spunti per il suo utilizzo con RACER in relazione agli schemi OO. Infine l'ottavo capitolo presenta OOSWiR, il software sviluppato sulla base del lavoro svolto.

Capitolo 1

I sistemi di rappresentazione della conoscenza

1.1 Cenni storici

Le Logiche Descrittive (DL – dall'inglese Description Logics) costituiscono un formalismo nucleo per i Sistemi di Rappresentazione della Conoscenza (KRS –Knowledge Representation Systems), linguaggi general purpose per la rappresentazione dell'informazione e per realizzare tecniche di ragionamento. La ricerca nel campo della rappresentazione della conoscenza (KR) si è spesso indirizzata alla formalizzazione di sistemi in grado di determinare, a partire da un corpus di conoscenza codificata in modo gerarchico, conseguenze implicite non ovvie e di trovare relazioni tra le entità descritte. Storicamente si sono seguite due tipologie di approccio alla rappresentazione:

- Approccio dichiarativo (logic-based): essenzialmente basato sul calcolo dei predicati e su procedure di inferenza classiche, utilizzato nella logica terminologica o descrittiva della famiglia KL-ONE.
- Approccio procedurale (non logic-based): sviluppato a partire da considerazioni cognitive, come strutture a rete e sistemi rule-based.

I sistemi appartenenti alla prima classe, almeno inizialmente più rigorosi da un punto di vista formale, si fondano sull'uso di un linguaggio rappresentativo basato su una qualche variante del Calcolo dei Predicati, e demandano le procedure di ragionamento alle consuete verifiche di conseguenza logica. Completamente differenti, gli approcci non logici si fondano invece su strutture di rappresentazione dedicate, come le reti semantiche ed i frame, e su strutture di ragionamento ad hoc. Le reti semantiche furono introdotte intorno alla metà degli anni '60 a seguito del lavoro seminale di Quillian [Qui67], come base rapprentazionale per vocaboli di lingua inglese: l'obiettivo era quello di caratterizzare con strutture cognitive reticolari la conoscenza ed il ragionamento in sistemi artificiali. I sistemi basati sul concetto di frame [Min81] come struttura di rappresentazione fondamentale, analogamente alle reti semantiche, miravano ad estrarre conoscenza sulle dinamiche decisionali a partire dalla struttura della rete, e non dalle singole componenti. Nonostante l'enorme potenzialità insita in questi sistemi, essi non risultarono soddisfacenti da un punto di vista teorico per due buone ragioni: la vaghezza e l'inconsistenza di alcuni costrutti e, soprattutto, la mancanza di un framework teorico di livello semantico indipendente dalla particolare implementazione a partire dal quale gestire un qualche tipo di inferenza. Alla fine degli anni '70 importanti lavori [Bra77, Hay79, BBC$^+$79] gettarono le basi per una precisa formalizzazione del campo. Un primo sforzo in tal senso venne dall'intuizione che una struttura gerarchica della rete avrebbe potuto migliorare le capacità rappresentative, unitamente ad aumentare l'efficienza del ragionamento. In [Hay79] si da una precisa caratterizzazione semantica delle strutture a frame, basata sulla Logica del Primo Ordine: in particolare, attraverso l'uso di predicati unari e binari, è possibile caratterizzare gli elementi base della rappresentazione e le relazioni tra gli individui nella rete. Un passo fondamentale verso una precisa formalizzazione del campo delle Reti Semantiche è costituito dal lavoro in [Bra77, BBC$^+$79]: innalzando il significato rappresentativo da un livello semantico ad un livello epistemologico, o di conoscenza strutturata, Brachman riuscì a determinare un insieme di primitive indipendente dalla particolare applicazione o dominio. Questo lavoro successivamente costituì le fondamenta per il sistema KL-ONE [BS85]. Il collegamento definitivo tra logica classica e Reti Semantiche fu evidenziato

in [BL85]: le Reti Semantiche possono essere viste come sottoinsiemi della Logica del Primo Ordine. Inoltre, differenti formalizzazioni nei costrutti delle entità rappresentative conducono a diversi sottoinsiemi. Questo significa in sostanza che a seconda di come sia formalmente definito il linguaggio di rappresentazione della rete (la terminologia) si hanno diverse analisi di complessità, completezza e consistenza dei meccanismi di inferenza. Per questo motivo, negli anni successivi a questi risultati si parlò spesso di sistemi terminologici o logiche terminologiche, e ciò per evidenziare l'importanza dei termini del linguaggio adottato. In tempi più recenti, la ricerca si è spostata verso le logiche coinvolte nei processi decisionali, ed oggi si parla di Logiche Descrittive. A tutt'oggi, le proprietà dei meccanismi di inferenza delle Logiche Descrittive sono stati largamente studiati e migliorati, e questo soprattutto grazie all'intima connessione tra lavoro teorico, implementativo ed utilizzo reale: come dimostrano importanti contributi [BMPSR99], risulta di fondamentale importanza il passaggio da una logica formalmente codificata ad una realizzazione pratica che ha a che fare con dettagli implementativi (con considerazioni di efficienza e complessità), costruzione di algoritmi effettivi, dettagli relativi all'uso del sistema da parte di non esperti.

1.2 KL-ONE

Il sistema KL-ONE [BS85] è uno dei modelli più popolari per la rappresentazione della conoscenza in problemi di Intelligenza Artificiale e può essere considerato il capostipite dei KRS basati su DL.

I componenti principali, su cui si basa il modello, sono i concetti e i ruoli. I concetti (classi) sono predicati unari; a ciascuno di essi corrisponde un insieme potenziale di individui con proprietà comuni, descritte dalla struttura interna, potenzialmente complessa, del concetto stesso. I ruoli, elementi primari dei concetti, corrispondono a predicati binari e rappresentano le relazioni potenziali tra gli individui descritti dal concetto e altri individui.

Risultato di un lungo lavoro teorico sul significato e sulla struttura delle Reti Semantiche, KL-ONE include tutte le caratteristiche chiave dei sistemi moderni:

- Uso distinto di conoscenza intensionale ed estensionale per la modellazione del dominio.
- Uso di relazioni (”ruoli“) per determinare le dipendenze tra i concetti, unitamente alle restrizioni imponibili su tali relazioni.
- Nozione di sussunzione e classificazione come metodologia di inferenza fondamentale.

Dal modello KL-ONE sono derivati una serie di formalismi di rappresentazione della conoscenza, i linguaggi terminologici, che si concentrano sulla gestione delle entità intensionali, i concetti. A seconda del linguaggio impiegato per la formalizzazione di una base di conoscenza strutturata, è possibile avere sistemi molto diversi riguardo a potenza descrittiva, efficienza della computazione e correttezza della deduzione. I primi lavori basati sul corpus iniziale di KL-ONE si possono quindi dividere in base al modo di porsi nei confronti del rapporto tra espressività del linguaggio e complessità del calcolo. Alcuni linguaggi, tra cui CLASSIC [BBMR89], sono caratterizzati da una espressività limitata nell’uso dei costruttori, ma garantiscono completezza e correttezza dell’inferenza, restringendo l’insieme dei costruttori utilizzabili, e sono caratterizzati da un’elevata efficienza di computazione, possibilmente in tempo polinomiale. Questo approccio è preferibile in contesti dove il linguaggio deve essere utilizzato in applicazioni reali e funzionanti, nei quali non è pensabile che il sistema non sia in grado di ricavare una conseguenza logica se questa esiste. Altri, tra cui LOOM [MB87, Mac88] sono molto più espressivi, permettendo descrizioni di concetti e relazioni molto complesse, tuttavia non sono completi, e cioè possono non trovare delle conseguenze logiche anche se queste sussistono. Esiste attualmente una ricerca molto fiorente che mira alla realizzazione di sistemi altamente espressivi e completi, sacrificando però l’efficienza di computazione, permettendo algoritmi di complessità temporale esponenziale per la gestione dei costruttori critici, ed affidandosi a tecniche di ottimizzazione per ridurre tali tempi.

1.3 Rappresentazione della conoscenza con le Logiche Descrittive

Un KRS è caratterizzato da due aspetti principali:

- La Base di Conoscenza (KB – dall'inglese Knowledge Base).
- L'ambiente di sviluppo.

Una precisa caratterizzazione di una KB implica la definizione del linguaggio di rappresentazione, del tipo di conoscenza che si vuole rappresentare e delle funzionalità gestibili dall'utente (umano o artificiale), e cioè quali sono le inferenze gestibili dalla KB, quali le domande che l'utente può porre e così via. La necessità di questa caratterizzazione si traduce in quello che viene comunemente detto approccio funzionale [BS85]: la specifica di una interfaccia che indichi formalmente le operazioni di costruzione e di query della KB, unitamente alle procedure di inferenza esportate. La definizione di un ambiente di sviluppo che incapsuli la KB risulta invece importante per come le funzionalità possono essere gestite.

1.4 Caratterizzazione di una Base di Conoscenza

Una Base di Conoscenza può essere suddivisa in due parti: la prima è costituita dalla conoscenza definita ed immutabile sul dominio rappresentato, a livello concettuale e strutturale, chiamata conoscenza intensionale; la seconda è invece costituita dalla conoscenza relativa al problema attuale, specifico, e viene detta estensionale. Dal punto di vista di una KB basata su DL, questo porta ad una suddivisione in:

- TBox, dove T sta per terminologico o tassonomico, che è la parte intensionale di una KB.
- ABox, dove A sta per asserzionale, e si riferisce alla conoscenza estensionale.

Una TBox viene costruita a partire da una definizione esaustiva di concetti e relazioni tra concetti e viene generalmente considerata invariabile durante la vita di un KRS; comunque in letteratura è stato studiato il problema dell'evoluzione temporale dello schema [AFM03]. Al contrario, una ABox, modellando conoscenza contingente sul dominio in esame, si riferisce ad individui specifici ed attuali. É opportuno notare che, spesso, la suddivisione concettuale tra TBox e ABox ha anche una controparte pratica: le procedure di inferenza che caratterizzano i due diversi tipi di conoscenza sono profondamente diverse, distinte e con differente complessità computazionale o performance. D'altro canto, dal punto di vista dell'utente del sistema, questa differenza è mascherata dai costrutti del linguaggio, simili per la definizione dei due tipi di conoscenza.

1.4.1 Tbox

La componente intensionale di una KB basata su DL, la terminologia, è formata da una serie di definizioni di concetti a partire da descrizioni, a loro volta costruite a partire da altri concetti e costruttori. Naturalmente esistono dei vincoli sulle definizioni. In particolare:

- Un concetto è definito in modo univoco una e una sola volta.
- Nella parte sinistra della dichiarazione può essere presente solo un nome di un concetto, inteso come entità atomica.
- Un concetto non può essere definito in funzione di se stesso.
- Un concetto non può essere definito nei termini di un secondo concetto che sussume, perché questo non può ancora essere stato definito nella KB.

Gli ultimi due vincoli possono essere riassunti imponendo che, in generale, non siano permesse definizioni cicliche o autoreferenti. Non solo questi assunti sono vitali per mantenere l'integrità semantica della KB, ma sono anche fondamentali per l'esistenza stessa delle procedure di inferenza e sussunzione. Tali procedure, infatti, si basano sull'espansione delle descrizioni dei termini della tassonomia in formule uniche, costituite dall'and

di diversi costruttori (processo di normalizzazione della KB). Laddove, in una descrizione, vi sia un concetto non primitivo, ma dotato a sua volta di descrizione strutturata, questo viene sostituito dalla sua descrizione, espandendo in tal modo la descrizione originaria. Nonostante questa procedura possa dare luogo a descrizioni esponenziali [Neb90b], nella pratica non si hanno descrizioni tali da intaccare la complessità del ragionamento: è possibile gestire la complessità inferenziale considerando tutte le formule pienamente espanse. Costruendo la KB in questo modo, la terminologia viene ad assumere un aspetto gerarchico, implicitamente strutturato come un sistema di classificazione. Nella fattispecie, ogni concetto si colloca esattamente tra i concetti che lo sussumono più direttamente e quelli che esso stesso sussume. Mano a mano che si procede dai concetti superiori a quelli inferiori, le definizioni si fanno via via più complesse e la rete più intricata. In anni recenti ci sono state alcune proposte finalizzate all'ampliamento della capacità rappresentativa delle TBox, cercando, al contempo, di mantenere gli algoritmi di sussunzione trattabili. Un'importante estensione concettuale è quella di definire assiomi di sussunzione, ovvero specifiche formule logiche che, arbitrariamente, definiscono relazioni di sussunzione tra descrizioni, slegate dalla tassonomia definita con la classificazione. In questo modo è possibile definire relazioni IS-A trasversali. Queste relazioni sono semanticamente valide considerando l'inferenza come un processo di implicazione logica: in tal modo qualunque procedura che agisca sulla KB modificandola coerentemente può essere equiparata ad un opportuno assioma logico di implicazione, e quindi essere ridotto in formule.

1.4.2 Abox

La ABox contiene asserzioni che specificano gli individui attuali nel dominio di interesse. Le asserzioni possono essere di due tipi fondamentali: asserzioni concettuali e asserzioni di ruolo. Generalmente la specifica delle asserzioni nella KB avviene in modo distinto: prima si dichiarano gli individui del dominio, e poi si indicano tutte le rispettive relazioni. Le asserzioni concettuali definiscono l'appartenenza di un individuo ad una categoria (o concetto) precedentemente specificato nella TBox. Le procedure di inferenza tipiche

di una ABox coinvolgono necessariamente anche la TBox. Infatti è possibile vedere se un individuo è un'istanza di un concetto, quali concetti lo sussumono, quali sono le relazioni con altri individui, quali sono gli individui istanze di un concetto e così via.

1.5 Applicazioni basate su un KRS

I KRS basati su DL sono alla base di numerosi sistemi informatici in diversi campi, dalla medicina all'ingegneria del software, dai sistemi informativi alle librerie digitali, dai sistemi di configurazione a sistemi di pianificazione. Nel campo dell'ingegneria del software, ad esempio, si sono realizzati alcuni sistemi informativi per la documentazione del software: il sistema LaSSIE [DBSB90] permette ad un programmatore di aggiornare interattivamente una KB mano a mano che inserisce nuove funzionalità nel proprio codice. Il sistema può successivamente rispondere a query riguardanti quali funzionalità sono implementate e dove. Il campo delle configurazioni riguarda la gestione di sistemi strutturati complessi, dove le diverse parti devono interagire vicendevolmente: il problema generalmente si può ricondurre a quello di trovare un insieme di parti costituenti da interconnettere, in modo da costruire un sistema con determinate specifiche. Tramite una DL risulta piuttosto facile modellare eventuali incompatibilità tra le parti, cosa molto importante nel momento in cui la complessità del sistema cresce in modo smisurato. Le grandi capacità di classificazione e recupero di informazione delle DL hanno fatto si che queste fossero impiegate con successo in sistemi digitali di archiviazione, dove è un requisito fondamentale la capacità di rappresentare link con altre fonti di informazione. Queste metodologie sono state diffusamente estese fino alla visione del World Wide Web (WWW) come una immensa Rete Semantica [KLSS95]: l'idea è quella di aggiungere informazione alle pagine Web in modo tale che queste possano essere scandagliate semanticamente, e non solo sintatticamente, come viene fatto oggi. Le relazioni tra le logiche descrittive e i linguaggi di markup, come XML, sono state diffusamente indagate e formalizzate [CGL99]. Nel campo della pianificazione sono stati portati avanti alcuni tentativi per la rappresentazione di piani ed azioni: un DL-KRS per il ragionamento su piani ed azioni è CLASP

[YNM91], dove l'informazione tassonomica è utilizzata verificando se gli individui che rappresentano piani appartengono o meno a determinate categorie note a priori.

Capitolo 2

La logica descrittiva $\mathcal{AL}$

2.1 La descrizione intensionale: la TBox

Il linguaggio $\mathcal{AL}$ (attributive language), introdotto per la prima volta in [SSS91] come linguaggio minimale di interesse pratico, è il capostipite di una numerosa famiglia, largamente utilizzata nei sistemi basati sulle logiche descrittive.
Elemento caratterizzante di ciascun linguaggio descrittivo sono i costruttori che fornisce, infatti è grazie ad essi che, partendo dai concetti atomici e dai ruoli atomici, elementi fondamentali del linguaggio concettuale, è possibile creare descrizioni complesse. Nella notazione astratta, esposta nel seguito, vengono utilizzate le lettere A e B per esprimere i concetti atomici, R per i ruoli atomici e infine C e D per le descrizioni dei concetti.

La descrizione dei concetti il $\mathcal{AL}$ avviene secondo la seguente regola sintattica astratta:

C, D	$\rightarrow$	$\top$	concetto universale
		$\bot$	concetto vuoto
		A	concetto atomico
		$\neg A$	negazione atomica
		$C \sqcap D$	intersezione
		$\forall R.C$	qualificatore universale
		$\exists R.\top$	qualificatore esistenziale

Come si può notare, la negazione, in $\mathcal{AL}$, è applicabile solamente ai concetti atomici e, per quanto riguarda i ruoli, il qualificatore esistenziale ha sempre, e solo, come oggetto l'insieme universale.

Per dare un'esempio di quello che può essere espresso il $\mathcal{AL}$, supponiamo che *Person* e *Female* siano due concetti atomici. Allora:

$$\text{Person} \sqcap \text{Female},$$

ed

$$\text{Person} \sqcap \neg\text{Female},$$

rappresentano due descrizioni di concetto, intuitivamente, le persone di sesso femminile e le persone che non sono di sesso femminile. Supponiamo inoltre che *hasChild* sia un ruolo atomico; possiamo formare il concetto:

$$\text{Person} \sqcap \forall\text{hasChild.Female},$$

per indicare l'insieme delle persone che, se hanno dei figli, questi sono tutti femmine. Altresì possiamo descrivere le persone che non hanno dei figli, semplicemente tramite la descrizione

$$\text{Person} \sqcap \text{hasChild.}\bot.$$

Per definire una semantica formale dei concetti in $\mathcal{AL}$, dobbiamo considerare le interpretazioni $\mathcal{I}$=($\Delta_I, \cdot^{\mathcal{I}}$), che consistono di un insieme non vuoto Δ_I, detto dominio delle interpretazioni, e di una funzione d'interpretazione, la quale assegna a ciascun concetto atomico A un insieme $\mathrm{A}^{\mathcal{I}} \subseteq \Delta_I$ e a ciascun ruolo atomico R una relazione binaria $\mathrm{R}^{\mathcal{I}} \subseteq \Delta_I \times \Delta_I$. Induttivamente, estendendo il discorso alla descrizione dei concetti, possiamo dare una definizione formale della funzione di interpretazione.

Definizione 1 (Funzione d'interpretazione per la logica $\mathcal{AL}$) *Una funzione d'interpretazione $\cdot^{\mathcal{I}}$ su Δ_I per la logica $\mathcal{AL}$, è una funzione tale che:*

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

Tramite la funzione di interpretazione è possibile verificare, per esempio, che la descrizione di concetto:

$$\forall \text{hasChild.Female} \sqcap \forall \text{hasChild.Student},$$

rappresenta lo stesso insieme di individui rappresentato dalla descrizione di concetto

$$\forall \text{ hasChild.(Female} \sqcap \text{ Student)}.$$

Definizione 2 (Concetti Equivalenti) *Date due descrizioni di concetto, C e D, si dice che queste sono tra loro equivalenti, indicato $C \equiv D$, se e solo se $C^{\mathcal{I}} = D^{\mathcal{I}}$ per ogni interpretazione $\mathcal{I}$.*

Uno dei punti cardine delle logiche descrittive è rappresentato dal significato di "soddisfacibilità", del quale viene data una prima definizione.

Definizione 3 (Soddisfacibilità di concetto) *Sia $\mathcal{I}$=(Δ_I, $\cdot^{\mathcal{I}}$) un'interpretazione. Dato un concetto C∈ $\mathbf{C}$, si dice che una interpretazione $\mathcal{I}$=(Δ_I, $\cdot^{\mathcal{I}}$) è modello del concetto C, o soddisfa C, se e solo se $C^{\mathcal{I}} \neq \emptyset$.*

2.1.1 Gli assiomi terminologici

Nel paragrafo precedente sono state mostrate le regole per creare delle descrizioni complesse di concetti. Ora vengono introdotti gli assiomi terminologici, che dichiarano le relazioni esistenti tra concetti e tra ruoli. Nel caso più generale gli assiomi terminologici hanno una delle forme:

$$C \sqsubseteq D \quad (R \sqsubseteq S),$$

oppure:

$$C \equiv D \quad (R \equiv S),$$

dove C ed D sono concetti e R ed S ruoli. Gli assiomi espressi nella prima forma sono detti assiomi di inclusione, mentre quelli nella seconda forma vengono detti assiomi di equivalenza.

Definizione 4 (Assioma di inclusione tra concetti) *Se C,D $\in$ $\mathbf{C}$, allora $C \sqsubseteq D$ rappresenta un assioma di inclusione tra concetti (GCI: general concept inclusion).*

Definizione 5 (Assioma di equivalenza tra concetti) *Se C,D $\in$ $\mathbf{C}$, allora $C \equiv D$ è detto assioma di equivalenza tra concetti, o semplicemente equivalenza tra concetti.*

Definizione 6 (Assioma di inclusione tra ruoli) *Se R,S $\in$ $\mathbf{R}$, allora $R \sqsubseteq S$ è detto assioma di inclusione tra ruolo, o semplicemente inclusione tra ruoli.*

Definizione 7 (Assioma di equivalenza tra ruoli) *Se R,S $\in$ $\mathbf{R}$, allora $R \equiv S$ è detto assioma di equivalenza tra ruoli, o semplicemente equivalenza tra ruoli.*

Data un'interpretazione $\mathcal{I}$ diciamo che essa soddisfa un'assioma di inclusione, $C \sqsubseteq D$, se e solo se $C^{\mathcal{I}} \sqsubseteq D^{\mathcal{I}}$, mentre soddisfa un assioma di equivalenza, $C \equiv D$, se e solo se $C^{\mathcal{I}} \equiv D^{\mathcal{I}}$; considerando un insieme di assiomi è possibile dare una defizione di soddisfacibilità terminologica:

Definizione 8 (Soddisfacibilità terminologica) *Dato un insieme finito di assiomi terminologici, $\mathcal{T}$, si dice che un'interpretazione $\mathcal{I}$ è un modello di $\mathcal{T}$, se e solo se soddisfa tutti gli assiomi definiti in $\mathcal{T}$.*

Definizione 9 (Terminologia(TBox)) *Un insieme finito di assiomi terminologici $\mathcal{T}$ è detto terminologia (TBox).*

Definizione 10 (Assioma di definizione) *Un assioma di equivalenza in cui compaia a sinistra un concetto atomico è detto definizione.*

Le definizioni vengono utilizzate per identificare, con dei nomi simbolici, concetti complessi. Per esempio, l'assioma:

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

è una definizione perchè associa la descrizione, a destra, al nome $Mother$.

Definizione 11 (Assioma di specializzazione) *Un assioma di inclusione nel quale, a sinistra, compaia un concetto atomico è detto specializzazione.*

Gli assiomi di specializzazione, in teoria, non aggiungono espressività al linguaggio, mentre, nella pratica, offrono il vantaggio di poter introdurre, in una terminologia, dei concetti non completamente definiti.

2.1.2 Semantica descrittiva

Supponiamo che $\mathcal{T}$ sia una terminologia formata da un insieme finito di definizioni, tale che ciascun nome simbolico non sia definito più di una volta. Possiamo dividere i concetti

$$
\begin{aligned}
Woman &\equiv Person \sqcap Female \\
Man &\equiv Person \sqcap \neg Woman \\
Mother &\equiv Woman \sqcap \exists hasChild.Person \\
Father &\equiv Man \sqcap \exists hasChild.Person \\
MotherWithoutDaughter &\equiv Mother \sqcap \forall hasChild.\neg Woman \\
Wife &\equiv Woman \sqcap \exists hasHusband.Man
\end{aligned}
$$

Tabella 2.1: Esempio di terminologia che esprime alcune “relazioni famigliari”.

atomici che compaiono in $\mathcal{T}$ in due insiemi: l’insieme dei nomi simbolici, che compaiono solo a destra negli assiomi di definizione, e l’insieme dei nomi base, che compaiono solo nella parte sinistra degli assiomi. I nomi simbolici vengono detti concetti definiti, mentre i nomi base sono detti concetti primitivi. Ci aspettiamo che la $\mathcal{T}$ definisca i nomi simbolici nei termini dei nomi base. Un’interpretazione di base per $\mathcal{T}$ è un’interpretazione che interpreta solo i nomi base. Sia $\mathcal{J}$ un’interpretazione di base. Un’interpretazione $\mathcal{I}$ che interpreta anche i nomi simbolici è un’estensione di $\mathcal{J}$ se ha lo stesso dominio e se la sua interpretazione dei nomi base è in accordo con $\mathcal{J}$. Diciamo che $\mathcal{T}$ è determinabile se ciascuna interpretazione di base ha esattamente un’interpretazione estensionale che è un modello per $\mathcal{T}$. In altre parole, se noi conosciamo la rappresentazione dei nomi base e $\mathcal{T}$ è determinabile, allora il significato dei nomi simbolici è completamente determinato. Ovviamente se una terminologia è determinabile allora qualsiasi altra terminologia equivalente è determinabile.

Il problema per il quale risulta importante sapere se una terminologia è determinabile è collegato alle definizioni cicliche. Per esempio:

$$\text{Human} = \text{Animal} \sqcap \text{hasParent.Human}$$

è una definizione ciclica.

In generale definiamo i cicli in una terminologia nel modo seguente.

Definizione 12 (Terminologia ciclica) *Siano A,B due concetti atomici che compaiono in $\mathcal{T}$. Diciamo che A utilizza direttamente B in $\mathcal{T}$ se B appare alla destra nell'assioma di definizione di A. Allora $\mathcal{T}$ contiene un ciclo se e solo se esiste un concetto atomico in $\mathcal{T}$ che usa se stesso; in caso contrario $\mathcal{T}$ è detta aciclica.*

Per le terminologie cicliche non esiste un'unica interpretazione estensionale. Nell'esempio precedente, *Human* è un nome simbolico, *Animal* e *hasParent* sono simboli base. Per un'interpretazione, in cui ciascun *Animal* è in relazione con il suo progenitore, sono possibili molte estensioni che interpretano *Human* in modo tale che l'assioma sia soddisfatto. *Human* può essere interpretato come l'intero insieme di animali, una sola specie, o come l'insieme di animali che ha la proprietà di contenere, per ciascuno di essi, tutti i progenitori.

$$
\begin{aligned}
Woman &\equiv Person \sqcap Female \\
Man &\equiv Person \sqcap \neg(Person \sqcap Female) \\
Mother &\equiv (Person \sqcap Female) \sqcap \exists hasChild.Person \\
Father &\equiv (Person \sqcap \neg(Person \sqcap Female)) \\
&\quad \sqcap \exists hasChild.Person \\
MotherWithoutDaughter &\equiv ((Person \sqcap Female) \sqcap \exists hasChild.Person) \\
&\quad \sqcap \forall hasChild.\neg(Person \sqcap Female) \\
Wife &\equiv (Person \sqcap Female) \sqcap \exists hasHusband. \\
&\quad (Person \sqcap \neg(Person \sqcap Female))
\end{aligned}
$$

Tabella 2.2: Espansione della terminologia "relazioni famigliari".

Nel caso di terminologia aciclica non esiste il problema dell'interpretazione, infatti essa è sempre determinabile. Questo si può spiegare col fatto che è possibile, come mostrato

nell'esempio 2.2, espandere mediante un processo iterativo le definizioni in $\mathcal{T}$, sostituendo le occorrenze dei nomi simbolici, che compaiono a destra, con la loro descrizione, fino a quando non sono stati eliminati tutti i cicli dalla terminologia.

Proposizione 1 (Terminologie acicliche) *Sia $\mathcal{T}$ una terminologia aciclica e sia $\mathcal{T}$' la sua espansione. Allora:*

- *$\mathcal{T}$ e $\mathcal{T}$' hanno gli stessi insiemi di nomi simboloci e simboli base;*
- *$\mathcal{T}$ e $\mathcal{T}$' sono equivalenti;*
- *entrambe $\mathcal{T}$ e $\mathcal{T}$' sono determinabili.*

Esistono terminologie cicliche che sono determinabili. Consideriamo l'assioma:

$$\mathrm{A} \equiv \forall\, \mathrm{R.B} \sqcup \exists\, \mathrm{R.(A} \sqcap \neg \mathrm{A}),$$

in cui compare un ciclo. Ma $\exists$ *R.(A* $\sqcap$ $\neg$*A)* rappresenta l'insieme vuoto e quindi l'assioma precedente si può riscrivere come:

$$\mathrm{A} \equiv \forall\, \mathrm{R.B}.$$

In ragione di ciò, possiamo enunciare il seguente teorema, riformulazione del teorema di determinabilità di Beth per le logiche propositive $\mathcal{K}_n$, che, come mostrato da Schild [Sch91] rappresentano una variante notazionale delle logiche $\mathcal{ALC}$.

Teorema 1 (Terminologie $\mathcal{ALC}$) *Ciascuna terminologia determinabile, espressa mediante la logica descrittiva $\mathcal{ALC}$, è equivalente ad una terminologia aciclica.*

In definitiva possiamo dire che, rispetto alla semantica che abbiamo spiegato fin'ora, che è essenzialmente la semantica delle logiche del primo ordine, le terminologie hanno un "impatto" determinabile solo se sono acicliche. Nebel, [Neb91], attribuisce a queste semantiche il nome di semantiche descrittive, per differenziarle dalla semantiche di punto fisso introdotte nel prossimo paragrafo.

2.1.3 Semantica di punto fisso

Le semantiche di punto fisso sono motivate dal fatto che, in certe situazioni, i cicli sono intuitivamente significativi e che questa intuizione può essere catturata dalla semantica di massimo, oppure di minimo, punto fisso.

Supponiamo, ad esempio, di voler specificare il concetto di "uomo che ha solo discendenti maschi", e attribuiamogli il nome simbolico di *Momo* (man only male offspring). Possiamo scrivere, in modo ricorsivo:

$$\text{Momo} \equiv \text{Man} \sqcap \forall\ \text{hasChild.Momo}.$$

La definizione è ciclica, ma il suo significato è chiaro per tutti e, concettualmente, la sua interpretazione rappresenta una dinastia di uomini. Intuitivamente noi attribuiamo a *Momo* il significato corretto perchè interpretiamo questa definizione sotto l'ottica della semantica di massimo punto fisso.

Prendiamo un'altro esempio, [Neb91]. Immaginiamo di avere un'insieme di oggetti *Tree*, e di relazioni binarie, *hasBranch*, che legano gli oggetti *Tree* con i propri sotto alberi. Allora la definizione di "*BinaryTree*", che indica quegli "alberi che hanno al massimo due rami che sono a loro volta alberi binari", può essere espressa come:

$$\text{BinaryTree} \equiv \text{Tree} \sqcap \leq 2\ \text{hasBranch} \sqcap \forall\ \text{hasBranch.BinaryTree},$$

dove, con:

$$\leq 2\ \text{hasBranch},$$

s'indica una restrizione di cardinalità, 2.5, per il ruolo *hasBranch*; tale restrizione impone che il ruolo abbia, nella descrizione *BinaryTree*, al più due riempitivi. In questo caso è la semantica di minimo punto fisso ad attribuire il giusto significato alla definizione.

Ricordando che stiamo considerando delle terminologie $\mathcal{T}$ composte di sole definizioni, in cui ciascun nome simbolico A ricorre solo una volta alla sinistra di un assioma, possiamo vedere $\mathcal{T}$ come una funzione che mappa ciascun nome simbolico A in una

descrizione di concetto $\mathcal{T}$(A)=C. Con questa notazione un'interpretazione $\mathcal{I}$ è un modello per $\mathcal{T}$ se e solo se $A^{\mathcal{I}}=(\mathcal{T}(A))^{\mathcal{I}}$. Questa caratterizzazione contiene l'essenza della semantica di punto fisso.

Definizione 13 (Punto fisso) *Sia $\mathcal{T}$ una terminologia e $\mathcal{J}$ una base d'interpretazione fissata per $\mathcal{T}$. Con Ext$_J$ indichiamo l'insieme di tutte le estensioni di $\mathcal{J}$. Sia $\mathcal{T}_{\mathcal{J}}$:Ext$_{\mathcal{J}}$ $\rightarrow$ Ext$_{\mathcal{J}}$ una funzione che mappa ciascuna estensione $\mathcal{I}$ nell'estensione $\mathcal{T}_{\mathcal{J}}(\mathcal{I})$ definita da $A^{\mathcal{T}_{\mathcal{J}}(\mathcal{I})} = (\mathcal{T}(A))^{\mathcal{I}}$ per ciascun nome simbolico A. Diciamo che $\mathcal{I}$ è un punto fisso di $\mathcal{T}_{\mathcal{J}}$ se e solo se $\mathcal{I}$=$\mathcal{T}_{\mathcal{J}}(\mathcal{I})$ per ciascun nome simbolico.*

Per fissare le idee possiamo riferirci all'esempio *Momo* e considerare gli insiemi:

$$\begin{aligned} \Delta^J &= \{Charles_1, Charles_2, \ldots\} \cup \{James_1, \ldots, James_m\}, \\ Uomo^{\mathcal{J}} &= \Delta^J, \\ hasChild^{\mathcal{J}} &= \{(Charles_i, Charles_{i+1}) \mid i \geq 1\} \cup \\ &\quad \{(James_i, James_{i+1}) \mid 1 \leq i \leq m\}, \end{aligned}$$

definiti in modo tale che la dinastia dei *Charles* non sia estinta fintantochè è ancora in vita l'ultimo dei *James*. Vogliamo individuare il punto fisso di ${\mathcal{T}_{\mathcal{J}}}^{Momo}$. Possiamo notare che qualsiasi individuo senza figli appartiene all'interpretazione di

$$\forall \text{ hasChild.Momo.}$$

Perciò, se $\mathcal{I}$ è un punto fisso che estende $\mathcal{J}$, allora *James*$_m$ appartiene a ($\forall$ *hasChild.Momo*)$^{\mathcal{I}}$ e quindi a *Momo*$^{\mathcal{I}}$. Si può concludere allora che tutti i *James* sono *Momo*. Sia $\mathcal{I}_1$ l'estensione di $\mathcal{J}$ che include la dinastia dei *James*. In base alle considerazioni precedenti anche alcuni *Charles* sono dei *Momo*, e quindi lo devono essere anche i tutti loro antenati e i loro discendenti. È facile verificare che un'estensione $\mathcal{I}_2$ che interpreti come *Momo* l'intero dominio è anch'essa un punto fisso.

Per dare una precisa caratterizzazione alla semantica di punto fisso, massima o minima, è necessario definire un ordine parziale, $\preceq$, dell'estensioni su $\mathcal{J}$.

Definizione 14 (Ordine parziale) *Si può dire che* $\mathcal{I} \preceq \mathcal{I}'$ *se* $A^{\mathcal{I}} \subseteq A^{\mathcal{I}'}$ *per ciascun nome simbolico in* $\mathcal{T}$.

Nell'esempio in esame, *Momo* è l'unico nome simbolico e dato che $Momo^{\mathcal{I}_1} \subseteq Momo^{\mathcal{I}_2}$, si ha che $\mathcal{I}_1 \preceq \mathcal{I}_2$.

Definizione 15 (Minimo punto fisso) *Un punto fisso di* $\mathcal{I}$ *di* $\mathcal{T}_{\mathcal{J}}$ *è detto minimo punto fisso se* $\mathcal{I} \preceq \mathcal{I}'$ *per ogni* $\mathcal{I}'$.

Definizione 16 (Massimo punto fisso) *Un punto fisso di* $\mathcal{I}$ *di* $\mathcal{T}_{\mathcal{J}}$ *è detto massimo punto fisso se* $\mathcal{I}' \preceq \mathcal{I}$ *per ogni* $\mathcal{I}'$.

Con riferimento all'esempio, $\mathcal{I}_1$ è il minimo punto fisso mentre $\mathcal{I}_2$ il massimo punto fisso.

2.2 La descrizione del mondo: l'ABox

La seconda componente di una base di conoscenza, in aggiunta alla terminologia, è l'A-Box. Tramite l'ABox è possibile introdurre gli individui appartenenti al dominio di applicazione, dare loro un nome e asserirne le proprietà.

Indicando con le lettere a, b, c tre nomi di individui, e considerando un concetto C e un ruolo R, possiamo fare due tipi di asserzioni nell'ABox:

C(a), R(b,c)

Un'asserzione del primo tipo è detta asserzione di concetto e dichiara che a appartiene all'interpretazione di C, mentre un'asserzione della seconda specie, chiamate asserzione di ruolo, dichiara che c è un riempitivo del ruolo R per b.

Definizione 17 (ABox) *Un insieme finito di assiomi asserzionali* $\mathcal{A}$ *è detto ABox.*

Si potrebbe quindi pensare di vedere l'ABox come l'insieme delle istanze di un database relazionale, in cui vengono utilizzate solo le associazioni unarie e binarie. Questo in realtà

non è corretto, infatti al contrario della semantica di mondo chiuso dei database classici, la semantica dell'ABox adotta l'approccio di mondo aperto e, oltrettutto, la TBox impone delle relazioni tra i concetti e i ruoli, che non hanno una controparte nella semantica dei database.

2.2.1 Assunzione di univocità dei nomi (UNA)

Possiamo dare una semantica all'ABox estendendo l'interpretazione $\mathcal{I} = (\Delta_I, \cdot^{\mathcal{I}})$ ai nomi degli individui, in modo che ciascun a appartente ad essi sia mappato in un elemento del dominio $a^{\mathcal{I}} \in \Delta_I$, con l'assunzione che a nomi distinti corrispondano oggetti distinti del dominio:

Definizione 18 (Assunzione di univocità dei nomi) *Sia una funzione di interpretazione $\mathcal{I}$ e sia* **O** *un insieme numerabile di nomi d'individui. Si dice che $\mathcal{I}$ verifica l'assunto di univocità dei nomi se e solo se per qualsiasi coppia $(a, b) \in$* **O** *tale che $a \neq b$ si ha $a^{\mathcal{I}} \neq b^{\mathcal{I}}$.*

Si dice che un'interpretazione $\mathcal{I}$ soddisfa un'asserzione di concetto C(a) se $a^{\mathcal{I}} \in C^{\mathcal{I}}$, e che $\mathcal{I}$ soddisfa un'asserzione di ruolo R(a,b) se $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$. Se un'interpretazione $\mathcal{I}$ soddisfa tutte le asserzioni di concetto C(a) e di ruolo R(a,b) allora soddisfa l'ABox $\mathcal{A}$ e si dice che $\mathcal{I}$ è modello di $\mathcal{A}$. Infine, se $\mathcal{I}$ è un modello di $\mathcal{A}$ rispetto a $\mathcal{T}$ allora $\mathcal{I}$ è anche modello di $\mathcal{T}$.

Definizione 19 (Soddisfacibilità asserzionale) *Dati un ABox $\mathcal{A}$ e un TBox $\mathcal{T}$ si dice che un'interpretazione $\mathcal{I}$ è un modello di $\mathcal{A}$ rispetto a $\mathcal{T}$, se e solo se $\mathcal{I}$ è un modello per $\mathcal{T}$ e soddisfa tutte le asserzione definite in $\mathcal{A}$.*

Un modello di $\mathcal{A}$ e $\mathcal{T}$ è un'astrazione del mondo reale, dove i concetti sono interpretati come sottoinsiemi del dominio, come richiesto dalla TBox, e dove l'appartenenza di un individuo ad un concetto e le sue relazioni con gli altri individui, tramite i ruoli, rispecchiano le asserzioni dell'ABox.

2.3 Inferenza

I sistemi di rappresentazione della conoscenza basati sulle logiche descrittive sono capaci di diverse forme di ragionamento. Una KB, intesa come l'insieme di TBox e ABox, ha una semantica che la rende equivalente ad un insieme finito di assiomi in logica dei predicati del primo ordine. Quindi, come qualsiasi altro insieme di assiomi, contiene una conoscenza implicita che può essere esplicitata tramite il meccanismo d'inferenza; uno degli aspetti che distingue nettamente una base di conoscenze da una base di dati è la possibilità di condurre ragionamenti in modo automatico. I diversi tipi di ragionamento, eseguibili da un sistema di DL, sono detti inferenze logiche, e riguardano, o possono riguardare, sia la TBox che l'ABox.

2.3.1 La sussunzione e il ragionamento sui concetti

Quando si costruisce una terminologia, si definiscono nuovi concetti nei termini di quelli definiti precedentemente. Durante questo processo è importante che i nuovi concetti abbiano senso e non siano contradditori. Dal punto di vista logico, un concetto ha senso se esiste un'interpretazione, che soddisfi gli assiomi in $\mathcal{T}$, tale che il concetto denoti un insieme non vuoto per quella interpretazione. Un concetto con questa proprietà è detto soddisfacibile rispetto a $\mathcal{T}$ e insoddisfacibile altrimenti. Il controllo di soddisfacibilità di un concetto è uno dei punti chiave dei processi di inferenza.

Definizione 20 (Soddisfacibilità) *Un concetto C è soddisfacibile rispetto a $\mathcal{T}$ se esiste un modello $\mathcal{I}$ di $\mathcal{T}$ tale che $C^{\mathcal{I}}$ non è vuoto. In questo caso si può anche dire che $\mathcal{I}$ è un modello per C.*

Un'altra forma principale di ragionamento è la sussunzione, che verifica se un concetto, *(the subsumer)*, è un soprainsieme, ovvero è più generale, di un'altro, *(the subsumee)*.

Definizione 21 (Sussunzione) *Un concetto C è sussunto da un concetto D rispetto a $\mathcal{T}$ se $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$ per ogni $\mathcal{I}$ di $\mathcal{T}$ e, in tal caso, si scrive $C \sqsubseteq_{\mathcal{T}} D$ oppure $\mathcal{T} \models C \sqsubseteq D$.*

Dalle definizioni di soddisfacibilità e sussunzione derivano quelle di disgiunzione e equivalenza.

Definizione 22 (Equivalenza) *Un concetto C è equivalente da un concetto D rispetto a* $\mathcal{T}$ *se* $C^{\mathcal{I}} = D^{\mathcal{I}}$ *per ogni* $\mathcal{I}$ *di* $\mathcal{T}$ *e, in tal caso, si scrive C* $\equiv_{\mathcal{T}}$ *D oppure* $\mathcal{T} \models C \equiv D$.

Definizione 23 (Disgiunzione) *Due concetti C e D sono disgiunti rispetto a* $\mathcal{T}$ *se* $C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$ *per qualsiasi modello* $\mathcal{I}$ *di* $\mathcal{T}$.

Si può notare che il problema di soddisfacibilità è riconducibile ad un problema di non sussunzione, dove il sussunto è l'insieme vuoto, e viceversa che è possibile risolvere un problema di sussunzione nei termini di un problema di insoddisfacibilità. Da queste considerazioni derivano le seguenti proposizioni.

Proposizione 2 (Riduzione a sussunzione) *Siano C e D due concetti, allora:*

1. *C è insoddisfacibile* $\Longleftrightarrow C \sqsubseteq \bot$;
2. $C \equiv D \Longleftrightarrow C \sqsubseteq D \sqcap D \sqsubseteq C$;
3. *C, D disgiunti* $\Longleftrightarrow (C \sqcap D) \sqsubseteq \bot$.

Proposizione 3 (Riduzione a insoddisfacibilità) *Siano C e D due concetti, allora:*

1. $C \equiv D \Longleftrightarrow C \sqcap \neg D$ *è insoddisfacibile ed anche* $\neg C \sqcap D$ *è insoddisfacibile;*
2. $C \sqsubseteq D \Longleftrightarrow C \sqcap \neg D$ *è insoddisfacibile;*
3. *C, D disgiunti* $\Longleftrightarrow (C \sqcap D)$ *è insoddisfacibile.*

2.3.2 Il ragionamento sulla TBox

Tutte le forme di ragionamento viste singolarmente per i concetti possono essere estese ad una terminologia, senza restrizione alcuna. In particolare però, quando si parla di ragionamento sulla TBox, ci si riferisce alla classificazione, che comporta il posizionamento

di una nuova definizione di concetto nella tassonomia, infatti la maggior parte delle proprietà di una terminologia sono rappresentate dalla relazione di sussunzione tra i termini definiti nella terminologia stessa. Dato un termine è fondamentale stabilire l'insieme dei suoi sussuntori immediati, cioè quell'insieme, minimo rispetto alla sussunzione, di concetti atomici che sussumono il termine stesso. Questo insieme stabilisce una tassonomia tra concetti atomici che organizza gerarchicamente il dominio del problema.

Data una definizione di concetto C, il sistema, tramite test di sussunzione, la classifica tra:

- i concetti più generali che sussumono C;
- i concetti più specifici che C sussume.

La tassonomia dei concetti, oltre a rispecchiare l'organizzazione concettuale del dominio, riveste un ruolo fondamentale per l'efficienza del sistema nelle risposte alle interrogazioni. Disponendo dell'insieme degli immediati sussuntori di ogni concetto atomico è possibile, per esempio, calcolare immediatamente la relazione di sussunzione tra due diversi concetti atomici; semplicemente analizzando la struttura della tassonomia.

2.3.3 Il ragionamento sull'ABox

Fino ad ora abbiamo preso in considerazione i compiti di ragionamento relativi ai concetti e alle TBox, ora ci occuperemo dei compiti di ragionamento relativi alle ABox. Tali compiti sono basati sul controllo della consistenza della parte asserzionale rispetto ad una TBox e quindi hanno senso solo se quest'ultima è definita.

Definizione 24 (Consistenza di un'ABox rispetto ad una TBox) *Dati un'ABox $\mathcal{A}$ e una TBox $\mathcal{T}$ si dice che un'interpretazione $\mathcal{I}$ è un modello di $\mathcal{A}$ rispetto a $\mathcal{T}$, se e solo se $\mathcal{I}$ è un modello per $\mathcal{T}$ e soddisfa tutte le asserzioni definite in $\mathcal{A}$. In questo caso si dice che l'ABox $\mathcal{A}$ e consistente rispetto alla TBox $\mathcal{T}$.*

Sulla base della soddisfacibilità asserzionale è quindi possibile risolvere i problemi di:

- controllo di istanza (*instance check*): ovvero di problemi che richiedono di verificare che un'individuo sia istanza di un dato concetto, o in altri termini, problemi che richiedono di verificare se un'asserzione è implicata o meno dall'ABox;
- recupero (*retrieval problem*): problemi per i quali è necessario trovare di tutti gli individui che sono istanza di un dato concetto;
- realizzazione (*realizzazion problem*): problemi fondati sulla determinazione del concetto più specifico di cui un dato individuo è istanza.

Definizione 25 (Implicazione) *Si dice che un'asserzione α è implicata da un'ABox $\mathcal{A}$, e scriviamo $\mathcal{A} \models \alpha$, se ciascun modello che soddisfa $\mathcal{A}$ soddisfa anche α.*

Per spiegare come sia possibile ricondurre i problemi precedenti alla verifica della consistenza è sufficiente considerare la proposizione:

Proposizione 4 (Riduzione a consistenza su ABox) *$\mathcal{A} \models C(a)$ se e solo se $\mathcal{A} \cup \{\neg\, C\ (\ a\ )\ \}$ è inconsistente.*

2.3.4 Semantica di mondo aperto e semantica di mondo chiuso

Di solito viene stabilita un'analogia tra i database e le logiche descrittive, in cui lo schema del database è paragonato alla TBox e, i dati attuali contenuti in esso vengono assimilati all'ABox. Tuttavia, la semantica dell'ABox differisce dall'usuale semantica delle istanze dei database e questo ha come conseguenza che, mentre l'assenza d'informazione in un database viene interpretata come un'informazione negativa, nell'ABox indica solamente una mancanza di conoscenza.

Per esempio, se l'unica asserzione riguardo *Peter* è:

hasChild(Peter, Harry),

questo, in un ambiente database, viene inteso come rappresentazione del fatto che *Peter* ha un solo figlio, *Harry*. Invece, in un'ABox, l'asserzione precedente esprime solamente

che *Harry* è figlio di *Peter*. Infatti un'ABox ha diversi modelli d'interpretazione: alcuni in cui *Harry* è l'unico figlio e altri in cui ha fratelli e/o sorelle, di conseguenza anche se si indica, tramite un'altra asserzione, che *Harry* è maschio, non è possibile dedurre che tutti i figli di *Peter* sono maschi. L'unico modo di indicare, in un'ABox, che *Harry* è l'unico figlio di *Peter* è quello di dichiararlo esplicitamente aggiungendo l'asserzione di concetto:

$$(\leq 1 \text{ hasChild}) \text{ (Peter)}$$

Ciò significa che, mentre l'informazione in un database è sempre intesa come "completa", l'informazione in un'ABox è generalmente vista come "incompleta". La semantica che caratterizzata l'ABox è detta semantica di mondo aperto (open world semantics), mentre quella dei database viene detta semantica di mondo chiuso (close world semantics).

La semantica ha ripercussioni anche sul modo in cui viene data una risposta alle interrogazioni (query). Essenzialmente una query è una descrizione di una classe di oggetti, e nel contesto delle logiche descrittive questo corrisponde alla descrizione, anche complessa, di un concetto. Da un punto di vista logico, la valutazione di una query in un database non rappresenta un ragionamento, ma la computazione di un'interpretazione, ovvero il controllo di un modello finito. Differentemente, in un'ABox la risposta ad una query è molto complessa e richiede un ragionamento non banale. L'esempio che segue, detto "l'esempio di Oedipus", illustra meglio questa differenza.
Esso è basato sulla storia di Edipo, *Oedipus*, dell'antica mitologia greca, nella quale viene raccontato di come *Oedipus* uccise suo padre, sposò sua madre *Iokaste* e ebbe da lei un figlio: *Polyneikes*; infine anche *Polyneikes* ebbe un figlio: *Thersandros*. Un'ABox che rappresenti questi eventi si potrebbe esprimere come:

$$hasChild(Iokaste, Oedipus)$$
$$Parricida(Oedipus)$$
$$hasChild(Oedipus, Polyneikes)$$
$$hasChild(Iokaste, Polyneikes)$$
$$hasChild(Polyneikes, Thersandros)$$

$$\neg Parricida(Thersandros)$$

Supponiamo ora di voler interrogare l'ABox per sapere se "*Iokaste ha un figlio che è un parricida, che a sua volta ha un figlio che non è un parricida*". In altri termini si chiede di dare una risposta al problema di istance checking:

$$\mathcal{A}_{OE} \models (\exists hasChild.(Parricida \sqcap \exists hasChild.\neg Parricida))(Iokaste)?$$

Uno dei ragionamenti che si potrebbe effettuare per risolvere questo problema è il seguente.

Iokaste ha due figli. Uno di questi, *Oedipus*, è un parricida ed ha sua volta un figlio chiamato *Polyneikes*; di quest'ultimo, però, nulla è detto riguardo il fatto che non sia un parricida. L'altro figlio di *Iokaste* è *Polyneikes* ma, come prima, nulla è detto riguardo il fatto che sia un parricida e, in base a tutto questo, si potrebbe concludere che il problema è senza risposta.

Tuttavia il ragionamento corretto è differente. Tutti i modelli di $\mathcal{A}_{OE}$ possono essere divisi in due classi: una in cui *Polyneikes* è parricida e l'altra in cui non lo è. In un modello della prima specie *Polyneikes* è figlio di *Iokaste*, è parricida ed ha un figlio, *Thersandros*, che non lo è; in un modello della seconda specie *Oedipus* è figlio di *Iokaste*, è parricida ed ha un figlio, *Polyneikes*, che non lo è. Quindi, in entrambi i modelli, *Iokaste* è implicata dal problema, che perciò ha una risposta affermativa.

2.4 La famiglia delle logiche $\mathcal{AL}$

Si può ottenere una maggior espressività da un linguaggio aumentandone il numero di costruttori. L'estensione del linguaggio $\mathcal{AL}$ mediante sott'insiemi di costruttori genera dei particolari linguaggi denotati come linguaggi della famiglia $\mathcal{AL}$. A ciascuno di essi è assegnato un nome nella forma:

$$\mathcal{AL}\ [\mathcal{U}][\mathcal{E}][\mathcal{N}][\mathcal{C}],[\mathcal{O}],[\mathcal{H}],[\mathcal{I}],[\mathcal{Q}],[\mathcal{R}],[\mathcal{R}^{+}],[f]$$

dove ciascuna lettera indica la presenza di un particolare costruttore, secondo quanto riportato nella tabella a seguire.

È bene notare che dal punto di vista semantico non tutti questi linguaggi sono distinguibili. Infatti, tramite l'interpretazione semantica è facile vericare che:

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

e che

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

Questo dimostra che tramite l'operatore di unione e il qualificatore universale qualificato è possibile esprimere il complemento e viceversa. In ragione di questo, molti testi indicano indistintamente con $\mathcal{ALC}*$, sia i linguaggi $\mathcal{ALC}*$ veri e proprie, sia quelli $\mathcal{ALCUE}*$.

2.5 Complessità del ragionamento

In questo paragrafo viene mostrata la complessità di alcuni linguaggi terminologici della famiglia $\mathcal{AL}$. La tabella 2.4 indica le complessità, riferita a terminologie vuote, per sussunzione terminologica e soddisfacibilità di una base di conoscenza.

Senza entrare nel merito, si è visto che un sottoinsieme ridottissimo di costrutti permette di mantenere trattabile un linguaggio; viceversa l'aumento di espressività rischia di renderlo indecidibile. Dato gran parte degli operatori "interessanti" portano il frammento del linguaggio ad essere intrattabile, nei KRS, basati su DLs, effettivamente implementati si è reso necessario scendere a compromessi per gestire quest'intrinseca intrattabilità ([Bor92]). Le strade seguite per lo sviluppo di questi sistemi si possono riassumere in:

- Linguaggi limitati: dove il linguaggio terminologico ammesso è limitato ai costrutti che mantengono il problema trattabile, ed il sistema a logiche descrittive viene usato come un componente di risolutori più generali. Tale approccio è stato scelto per lo sviluppo di CLASSIC e KRYPTON.

Costrutto	Sintassi DL	Linguaggio
Nome concetto	C	$\mathcal{AL}*$
Nome ruolo	R	
Insieme universo	$\top$	
Insieme vuoto	$\bot$	
Negazione concetto atomico	$\neg A$	
Intersezione di concetti	$C_1 \sqcap C_2$	
Qualificatore esistenziale	$\exists R.\top$	
Qualificatore universale	$\forall R.C$	
Negazione	$\neg C$	$\mathcal{C}$
Disgiunzione di concetti	$C_1 \sqcup C_2$	$\mathcal{U}$
Qual. esistenziale qualificato	$\exists R.C$	$\mathcal{E}$
Restrizione cardinalità	$(\leqslant nR)$ $(\geqslant nR)$ $(= nR)$	$\mathcal{N}$
Collezione di individui	$a_1, a_2, \ldots, a_n$	$\mathcal{O}$
Gerarchia di ruoli	$R_1 \subseteq R_2$	$\mathcal{H}$
Ruolo inverso	R^-	$\mathcal{I}$
Restrizione cardinalità qualificata	$(\leqslant nR.C)$ $(\geqslant nR.C)$ $(= nR.C)$	$\mathcal{Q}$
Congiunzione di ruoli	$R_1 \sqcap R_2$	$\mathcal{R}$
Ruoli transitivi	$R \subseteq R^+$	$\mathcal{R}^+$
Funzioni parziali	$R : \{state_1, state_2, ...\}$	f

Tabella 2.3: Nomenclatura delle estensioni della logica $\mathcal{AL}$

Linguaggio	Sussunzione	Soddisfacibilità
$\mathcal{AL}$	P	P
$\mathcal{ALN}$	P	P
$\mathcal{ALE}$	NP	coNP
$\mathcal{ALR}$	NP	coNP
$\mathcal{ALU}$	coNP	NP
$\mathcal{ALC}$	PSPACE	PSPACE

Tabella 2.4: Complessità per di alcuni linguaggi della famiglia $\mathcal{AL}$

- Ragionatori Completi: dove si è scelto di sviluppare un sistema che implementa un ragionatore completo per il frammento del linguaggio terminologico adottato, lasciando in secondo piano il problema della trattabilità.

- Ragionatori incompleti: dove il sistema, sebbene copra un frammento del linguaggio molto espressivo, non implementa tutte le inferenze che derivano dalla semantica dei costrutti e lascia quindi la possibilità di risultati impredicibili. Tale approccio è stato scelto ad esempio per LOOM.

2.6 Cenni sugli algoritmi

Uno degli algoritmi più completi per la valutazione della soddisfacibilità, in grado di operare su una vasta classe di linguaggi, è l'algoritmo a regole, basato sul tableaux, presentato per la prima volta in [SSS91]. La sua caratteristica fondamentale è quella di essere abbastanza generale da permettere l'estensione del linguaggio terminologico in modo uniforme. Infatti possono essere aggiunti nuovi operatori, semplicemente aggiungendo nuove regole di completamento, che tengano naturalmente conto della decidibilità del linguaggio, a quelle di base. Gli algoritmi funzionali, derivati come casi particolari del metodo a tableaux, non mantengono questa generalità rendendo necessario, per ogni nuovo operatore introdotto, uno studio ad hoc del problema.

Vi sono diversi lavori che, usando la tecnica di estendere le regole di base, includono nuovi operatori nel linguaggio oppure lo specializzano ad un dominio particolare. Tra queste estensioni si citano:

- introduzione dei ruoli funzionali [HNSS90];
- l'operatore epistemico K, usato nel linguaggio d'interrogazione delle basi di conoscenza [DLN + 92, DLN + 93];
- l'introduzione di domini concreti, quali possono essere stringhe, numeri, tipi di dato del linguaggio ospite [BH91b];
- introduzione di collezioni di individui per trattare problemi d'interpretazione del linguaggio naturale [Fra93];
- gli assiomi terminologici [BDS93].

2.7 La logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

Sia $\mathbf{C}$ un insieme numerabile di nomi di concetto, $\mathbf{R}$ un insieme numerabile di nomi di ruolo, $\mathbf{F}$ un insieme numerabile di nomi di funzioni parziali (concrete feature), $\mathbf{O}$ un insieme numerabile di nomi di individui e $\mathbf{O_c}$ un insieme numerabile di nomi di oggetti appartenenti a domini concreti, tali che $\mathbf{C}$, $\mathbf{R}$, $\mathbf{F}$, $\mathbf{O}$, $\mathbf{O_c}$ siano a due a due disgiunti. $\mathbf{R}$ è partizionato in due insiemi disgiunti: $\mathbf{P}$ l'insieme dei nomi di ruolo non transitivi, e $\mathbf{T}$ l'insieme dei nomi di ruolo transitivi, tali che $\mathbf{R} = \mathbf{P} \cup \mathbf{T}$. Sia inoltre α, una funzione detta assegnamento di variabile, che mappa i nomi di oggetti relativi ai domini concreti nei loro valori in $\Delta_{\mathcal{D}}$.

Il linguaggio $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ è introdotto nella tabella 2.5, secondo lo stile semantico Tarski, considerando una funzione d'interpretazione $I_{\mathcal{D}} = (\Delta_I, \Delta_{\mathcal{D}}, \cdot^I)$ dove $\Delta_I \cap \Delta_{\mathcal{D}} = \emptyset$.

Sintassi	Semantica
Concetti (R$\in \mathbf{R}$, S $\in \mathbf{S}$, f $\in \mathbf{F}$)	
$\top$	Δ_I
$\bot$	$\emptyset$
A	$A^{\mathcal{I}} \subseteq \Delta_I$
$\neg$ C	$\Delta_I \setminus C^{\mathcal{I}}$
C $\sqcap$ D	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
C $\sqcup$ D	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
$\exists$ R.C	$\{ a \in \Delta_I \mid \exists\, b \in \Delta_I : (a,b) \in R^{\mathcal{I}}, b \in C^{\mathcal{I}}\}$
$\forall$ R.C	$\{ a \in \Delta_I \mid \forall\, b \in \Delta_I : (a,b) \in R^{\mathcal{I}} \Longrightarrow b \in C^{\mathcal{I}}\}$
$\exists(\leq$ n S)	$\{ a \in \Delta_I \mid \|\{ b \in \Delta_I \mid (a,b) \in S^{\mathcal{I}}\}\| \leq n\}$
$\exists(\geq$ n S)	$\{ a \in \Delta_I \mid \|\{ b \in \Delta_I \mid (a,b) \in S^{\mathcal{I}}\}\| \geq n\}$
$\exists(=$ n S)	$\{ a \in \Delta_I \mid \|\{ b \in \Delta_I \mid (a,b) \in S^{\mathcal{I}}\}\| = n\}$
$\exists(\leq$ n S.C)	$\{ a \in \Delta_I \mid \|\{ b \in \Delta_I \mid (a,b) \in S^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\| \leq n\}$
$\exists(\geq$ n S.C)	$\{ a \in \Delta_I \mid \|\{ b \in \Delta_I \mid (a,b) \in S^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\| \geq n\}$
$\exists(=$ n S.C)	$\{ a \in \Delta_I \mid \|\{ b \in \Delta_I \mid (a,b) \in S^{\mathcal{I}} \wedge b \in C^{\mathcal{I}}\}\| = n\}$
$\exists$ $f_1,\ldots,f_n$.P	$\{ a \in \Delta_I \mid \exists\, x_1,\ldots,x_n \in \Delta_{\mathcal{D}} : (a,x_1) \in f_1^{\mathcal{I}}, \ldots, (a,x_n) \in f_n^{\mathcal{I}}, (x_1,\ldots,x_n) \in P^{\mathcal{I}}\}$
$\forall$ f.$\bot_{\mathcal{D}}$	$\{ a \in \Delta_I \mid \neg\exists\, x_1 \in \Delta_{\mathcal{D}} : (a,x_1) \in f^{\mathcal{I}}\}$
Ruoli e concrete features	
R	$R^{\mathcal{I}} \subseteq \Delta_I \times \Delta_I$
f	$f^{\mathcal{I}} : \Delta_I \rightarrow \Delta_{\mathcal{D}}$

Tabella 2.5: Sintassi e semantica della logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

Assiomi	
Sintassi	Soddisfacibile se
$R \in T$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$
$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$
$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$

Tabella 2.6: Assiomi in logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

Asserzioni (a, b $\in \mathbf{O}$, x, $x_i \in \mathbf{O_c}$)	
Sintassi	Soddisfacibile se
a : C	$a^{\mathcal{I}} \in C^{\mathcal{I}}$
(a, b) : R	$(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$
(a, x) : f	$(a^{\mathcal{I}}, \alpha(x)) \in f^{\mathcal{I}}$
$(x_1,\ldots,x_n)$: P	$(\alpha(x_1,\ldots,x_n)) \in P^{\mathcal{I}}$

Tabella 2.7: Asserzioni in logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

2.7.1 Gerarchie di ruoli

Una delle estensioni introdotte dalla logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ sono le gerarchie di ruoli; con riferimento ad esse vengono riportate le definizioni:

Definizione 26 (Gerarchia di ruoli) *Si definisce gerarchia di ruoli $\mathcal{R}$ un insieme finito di assiomi di inclusione di ruoli.*

Sia $\sqsubseteq^*$ la chiusura transitiva e riflessiva di $\sqsubseteq$ rispetto ad una gerarchia di ruoli $\mathcal{R}$.

Definizione 27 (Sub-ruolo) *Dato $\sqsubseteq^*$, si definisce "insieme dei sotto-ruoli di R" l'insieme: $R^{\downarrow} = \{S \in \mathbf{R} | S \sqsubseteq^* R\}$.*

Definizione 28 (Super-ruolo) *Dato $\sqsubseteq^*$, si definisce "insieme dei super-ruoli di R" l'insieme: $R^{\uparrow} = \{S \in \mathbf{R} | R \sqsubseteq^* S\}$.*

Definizione 29 (Ruolo semplice) *Si definisce ruolo semplice ciascun elemento appartenente all'insieme $\mathbf{S} = \{R \in \mathbf{P} | R^{\downarrow} \cap \mathbf{T} = \emptyset\}$, detto "insieme dei ruoli semplici".*

Il linguaggio concettuale di $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ è sintatticamente limitato, dato che permette di esprimere vincoli di cardinalità solamente per i ruoli semplici. Infatti, senza questa limitazione, la logica diventerebbe indecidibile, in accordo con [HST99].

2.7.2 I domini concreti

Nel campo delle logiche descrittive il ragionamento sui domini concreti è stato oggetto di approfonditi studi, specialmente in ragione della sua importanza per le applicazioni pratiche.

In particolare in [BH91b] è stata investigata la logica $\mathcal{ALC}(\mathcal{D})$, mostrando che, unitamente ad una procedura di decisione per i domini concreti $\mathcal{D}$, essa è decidibile.
Partendo da questo risultato si è cercato d'integrare i domini concreti anche in logiche più

espressive, come ad esempio in $\mathcal{ALCNH}_{\mathcal{R}^+}$ [HM00], ma sfortunatamente, si è constatato che questo causa dei problemi di indecidibilità inferenziale. Per esempio, in [BH93] viene dimostrato come la logica $\mathcal{ALC}(\mathcal{D})$, unita all'operatore per la chiusura transitiva dei ruoli, risulti indecidibile se i domini concreti che la estendono sono troppo espressivi, ovvero se presentano un set di operatori complesso. Inoltre, in [Lut99], viene provato che gli assiomi generalizzati di inclusione nella logica $\mathcal{ALC}(\mathcal{D})$ possono rendere quest'ultima indecidibile. Un'eccezione a [Lut99] è stata verificata in [LSW00], dove viene investigata una logica $\mathcal{ALC}$ in combinazione con il dominio concreto dei numeri razionali contenente i soli operatori di uguaglianza e disuguaglianza. Si può notare che, in accordo con [BH93], la logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ presenta i ruoli transitivi ma non presenta l'operatore per la chiusura transitiva dei ruoli. La presenza delle gerarchie di ruoli fa però si che essa risulti potenzialmente indecidibile. Le gerarchie dei ruoli e i ruoli transitivi forniscono infatti la stessa espressivita degli assiomi generalizzati di inclusione. Tramite le gerarchie dei ruoli e i ruoli transitivi è possibile dichiarare, implicitamente, un ruolo universale che, usato in combinazione con una restrizione di valore, può avere lo stesso effetto di un assioma di inclusione generalizzata. Pertanto, come dimostrato in [HMT01], la decidibilita della logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ è legata all'utilizzo di domini concreti con limitata espressività.

Così come in [BH91a], si può fornire una definizione della nozione di dominio concreto nel modo seguente:

Definizione 30 (Dominio concreto) *Un dominio concreto D è una coppia $(\Delta_{\mathcal{D}}, \Phi_{\mathcal{D}})$, dove $\Delta_{\mathcal{D}}$ è l'insieme chiamato dominio e $\Phi_{\mathcal{D}}$ rappresenta l'insieme numerabile dei nomi di predicato (denotati da $P_1, P_2 \ldots$).*

La funzione di interpretazione $I_{\mathcal{D}}$ mappa ciascun nome di predicato $P_n \in \Phi_{\mathcal{D}}$, con molteplicità n, in un sottoinsieme $P^I \subseteq \Delta_D^n$ e ciascun $\{ocd_1, ocd_2, \ldots, ocd_2\} \in \mathbf{O_c}$ è mappato in un elemento di $\Delta_{\mathcal{D}}$. Assumiamo che $\bot_D$ la negazione di $\top_D$.

Definizione 31 (Dominio concreto ammissibile) *Un dominio concreto è detto ammis-*

sibile se e solo se $\Phi_{\mathcal{D}}$ è chiuso sotto negazione, $\top_D$ è contenuto in $\Phi_{\mathcal{D}}$ e il problema soddisfacibilità: $P_1^{n_1}(x_{11}, \ldots, x_{1n_1}) \wedge \ldots \wedge P_m^{n_m}(x_{m1}, \ldots, x_{mn_m})$, è decidibile.

Nella defizione precedente: m è un numero finito, $P_i^{n_i} \in \Phi_{\mathcal{D}}$, n_i è la molteplicità di P e $x_{jk} \in \mathbf{O_c}$.

Capitolo 3

RACER

3.1 Descrizione generale

RACER (Renamed ABox and Concept Expression Reasoner)[HM99, HM01] è un KRS basato su un algoritmo, altamente ottimizzato, del tableaux calculus. Il sistema implementa la DL $\mathcal{ALCQHI}_{\mathcal{R}+}$, conosciuta anche con il nome di SHIQ [HST00], che, come visto nel capitolo precedente, estende la logica $\mathcal{ALC}$ con l'introduzione delle restrizioni numeriche, delle gerarchie dei ruoli, dei ruoli inversi e dei ruoli transitivi. In aggiunta a queste caratteristiche "standard", RACER fornisce un supporto per il ragionamento algebrico includendo i domini concreti, tramite i quali è in grado di trattare diverse tipologie di problemi legate ai numeri interi, ai numeri reali e alle stringhe. In ragione di quest'ultima caratteristica si può dire che RACER implementa a tutti gli effetti la logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, detta anche SHOIQ(D).

RACER distingue chiaramente la conoscenza intensionale, TBox, da quella estensionale, ABox, separando le funzionalià per entrambe le componenti. Nella parte intensionale è possibile esprimere la relazione di sussunzione tra due concetti, dare definizioni multiple dello stesso concetto o, addirittura, definizioni cicliche, che il sistema interpreta secondo la semantica descrittiva.

I principali servizi inferenziali forniti, riguardo la parte intensionale, si possono riassumere in:

- Verifica della consistenza di un concetto rispetto alla TBox.
- Verifica dell'esistenza della sussunzione tra due concetti rispetto alla TBox.
- Ricerca di tutti i concetti inconsistenti rispetto alla TBox.
- Determinazione della tassonomia dei concetti rispetto alla TBox.

In aggiunta a questi troviamo i servizi inferenziali implementati per la parte estestensionale:

- Verifica della consistenza dell'ABox rispetto al TBox dato.
- Test per la verifica dell'appartenenza di un individuo dell'ABox ad un determinato concetto del TBox.
- Recupero di tutti gli individui dell'ABox appartenenti a un dato concetto del TBox.
- Computazione del concetto più specifico, rispetto al TBox, per il quale un individuo del ABox è istanza.
- Computazione dei riempitivi di un ruolo in riferimento ad un determinato individuo.

Sviluppato, interamente in CommonLisp, dall'Università di Amburgo, Germania, RACER viene ora attivamente supportato dalla "Concordia University" di Montreal, Canada, e dall' "Università delle Scienze Applicate" di Wedel, Germania.

3.1.1 La versione 1.7

La versione corrente del sistema è la 1.7, che presenta le seguenti peculiarità :

- Supporto per lo standard DIG; tale standard permette l'interfacciamento di RACER server con editor grafici di ontologie, come ad esempio OilEd.

- Supporto per i formati OWL, DAM+OIL,RDFS e RDF standard.
- Nuove funzionalità di query riguardanti la TBox.
- Possibilità di modificare la TBox a “tempo di esecuzione”, senza quindi bisogno dell’intera ritrasmissione della TBox stessa.
- Miglioramento delle performance per le funzioni di interrogazione sull’ABox.
- Possibilità di controllare il ragionamento per rispondere alle interrogazioni sull’ABox.
- Supporto per l’accesso multiutente alla base di conoscenza.
- Possibilità di utilizzare, in rete, RACER Server multipli, accessibili mediante un’unico RACER Proxy con funzionalità di bilanciamento del carico.
- Introduzione dei domini concreti: numeri interi, cardinali, reali e stringhe.
- Utilizzo mediante un client grafico interattivo: RICE (RACER Interactive Client Enviroment), sviluppato da Ronal Cornet presso l’Università di Amsterdam, Olanda, e scaricabile all’indirizzo: http://www.b1g-systems.com/ronald/rice.

Dalla realese 1.7.17 è inoltre disponibile un nuovo linguaggio di interrogazione per la parte estensionale, chiamato new RACER Query Language (nRQL), che permette l’utilizzo di variabili per la creazione d’interrogazioni complesse. Negli ultimi mesi si sono succedute numerose release del sistema, passando release 1.7.17 all’attuale 1.7.23, utilizzata nella presente tesi.

3.1.2 Assunzione di mondo aperto e dei nomi univoci

Come molti altri motori inferenziali anche RACER basa il suo ragionamento sull’assunzione di mondo aperto, Open World Assumption (OWA). Questo significa che, tutto quello che non è provato che sia vero non è detto che, per forza, sia falso. Il sistema può quindi

fornire solamente risposte in termini di “è vero”, indicato dal simbolo “T”, oppure “non posso dire niente in base alle informazioni che possiedo”, restituendo il valore “NIL”. In aggiunta all’OWA, RACER impiega l’UNA, in modo tale che tutti gli individui specificati nell’ABox siano mappati in diversi elementi dell’universo di rappresentazione.

3.1.3 Domini concreti

I servizi di ragionamento algebrico, offerti da RACER, si traducono nella possibilità di trattare:

- Restrizioni di minimo e/o massimo nel campo degli interi.
- Equazioni e disequazioni polinomiali lineari nel campo dei reali.
- Equazioni polinomiali non lineari multivariabili nel campo dei complessi.
- Equivalenze e disuguaglianza di stringhe.

Gli algoritmi utilizzati per il ragiomento sui domini concreti derivano dall’algoritmo di risoluzione di Jaffar, [JL87] e da quello di Gomory [Wei92], a cui si rimanda per ulteriori approfondimenti.

3.1.4 Aree di applicazione

Ci sono numerosi documenti che descrivono o offrono spunti su come utilizzare RACER per risolvere problemi inerenti diverse aree di applicazione:

- Web semantico: [JGC01];
- Commercio Elettronico: [NSDM03];
- Bioinformatica: [BSKB];
- Linguaggio naturale: [GKS02, BFG$^+$];
- Ingegneria del Software: [ABdR99, Ber02];

- Sistemi informativi territoriali (GIS: Geographic Information Systems): [Wes03].

3.2 Installazione e funzionamento

RACER viene distribuito in forma eseguibile, senza restrizioni e senza che sia necessaria una licenza d'uso. La versione eseguibile, per la quale viene fornito anche il manuale [HM04], in formato pdf, è conosciuta con il nome di "RACER Server". RACER Server è disponibile compilato per piattaforma Windows, oppure Linux o Mac e può essere scaricato, gratuitamente, da uno dei seguenti siti:

Europa: http://www.fh-wedel.de/ mo/racer/

America: http://www.cs.concordia.ca/ faculty/haarslev/racer/

3.2.1 Requisiti di sistema

Dopo aver scaricato il file, contenente la versione eseguibile idonea al sistema operativo che s'intende utilizzare, l'installazione di RACER Server si conclude scompattandolo, mediante WinZip per Windows o gzip/tar per Unix, in una directory a scelta; questa operazione richiede circa 20 Megabyte di spazio libero sul disco fisso.

Benchè non siano formalmente indicati dei limiti alla quantità di memoria necessaria, per il corretto funzionamento di RACER Sever sono consigliabili almeno 32 Megabyte di memoria RAM disponibile. Per l'utilizzo commerciale, le personalizzazioni e il supporto per la compilazione dei sorgenti su diverse architettture, bisogna rivolgersi ad una ditta esterna (RACER Service gmbh).

3.2.2 RACER Server

RACER Server è un file eseguibile che può essere lanciato dalla riga di comando di una shell Linux o di una finestra Dos.

Il server può scambiare informazioni con l'esterno in tre modi: tramite file, tramite un canale basato sul protocollo HTTP, oppure utilizzando un canale su socket TCP/IP. In base alle opzioni specificate all'avvio dell'applicazione si possono impostare diversi modi di funzionamento e quale interfaccia si preferisce utilizzare.

Interfaccia basata sui file

Se la base di conoscenza e le query sono gia disponibili sotto forma di file è possibile lanciare RACER Sever con il comando:

$ racer -f <*nomefileKB*> -q <*nomefileQUERY*> -o <*nomefileOUTPUT*>

dove <*nomefileKB*> rappresenta il file contenente la base di conoscenza completa di TBox e ABox, <*nomefileQUERY*> è il file contenente le query, nel linguaggio proprietario, a cui il sistema deve rispondere e <*nomefileOUTPUT*> è il file sul quale si vuole reindirizzare l'output.

La sintassi, da utilizzare per processare il file di input contenente la KB, viene determinata dal sistema in base all'estensione del file stesso. I formati, e le estensioni, supportati sono:

LISP (List Processing (programming language));

KRSS (Knowledge Representation System Specification);

RACER linguaggio nativo di RACER;

RDFS (Resource Description Framework Standard);

DAML-OIL (DARPA Agent Markup Language - Ontology Inference Layer);

DIG (DL Implementation Group);

OWL (Web Ontology Language);

Qualora si desideri forzare la sintassi è possibile utilizzare una delle opzioni: *"-f: -rdfs"*, *"-f: -daml"*, *"-f: -dig"*.

L'opzione *"-t <nrosecondi>"*, particolarmente utilizzata quando si vogliono risolvere dei problemi di benchmarking, permette di specificare un timeout del server.

É da notare che RACER supporta il formato OWL incorporando al suo interno un parser OWL basato sul progetto Wilbur[1], tramite il quale è in grado di importare ed esportare KB.

Interfaccia TCP SOCKET e HTTP

Il nome dell'eseguibile, senza opzioni, avvia la modalità server e apre le interfacce basate su socket TCP e sul protocollo HTTP. La porta TCP di comunicazione è, per default, la numero 8088 ma è possibile specificarne una diversa tramite l'opzione *"-p <nroporta>"*.

L'idea di base dell'interfaccia socket è quella di aprire un canale di comunicazione che permetta al client e al server di comunicare mediante lo scambio di stringhe; grazie alle API JAVA, fornite nel pacchetto JRACER2, è possibile sviluppare applicazioni che comunicano con il server sfruttando questo metodo. E' da notare che la versione attuale dell'interfaccia TCP non permette il logging delle richieste.

Per quanto riguarda l'interfaccia HTTP, tramite l'opzione *"-http <nroporta>"* si può impostare un numero di porta, diverso dal valore standard 8080, oppure, con *"-http 0"* si può disabilitare questo servizio, che altrimenti è abilitato di default. Le applicazioni che vogliono sfruttare questa interfaccia devono utilizzare il metodo POST. Per default è previsto il logging delle richieste POST in ingresso e, con le opzioni *"-httplogdir <directory>"* e *"-nohttpconsolelog"* si può rispettivamente redirezionare il logging in una directory specificata oppure di disabilitarlo.

Quando RACER è avviato in modalià server è possibile caricare una base di conoscenza, ancor prima che i client si connettano, mediante *"-init <nomefileKB>"*. In alcune applicazioni può essere necessario indirizzare RACER Server mediante delle connessio-

[1] Wilbur è parte dell'attività di ricerca che Nokia svolge sul Web Semantico

ni persistenti e quindi potrebbe risultare utile impostare, con *"-c <numsecondi>"*, un meccanismo di timeout per controllare la durata di tali connessioni.

Opzioni addizionali

RACER server si riserva uno spazio di memoria per lo stack pari a 320000 byte ma, qualora questo sia insufficiente per processare certe interrogazioni, lo si può aumentare lanciando l'applicazione seguita dall'opzione *"-s <value>"*. Per problemi legati alla sicurezza, specialmente in ambienti distribuiti, è inibito per default l'utilizzo di certi comandi che potrebbero generarare nuovi file sul computer nel quale è attivo RACER Server; per disabilitare questa caratteristica, attivando quindi la modalià insicura (unsafe), è necessario specificare l'opzione *"-u"*.

3.3 Il linguaggio proprietario

Tra le varie possibilità, che il sistema offre per descrivere le KB, quella rappresentata dal linguaggio proprietario, RACER Language Interface (RLI), è senza dubbio la più vantaggiosa, perchè permette il completo utilizzo delle potenzialità del sistema, supportando la logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$. Il linguaggio proprietario di RACER è un linguaggio di derivazione KRSS [PSS93], con caratteristiche simili al Lisp, che, a differenza di altri linguaggi, risulta di facile comprensione anche per il programmatore; tramite esso è possibile specificare la conoscenza intensionale, estensionale, effettuare delle interrogazioni semplici e, tramite la sua estensione nRQL[HM04, pag.139-159], complesse. Inoltre fornisce tutte le funzioni per la gestione completa del sistema. L'elenco completo di tutte le funzioni, suddivise per categoria, è riportato in appendice 8.6.1.

3.3.1 Convenzioni e abbreviazioni

RLI introduce i ruoli e distingue due tipi di features: le concrete features che, in accordo con quanto specificato nella logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, indicano le funzioni parziali, cioè

	Semantica	
Terminologia	$\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$	RACER
Ruolo	$R^{\mathcal{I}} \subseteq \Delta_I \times \Delta_I$	$R^{\mathcal{I}} \subseteq \Delta_I \times \Delta_I$
Concrete Feature	$f^{\mathcal{I}} : \Delta_I \rightarrow \Delta_{\mathcal{D}}$	$f^{\mathcal{I}} : \Delta_I \rightarrow \Delta_I$
Abstract Feature	———	$AN^{\mathcal{I}} : \Delta_I \rightarrow \Delta_{\mathcal{D}}$

Tabella 3.1: Significato dei termini.

quelle relazioni che hanno come riempitivo un elemento appartenente ai domini concreti, e le abstract features, ovvero quei ruoli particolari in cui la cardinalità massima dell'insieme dei riempitivi (fillers) è ristretta ad un elemento, che non appartiene ai domini concreti. Gli elementi appartenenti ai domini concreti vengono chiamati "oggetti", mentre gli altri vengono semplicemente detti "individui".

Gli insiemi considerati da RACER [HM04, pag.45,50,51,54], e ai quali si farà riferimento nel seguito sono:

- $\mathcal{C}$: insieme dei nomi di concetto, o dei concetti atomici;
- $\mathcal{R}$: insieme dei nomi di ruolo;
- $\mathcal{R}^+$: insieme dei nomi di ruolo transitivi;
- $\mathcal{F}$: insieme dei nomi di abstract feature;
- $\mathcal{A}$: insieme dei nomi di concrete feature;
- $\mathcal{I}$: insieme degli individui;
- $\mathcal{O}$: insieme degli oggetti;

La tabella 3.2 riporta le abbreviazioni che verranno utilizzate nei prossimi paragrafi.

C	Descrizione di concetto
CN	Nome di concetto
R	Descrizione di ruolo
RN	Nome di ruolo
AN	Nome di concrete feature
IN	Nome di individuo
ON	Nome di oggetto
CDC	Concetto basato su un dominio concreto

Tabella 3.2: Abbreviazioni.

3.3.2 Sintassi e grammatica

Operatori booleani

Il linguaggio definisce i costruttori per i tre operatori booleani della logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$: l'operatore unario di complemento e gli operatori ennari di intersezione ed unione.

Significato	Notazione DL	Sintassi *RACER*
Complemento	$\neg$C	(not C)
Intersezione	$C_1 \sqcap ... \sqcap C_n$	(and C_1 ... C_n)
Unione	$C_1 \sqcup ... \sqcup C_n$	(or C_1 ... C_n)

Tabella 3.3: Operatori booleani.

Qualificatori: esistenziale ed universale

I qualificatori, esistenziale e universale, permettono di dichiarare il tipo di riempitivo ammesso per il ruolo in oggetto; tramite il qualificatore universale s'impone che tutti i riempitivi per il ruolo siano istanze del concetto specificato, mentre con il qualifica-

tore esistenziale s'impone che esista almeno un riempitivo che sia istanza del concetto specificato.

Significato	Notazione DL	Sintassi *RACER*
Qualificatore universale	$\forall$ R.C	(all R C)
Qualificatore esistenziale	$\exists$ R.C	(some R.C)

Tabella 3.4: Qualificatori.

Restrizioni di cardinalità

I costrutti per le restrizioni di cardinalità, applicabili solo ai ruoli non transitivi che non hanno sotto-ruoli transitivi, permettono di specificare un limite minimo, massimo o esatto, sul numero di riempitivi che il ruolo deve avere e, se qualificati, anche il tipo di concetto di cui i riempitivi devono essere istanze.

Significato	Notazione DL	Sintassi *RACER*
Restrizione cardinalità massima	$\leq$ n R	(at-most n R)
Restrizione cardinalità minima	$\geq$ n R	(at-least n R)
Restrizione cardinalità esatta	$=$ n R	(exactly n R)
Restrizione cardinalità massima qualificata	$\leq$ n R.C	(at-most n R C)
Restrizione cardinalità minima qualificata	$\geq$ n R.C	(at-least n R C)
Restrizione cardinalità esatta qualificata	$=$ n R.C	(exactly n R C)

Tabella 3.5: Restrizioni di cardinalità

Regole per la costruzione dei concetti e dei ruoli

Ciascuna TBox contiene due concetti implicitamente dichiarati: il concetto top, $\top$, che denota l'insieme più generale nella gerarchia dei concetti, e il concetto bottom, $\bot$, che rappresenta il concetto incosistente. Ad essi ci si può sempre riferire tramite le parole

riservate “*top*” (oppure “top”) e “*bottom*” (oppure “bottom”). Le regole sintattiche astratte che permettono la costruzione di concetti complessi sono mostrate nella tabella.

C	$\longrightarrow$	CN	|
		top	|
		bottom	|
		(not C)	|
		(and C_1...C_n...)	|
		(or C_1...C_n...)	|
		(some R C)	|
		(all R C)	|
		(at-least n R)	|
		(at-most n R)	|
		(exactly n R)	|
		(at-least n R C)	|
		(at-most n R C)	|
		(exactly n R C)	|
		(a AN)	|
		(an AN)	|
		(no AN)	|
		CDC	

Tabella 3.6: Regole per la costruzione dei concetti.

Dalla tabella riportata sopra, si può osservare che ai nomi di ruolo è applicabile l’operatore “inv”, il quale permette di riferirsi al ruolo inverso, ovvero al ruolo che ha dominio e codomio invertiti rispetto al ruolo specificato.

R	$\longrightarrow$	RN	|
		(inv RN)	

Tabella 3.7: Regole per la costruzione dei ruoli.

Domini Concreti

Il sistema supporta diversi insiemi numerici: i numeri naturali, i razionali, i reali, i complessi e le stringhe, considerandoli tra loro disgiunti. Per ciascuno di essi è specificato un insieme finito di operatori relazionali ed aritmetici. Per gli attributi, RLI implementa un

Insieme	Tipo RACER	Operatori relazionali	Operatori aritmetici
$\mathbb{N}$	cardinal	<, <=, >, >=, =, <>, divisible, not-divible	*, +
$\mathbb{Z}$	integer	equal, min, max	*, +
$\mathbb{R}$	real	<, <=, >, >=, =, <>	*, +
$\mathbb{C}$	complex	<, <=, >, >=, =, <>	*, +, expt[1]
$string$	string	string=, string<>	

[1]L'operatore aritmetico "expt" rappresenta l'operatore esponenziale e permette il calcolo dell'elevamento a potenza.

Tabella 3.8: Domini concreti supportati ed operatori.

particolare operatore esistenziale, "an" oppure "a", e il suo negato "no"; inoltre, tramite i CDC, permette di specificare delle restrizioni sul valore che il loro riempitivo deve avere. Le combinazioni possibili per la costruzione dei CDC sono limitate dal tipo di dominio

Significato	Notazione DL	Sintassi *RACER*
Esistenza riempitivo appartenente al dominio concreto	$\exists$ AN.$\top_{\mathcal{D}}$	(a AN)
Non esistenza riempitivo appartenente al dominio concreto	$\forall$ AN.$\bot_{\mathcal{D}}$	(no AN)
Restrizione numerica su un riempitivo di tipo intero con z$\in \mathbb{Z}$	$\exists$ AN.min_z	(min AN z)
	$\exists$ AN.max_z	(max AN z)
	$\exists$ AN.$=_z$	(equal AN z)
Restrizione numerica su un riempitivo di tipo reale con P$\in \{<, <=, >, >=, =\}$ e r$\in \mathbb{R}$	$\exists$ AN.P_r	(P AN r)

Tabella 3.9: Restrizioni sui domini concreti.

concreto considerato e devono seguire le regole indicate in tabella 3.10, tenendo conto delle limitazioni, imposte per le espressioni di attributi, mostrate nella tabella 3.11. Per la dichiarazione delle concrete features si rimanda alla sezione 3.3.3.

3.3.3 La costruzione della TBox

In RACER la descrizione intensionale include la modellazione dei concetti e dei ruoli; tale modellazione è basata sui tre insiemi disgiunti: $\mathcal{C}$, $\mathcal{R}$ e $\mathcal{A}$, dei quali l'ultimo è necessario per poter utilizzare i domini concreti. Il linguaggio prevede che la descrizione di una TBox inizi con il comando:

(in-tbox <nomeTBox>)

a cui devono seguire tutti gli assiomi dei concetti, dei ruoli e delle concrete features riguardanti la TBox stessa.

CDC	⟶	(min AN integer)	|
		(max AN integer)	|
		(equal AN integer)	|
		(equal AN AN)	|
		(divisible AN cardinal)	|
		(not-divisible AN cardinal)	|
		(> aexpr aexpr)	|
		(>= aexpr aexpr)	|
		(< aexpr aexpr)	|
		(<= aexpr aexpr)	|
		(= aexpr aexpr)	|
		(<> aexpr aexpr)	|
		(string= AN string)	|
		(string= AN AN)	|
		(string<> AN string)	|
		(string<> AN AN)	
string	⟶	|stringa|	
aexpr	⟶	AN	|
		real	|
		(+ aexpr1 aexpr1)	|
		aexpr1	

Tabella 3.10: Regole per la costruzione di concetti basati su domini concreti.

aexpr1	$\longrightarrow$	aexpr2	|	
		aexpr3	|	
		aexpr5		
aexpr2	$\longrightarrow$	real	|	
		AN	|	(AN di tipo reale o complesso)
		(* real AN)		(AN di tipo reale o complesso)
aexpr3	$\longrightarrow$	real	|	
		AN	|	(AN di tipo complesso)
		(* integer aexpr4 aexpr4)		
aexpr4	$\longrightarrow$	AN	|	(AN di tipo complesso)
		(expt AN n)		(AN di tipo complesso)
aexpr5	$\longrightarrow$	integer	|	
		AN	|	(AN di tipo intero)
		(* integer AN)		(AN di tipo intero)

Tabella 3.11: Regole per la costruzione di espressioni di attributi.

Concetti atomici e assiomi di concetto

Per introdurre un nuovo concetto atomico non è necessaria una dichiarazione esplicita, ma è sufficiente che ne sia specificato il nome in un assioma qualsiasi. In RACER gli assiomi di concetto possono essere ciclici ed inoltre la TBox può contenere più assiomi per ciascun concetto. RLI fornisce i costrutti che permettono di specificare assiomi di inclusione, equivalenza e disgiunzione.

- *Assioma di inclusione tra concetti*

  Dichiara la sussunzione tra due descrizioni di concetto.

Notazione DL	Sintassi RACER
$C_1 \sqsubseteq C_1$	(implies C_1 C_2)

  Qualora $C_1 \longrightarrow$CN, la funzione "implies" esprime un assioma di specializzazione.

- *Assioma di equivalenza tra concetti*

  Dichiara l'equivalenza tra due descrizioni di concetto.

Notazione DL	Sintassi RACER
$C_1 \equiv C_2$	(equivalent C_1 C_2)

  Qualora $C_1 \longrightarrow$CN, la funzione "equivalent" esprime un assioma di equivalenza.

- *Assioma di disgiunzione tra concetti*

  Dichiara che due, o più, descrizioni di concetto sono tra loro disgiunte.

Notazione DL	Sintassi RACER
$C_1 \sqsubseteq \neg(C_2 \sqcup C_3 \sqcup \ldots \sqcap C_n)$	
$C_2 \sqsubseteq \neg(C_3 \sqcup C_4 \sqcup \ldots \sqcap C_n)$	(disjoint $C_1 \ldots C_n$)
...	
$C_{n-1} \sqsubseteq \neg(C_n)$	

- *Assioma di specializzazione tra concetti*

  Per ragioni storiche e per garantire una maggiore compatibilità con le specifiche KRSS, il linguaggio fornisce un'ulteriore macro funzione, che permette di dichiarare la relazione di sussunzione tra un concetto atomico e la descrizione di un concetto.

Notazione DL	Sintassi RACER
CN $\sqsubseteq$ C	(define-primitive-concept CN C)

- *Assioma di definizione tra concetti*

  Per gli stessi motivi cui sopra, è possibile utilizzare la macro funzione "define-concept" che dichiara la relazione di equivalenza tra un concetto atomico e la descrizione di un concetto.

Notazione DL	Sintassi RACER
CN $\equiv$ C	(define-concept CN C)

Ruoli e assiomi di ruolo

Per evitare possibili errori d'interpretazione semantica da parte del sistema, i nomi di ruolo devono essere dichiarati esplicitamente. Dato che, per RACER, gli insiemi $\mathcal{F}$ e $\mathcal{R}^+$ sono contenuti in $\mathcal{R}$, RLI mette a disposizione un'unica funzione, "define-primitive-role", tramite la quale è possibile dichiarare dei ruoli:

- Semplici: (define-primitive-role RN);
- Transitivi: (define-primitive-role RN :transitive t);
- Abstract Features: (define-primitive-role RN :features t);

Il sistema non permette la dichiarazione di abstract features transitive, inquanto considera gli insiemi $\mathcal{F}$ e $\mathcal{R}^+$ tra loro disgiunti. Per esprimere delle restrizioni sul dominio, o sul codominio, dei ruoli, oppure definire delle gerarchie, è possibile specificare i parametri: ":domain", ":range", ":parents", così come mostrato nella tabella: A differenza di quanto

Significato	Notazione DL	Sintassi RACER
Assioma di inclusione tra ruoli	$RN_1 \sqsubseteq RN_2 \wedge \ldots \wedge RN_1 \sqsubseteq RN_n$	(define-primitive-role RN_1 :parents ($RN_2 \ldots RN_n$))
Definizione di un ruolo RN sul dominio C	$\exists$ RN.$\top \sqsubseteq$ C	(define-primitive-role RN :domain C)
Definizione di un ruolo RN sul codominio D	$\top \sqsubseteq \forall$ RN.D	(define-primitive-role RN :range D)

Tabella 3.12: Dichiarazione delle restrizioni per i ruoli.

previsto per i concetti, non è permesso, nella TBox, definire più di un assioma per ciascun ruolo. Qualora siano definiti due assiomi per lo stesso ruolo, il sistema considera valido solo il secondo dei due.

- *Assioma di sussunzione tra ruoli*
  Dichiara la relazione di sussunzione tra due ruoli.

Notazione DL	Sintassi RACER
$RN_1 \sqsubseteq RN_2$	(implies-role RN_1 RN_2)

- *Assioma di equivalenza tra ruoli*
  Dichiara la relazione di equivalenza tra due ruoli.

Notazione DL	Sintassi RACER
$RN_1 \equiv RN_2$	(roles-equivalent RN_1 RN_2)

É da notare che la dichiarazione di un ruolo rappresenta già di per sè un assioma. Questo significa che, per evitare possibili problemi, è meglio esprimere gli assiomi di sussunzione tra i ruoli direttamente nella loro dichiarazione, tramite l'opzione ":parents", evitando l'utilizzo delle funzioni "implies-role" e "roles-equivalent" mostrate sopra.

Per la dichiarazione delle abstract features il linguaggio mette a disposizione anche la macro:

(define-primitive-attribute RN).

Concrete Features

In ragione del fatto che i domini concreti sono tra loro disgiunti, le concrete features sono considerate “fortemente tipati” e , prima di essere utilizzate nella definizione di un CDC, devono essere dichiarate, specificando il loro nome e il dominio concreto di appartenza del loro riempitivo. La funzione:

(define-concrete-domain-attribute AN :types <*dominio*>)

in cui <*dominio*> è uno tra i tipi supportati da RACER: “integer”, “cardinal”, “real”, “complex” oppure “string”, aggiunge AN all’insieme $\mathcal{A}$. Per le specifiche imposte dal sistema, una concrete feature può essere utilizzata esclusivamente con gli operatori, e nei modi, previsti dal dominio concreto del suo riempitivo.

Esempio TBox

La seguente TBox, alla quale può essere dato il nome “family”:

$$
\begin{aligned}
Woman &\sqsubseteq Person \sqcap Female \\
Man &\sqsubseteq Person \sqcap \neg Woman \\
Mother &\equiv Woman \sqcap \exists hasChild.Person \\
GrandFather &\equiv Man \sqcap (\exists hasChild.(Person \sqcap \\
&\quad \exists hasChild.Person)) \\
Father &\equiv Man \sqcap \exists hasChild.Person \\
MotherWithoutDaughter &\equiv Mother \sqcap \forall hasChild.\neg Woman \\
Wife &\equiv Woman \sqcap \exists hasHusband.Man \\
Ederly &\equiv Person \sqcap (\exists hasage.\mathbf{min}_{50})
\end{aligned}
$$

$$Young \equiv Person \sqcap (\exists hasage.\mathbf{max}_{35})$$
$$Employee \sqsubseteq Person$$

corrisponde al seguente codice in RLI:

```
(IN-TBOX FAMILY)

(DEFINE-PRIMITIVE-ROLE HASCHILD)
(DEFINE-PRIMITIVE-ROLE HASHUSBAND)

(DEFINE-CONCRETE-DOMAIN-ATTRIBUTE HASAGE :TYPE INTEGER)

(IMPLIES WOMAN (AND PERSON FEMALE))
(IMPLIES MAN (AND PERSON (NOT WOMAN)))
(EQUIVALENT MOTHER (AND WOMAN
    (SOME HASCHILD PERSON)))
(EQUIVALENT GRANDFATHER (AND MAN
    (SOME HASCHILD
        (AND PERSON (SOME HASCHILD PERSON)))))
(EQUIVALENT FATHER (AND MAN
    (SOME HASCHILD PERSON)))
(EQUIVALENT MOTHERWITHOUTDAUGHTER (AND MOTHER
    (ALL HASCHILD (NOT WOMAN))))
(EQUIVALENT WIFE (AND WOMAN
    (SOME HASHUSBAND MAN)))
(EQUIVALENT EDERLY (AND PERSON
    (MIN HASAGE 50)))
(EQUIVALENT YOUNG (AND PERSON
    (MIN HASAGE 0)
    (MAX HASAGE 35)))
```

3.3.4 La costruzione dell' ABox

La parte estensionale deve cominciare con il costrutto:

(in-abox <nomeABox> <nomeTBox>),

dove <nomeTBox> è il nome della TBox alla quale l'ABox si riferisce. La costruzione dell'ABox, che si basa sugli insiemi $\mathcal{I}$ e $\mathcal{O}$, consiste in un insieme di asserzioni che possono essere formulate nei termini di asserzioni di concetto, di ruolo, di concrete feature o di predicato sulle concrete features, secondo quanto specificato in tabella 3.13. I nomi

Tipo Asserzione	Sintassi RACER	Soddisfacibile se
Asserzione di concetto	IN : C	$IN^{\mathcal{I}} \in C^{\mathcal{I}}$
Asserzione di ruolo	(IN_1, IN_2) : R	($IN_1{}^{\mathcal{I}}$, $IN_2{}^{\mathcal{I}}$) $\in R^{\mathcal{I}}$
Asserzione di concrete feature	(IN, ON) : AN	($IN^{\mathcal{I}}$, α(ON)) $\in AN^{\mathcal{I}}$
Asserzione di predicato	($ON_1 \ldots ON_n$) : P[1]	(α($ON_1 \ldots ON_n$)) $\in P^{\mathcal{I}}$

[1] P $\in$ {min, max, equal, divisible, not-divisible, =, <, >, <=, >=, <>, string=, string<>}

Tabella 3.13: Asserzioni esprimibili in RACER

degli individui e degli oggetti non devono essere dichiarati preventivamente, inquanto il sistema li ricava direttamente dagli assiomi e, se non sono già presenti, li aggiunge automaticamente nei rispettivi insiemi. É bene chiarire che RACER considera "oggetti" i riempitivi delle concrete features, quindi è da notare che ad essi non è applicabile l'UNA, dato che più oggetti possono riferirsi allo stesso elemento del dominio, per esempio allo stesso numero.

Asserzioni di concetto

Le asserzioni di concetto dichiarano che un'individuo IN è istanza di un concetto C; è possibile fare più asserzioni per ciascun individuo e vengono formulate tramite il costrutto:

(INSTANCE IN C).

Ad esempio le asserzioni:

(INSTANCE KATTY WOMAN)
(INSTANCE GEORGE MAN)
(INSTANCE GEORGE EMPLOYEE)

dichiarano rispettivamente che *KATTY* è istanza del concetto *WOMAN* e che *GEORGE* è istanza del concetto *MAN* e del concetto *EMPLOYEE*. Lo stesso risultato riguardo a *GEORGE* si poteva ottenere con:

(INSTANCE GEORGE (AND MAN EMPLOYEE))

Asserzioni di ruolo

Tramite le asserzioni di ruolo si dichiara che un'individuo IN_2 è un riempitivo per il ruolo RN riguardante l'individuo IN_1. Il costrutto che permette di specificare un'asserzione di ruolo è:

(RELATED IN_1 IN_2 R)

Esempio:

(RELATED KATTY GEORGE HASHUSBAND)

indica che *GEORGE* è il riempitivo del ruolo *HASHUSBAND* relativo a *KATTY*.

Asserzioni di concrete feature

Dichiara che un oggetto OB è il riempitivo della concrete feature AN riguardante l'individuo IN.

(CONSTRAINED IN ON AN)

Esempio:

(CONSTRAINED GEORGE AGE_OF_GEORGE HASAGE),

indica che la concrete feature *HASAGE*, relativa all'individuo *GEORGE*, ha come riempitivo *AGE_OF_GEORGE*.
AGE_OF_GEORGE è un oggetto del tipo specificato da *HASAGE*, ovvero se *HASAGE* è una concrete feature sugli interi allora *AGE_OF_GEORGE* è un oggetto intero.

Asserzioni di predicato

Asseriscono l'insieme di predicati relativi ai domini concreti per gli oggetti dei domini concreti; in altre parole, dichiarano la relazione esistente tra un oggetto ON e gli elementi del dominio concreto. Vengono formulate mediante la funzione:

(CONSTRAINTS (<operatore> ON <valore>)),

dove <*operatore*> è uno degli operatori relazionali tra quelli ammessi per il tipo di oggetto, mentre <*valore*> è un elemento del dominio considerato.
Esempio:

(CONSTRAINTS (EQUAL AGE_OF_GEORGE 33)),

indica che il valore *AGE_OF_GEORGE* è uguale al numero intero 33.

Esempio ABox

Il seguente frammento di codice descrive la parte estensionale relativa all'esempio 3.3.3.

```
(IN-ABOX AFAMILY FAMILY)

(INSTANCE KATTY WOMAN)
(INSTANCE GEORGE (AND MAN EMPLOYEE))
(INSTANCE ANN WOMAN)
(INSTANCE SUE WOMAN)
(INSTANCE BILL MAN)

(RELATED KATTY GEORGE HASHUSBAND)
(RELATED KATTY ANN HASCHILD)
(RELATED KATTY BILL HASCHILD)
(RELATED GEORGE ANN HASCHILD)
(RELATED GEORGE BILL HASCHILD)
(RELATED SUE GEORGE HASCHILD)

(CONSTRAINED GEORGE AGE_OF_GEORGE HASAGE)
(CONSTRAINED ANN AGE_OF_ANN HASAGE)
(CONSTRAINED BILL AGE_OF_BILL HASAGE)

(CONSTRAINTS (EQUAL AGE_OF_GEORGE 33)
        (EQUAL AGE_OF_ANN 3)
        (EQUAL AGE_OF_BILL 1)
```

3.4 Analisi d’inconsistenza

RLI offre numerose funzioni in grado di rispondere a diverse tipologie di problemi riguardanti la consistenza della KB. Ne vengono riportate ora le più importanti, mentre per l’elenco completo si rimanda in appendice.

3.4.1 Controllo di consistenza sulla TBox

Le funzioni che verificano la soddisfacibilità della TBox sono:

- (check-tbox-coherence): ritorna la lista di tutti i concetti insoddisfacibili. Se il valore ritornato è uguale a nil significa che non esistono sinonimi del concetto incoerente e quindi la TBox è consistente.
- (concept-coherent?): controlla che non ci siano concetti atomici incoerenti. Ritorna T se il controllo ha esito positivo, nil altrimenti.
- (classify-tbox): classifica la TBox, creando la tassonomia dei concetti. É necessario che sia eseguito prima di rispondere alle query.

É inoltre possibile utilizzare le seguenti funzioni di base, che ammettono come parametri delle descrizioni di concetto:

- (concept-satisfable? C): ritorna T se il concetto C specificato è soddisfacibile, nil altrimenti.
- (concept-subsumes? C_1 C_2): con T indica che C_1 sussume C_2, mentre con nil indica che C_1 non sussume C_2.

Derivate dalle precedenti abbiamo:

- (concept-equivalent C_1 C_2): ritorna t se C_1 è equivalente C_2, nil se C_1 non è equivalente C_2.
- (concept-disjoint C_1 C_2): ritorna t se C_1 è digiunto C_2, nil se C_1 non è disgiunto C_2.

3.4.2 Controllo di consistenza sull' ABox

Le principali funzioni che permettono di verificare la consistenza dell'ABox sono:

- (realize-abox): verifica la consistenza dell'ABox e computa il più specifico dei concetti per ciascun individuo.

- (check-abox-coherence): controlla che l'ABox sia consistente; in caso affermativo ritorna t, altrimenti ritorna le informazioni riguardanti le istanze che rendono l'ABox inconsistente.

- (abox-consistent?): ritorna t se l'ABox è consistente, nil altrimenti.

3.5 Ritrattare o aggiungere informazioni

Dopo aver caricato in memoria una KB, il sistema permette ritrattare gli assiomi e le asserzioni. Questo significa che non è necessario ricaricare l'intera TBox e/o ABox per eliminare o aggiungere un assioma e/o un asserzione, ma è sufficiente utilizzare uno dei comandi supportati dal sistema. In particolare è possibile eliminare delle asserzioni o degli assiomi mediante il comando:

- (forget :abox <nomeABox> <lista_asserzioni_da_eliminare>): per eliminare dall'ABox <nomeABox> la lista di asserzioni specificata da <lista_asserzioni_da_eliminare>.

- (forget :tbox <nomeTBox> <lista_assiomi_da_eliminare>): per eliminare dalla TBox <nomeTBox> la lista di assiomi specificata da <lista_assiomi_da_eliminare>.

Invece, per aggiungere degli assiomi, o delle asserzioni, è sempre possibile utilizzare le funzioni già viste:

- (in-tbox <nomeTBox>);

- (in-abox <nomeABox>),

seguite dalla lista di assiomi, o asserzioni, che si vogliono aggiungere alla KB.

Ovviamente queste funzionalità hanno rilevanza solo in considerazione di un utilizzo del sistema in modalità server, tramite un'interfaccia che permetta la manipolazione delle informazioni della KB.

Capitolo 4

Logiche descrittive e database

4.1 Il formalismo *ODL*

Il formalismo *ODL*(Object Definition Logics)[BN94a, D.B94] estende l'espressività dei linguaggi sviluppati nell'area delle DLs, implementando un modello per basi di dati orientate agli oggetti che permette la modellazione di valori complessi e dell'ereditarietà multipla. In questa sezione vengono riportati gli aspetti principali del modello, cosi come presentati in [D.B94], a cui si rimanda per eventuali approffondimenti.

Si assume una ricca struttura per il sistema di tipi atomici o sistema di tipi base; oltre ai tipi atomici: integer, boolean, string, real e tipi monovalore, consideriamo anche la possibilità che siano usati dei sottoinsiemi dei tipi atomici, come ad esempio intervalli di interi. L'unica restrizione è che il sistema dei tipi atomici sia chiuso rispetto all'intersezione e che l'intersezione di due tipi base deve poter essere calcolata in tempo polinomiale.

Basandosi su questi tipi atomici possono essere create tuple, sequenze, insiemi e, in particolare, tipi oggetto. I tipi oggetto sono definiti tramite il costruttore Δ : tale costruttore applicato ad un generico tipo S definisce un insieme di oggetti con valore associato di tipo S.

L'ereditarietà sia semplice che multipla, è espressa ditìrettamente nella descrizione di una classe, cioè nel tipo associato alla classe, tramite l'operatore intersezione $\sqcap$.

Infine ai tipi può essere dato un nome: avremo quindi nomi di tipo valore e nomi di tipo classe. Un nome di tipo classe può essere base o virtuale a seconda del tipo di condizioni che stabilisce sugli oggetti che ne sono istanza. In particolare i nomi di tipi virtuali descrivono condizioni necessarie e sufficienti per l'appartenenza di un oggetto del dominio ad un tipo (esprimono quindi il concetto di "vista"), mentre i nomi di tipi primitivi descrivono condizioni necessarie di appartenenza (e che quindi si ricollegano alle classi di oggetti).

4.1.1 Sistema dei tipi atomici

Sia $\mathbf{D}$ l'insieme infinito numerabile dei valori atomici (che saranno indicati con $d_1, d_2, \ldots$), e.g., l'unione dell'insieme degli interi, delle stringhe e dei booleani. Non distingueremo fra i valori atomici e la loro codifica.

Sia $\mathbf{B}$ un insieme numerabile di designatori di tipi atomici (denotati da $B, B', \ldots$) che contiene $\mathbf{D}$ (i.e., tutti i tipi mono–valore), e sia $\mathcal{I}_{\mathbf{B}}$ la funzione di interpretazione standard (fissata) da $\mathbf{B}$ a $2^{\mathbf{D}}$ tale che per ogni $d \in \mathbf{D}$: $\mathcal{I}_{\mathbf{B}}[d] = \{d\}$. Sia "$\sqcap$" un'operazione di congiunzione su $\mathbf{B}$ definita da:

$$B' \sqcap B'' = B \text{ sse } \mathcal{I}_{\mathbf{B}}[B'] \cap \mathcal{I}_{\mathbf{B}}[B''] = \mathcal{I}_{\mathbf{B}}[B].$$

Diciamo che $\mathbf{B}$ è un *sistema di tipi atomici* sse $\mathbf{B}$ è completo rispetto a $\sqcap$. Il tipo speciale che ha un'interpretazione vuota è detto *tipo vuoto* ed è indicato con $\bot$.[1] Un sistema di tipi atomici $\mathbf{B}$ è detto *PTIME* sse $B' \sqcap B'' = B$ può essere deciso in tempo polinomiale. In seguito assumiamo che un sistema di tipi atomici abbia questa proprietà.

A volte parleremo anche di sistemi di tipi atomici con una particolare semplice struttura, ovvero, sistemi tali che per ogni sottoinsieme $\mathbf{X} \subseteq \mathbf{B}$ con $\bigsqcap \mathbf{X} = B$, ci sono due elementi $B', B'' \in \mathbf{X}$ tali che $B' \sqcap B'' = B$. Tale sistema di tipi atomici è detto *compatto binario*.

Consideriamo il seguente insieme di designatori di tipi atomici, che useremo in tutti

[1] Questo tipo deve appartenere a $\mathbf{B}$ perchè la congiunzione di differenti tipi mono–valore è vuota.

gli esempi:

$$\mathbf{B} = \{\texttt{Integer}, \texttt{String}, \texttt{Bool}, \texttt{Real}, i_1\text{–}j_1, i_2\text{–}j_2, \ldots, d_1, d_2, \ldots\},$$

dove gli i_k–j_k indicano tutti i possibili intervalli di interi e i d_k indicano tutti gli elementi di $\texttt{Integer} \cup \texttt{String} \cup \texttt{Bool}$ (i_k può essere $-\infty$ per denotare il minimo elemento di `Integer` e j_k può essere $+\infty$ per denotare il massimo elemento di `Integer`). Assumendo l'interpretazione standard dei designatori di tipi atomici, $\mathbf{B}$ è ovviamente un sistema di tipi atomici compatto binario.

4.1.2 Oggetti Complessi, Tipi e Classi

Sia $\mathbf{A}$ un insieme numerabile di *attributi* (denotati da $a_1, a_2, \ldots$) e $\mathbf{N}$ un insieme numerabile di *nomi di tipi* (denotati da $N, N', \ldots$) tali che $\mathbf{A}$, $\mathbf{B}$, e $\mathbf{N}$ siano a due a due disgiunti. $\mathbf{N}$ è partizionato in tre insiemi $\mathbf{C}$, $\mathbf{D}$ e $\mathbf{V}$, dove $\mathbf{C}$ consiste di nomi per *tipi-classe basi* ($C, C' \ldots$), $\mathbf{D}$ consiste di nomi per *tipi-classe virtuali* ($D, D' \ldots$), e $\mathbf{V}$ consiste di nomi per *tipi-valori* ($T, T', \ldots$).

Definizione 32 (Tipi) *Dati gli insiemi* $\mathbf{B}$, $\mathbf{A}$ *e* $\mathbf{N}$, *il* sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ *denota l'insieme di tutte le* descrizioni dei tipi *(*S, S', *...), detti anche brevemente* tipi, *su* $\mathbf{A}, \mathbf{B}, \mathbf{N}$, *che sono costruiti rispettando la seguente regola sintattica astratta (assumendo* $a_i \neq a_j$ *per* $i \neq j$*):*

S	$\rightarrow$	$\top$	
		B	*tipo atomico*
		N	*nome di tipo*
		$\{S\}$	*tipo insieme*
		$\langle S \rangle$	*tipo sequenza*
		$[a_1 \colon S_1, \ldots, a_k \colon S_k]$	*tipo tupla*
		$S \sqcap S'$	*intersezione*
		$\triangle S$	*tipo oggetto*

Sia $\mathcal{O}$ un insieme numerabile di *identificatori di oggetti*, (detti anche brevemente *oid*, e denotati da $o, o', \ldots$) disgiunto da $\mathbf{D}$.

Definizione 33 (Valori) *Dati gli insiemi $\mathcal{O}$ e $\mathbf{D}$, si definisce l'insieme $\mathcal{V}(\mathcal{O})$ dei* valori su *$\mathcal{O}$ (denotati da v, v') come segue (assumendo $p \geq 0$ e $a_i \neq a_j$ per $i \neq j$):*

v	$\rightarrow$	d	*valore atomico*
		o	*identificatore di oggetto*
		$\{v_1, \ldots, v_p\}$	*valore insieme*
		$\langle v_1, \ldots, v_p \rangle$	*valore sequenza*
		$[a_1 \colon v_1, \ldots, a_p \colon v_p]$	*valore tupla*

Definizione 34 (Dominio) *Dato un insieme di identificatori di oggetti $\mathcal{O}$, uno* dominio δ *su $\mathcal{O}$ è una funzione totale da $\mathcal{O}$ a $\mathcal{V}(\mathcal{O})$.*

Un dominio δ associa agli identificatori di oggetti un valore. In genere si dice che il valore $\delta(o)$ è lo *stato* dell'oggetto identificato dall'l'oid o; Un dominio verrà detto *finito* se l'insieme $\mathcal{O}$ è finito.

Lo stato di un oggetto può essere un generico valore, quindi un valore atomico, un valore strutturato o un altro oid. In alcuni modelli, quali O_2 [LR89b] e LOGIDATA$^+$[Atz93], lo stato è solo una tupla, ma questa non è una restrizione severa in quanto uno stato che è, ad esempio, un insieme può essere rappresentato come una tupla con un singolo componente insieme.

4.1.3 Schema di Base di Dati

Definizione 35 (Schema) *Dato un sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, uno* schema *$\sigma$ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ è una funzione totale da $\mathbf{N}$ a $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$.*

Uno schema σ associa ai nomi di tipi–classe e di tipi–valori la loro descrizione.

La possibilità di utilizzare un nome di tipo nella descrizione di un altro nome può far sorgere nello schema *descrizioni circolari*, cioè delle descrizioni di nome che fanno riferimento, direttamente o indirettamente tramite altri nomi, al nome stesso. In uno schema

solo alcuni tipi di descrizioni circolari sono ammesse; a tale scopo si definiscono alcune condizioni affinchè uno schema sia *ben–formato*.

Dato uno schema σ su $\mathbf{S}$, l'insieme delle espressioni di tipo *usate come sottoespressioni* nel range di σ è denotato con $\mathbf{S}_\sigma$. Sia $\mathbf{U}$ una relazione su $\mathbf{S}_\sigma \cup \mathbf{N}$ definita dalle seguenti regole ($C \in \mathbf{C}, D \in \mathbf{D}, T \in \mathbf{V}$) :

$$(C, S) \in \mathbf{U} \quad \text{sse} \quad \sigma(C) = S \qquad \text{(1.a)}$$

$$(D, S) \in \mathbf{U} \quad \text{sse} \quad \sigma(D) = S \qquad \text{(1.b)}$$

$$(T, S) \in \mathbf{U} \quad \text{sse} \quad \sigma(T) = S \qquad \text{(1.c)}$$

$$(S \sqcap S', S), (S \sqcap S', S'), \in \mathbf{U} \quad \text{sse} \quad S \sqcap S' \in \mathbf{S}_\sigma \qquad (2)$$

$$(\{S\}, S) \in \mathbf{U} \quad \text{sse} \quad \{S\} \in \mathbf{S}_\sigma \qquad (3)$$

$$(\langle S \rangle, S) \in \mathbf{U} \quad \text{sse} \quad \langle S \rangle \in \mathbf{S}_\sigma \qquad (4)$$

$$([\ldots, a_i : S_i, \ldots], S_i) \in \mathbf{U} \quad \text{sse} \quad [\ldots, a_i : S_i, \ldots] \in \mathbf{S}_\sigma \qquad (5)$$

$$(\triangle S, S) \in \mathbf{U} \quad \text{sse} \quad \triangle S \in \mathbf{S}_\sigma \qquad (6)$$

e sia $\mathbf{U}^+$ la chiusura transitiva della relazione $\mathbf{U}$.

Definizione 36 (Cicli) *Dato uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, la descrizione di un nome di tipo $N \in \mathbf{N}$ è detta* circolare *o, più brevemente, il nome di tipo N è detto* ciclico*, sse* $(N, N) \in \mathbf{U}^+$.

Un nome di tipo ciclico rappresenta una definizione di tipo *ricorsiva*. Nelle basi di dati deduttive [MN83] si parla in genere di *cicli tra regole*; le *regole ricorsive* sono un caso particolare di ciclo nel quale è coinvolto una singola regola.

La prima condizione richiesta per uno schema ben–formato garantisce che i nomi di tipi–valori descrivano sempre valori con un annidamento finito. Formalmente, sia $\mathbf{U}_1$ la chiusura transitiva della relazione ottenuta con le regole da (1.c) a (5) comprese. Uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ è *ben–formato sui nomi di tipi-valori* sse per ogni $T \in \mathbf{V}$, $(T, T) \notin \mathbf{U}_1$.

In questo modo, uno schema ben–formato sui nomi di tipi-valori ammette nomi di tipi-valori ciclici, purchè nel ciclo sia coinvolto almeno un costruttore di tipo oggetto; ad esempio, uno schema contenente un $T_1 \in \mathbf{V}$ con la descrizione: $\sigma(T_1) = [a_1 \colon \triangle[a_2 \colon T_1]]$

è uno schema ben–formato sui nomi di tipi-valori, al contrario di uno schema contenente un $T_2 \in \mathbf{V}$ con: $\sigma(T_1) = [a_1 \colon [a_2 \colon T_2]]$.

Un'importante caratteristica del modello proposto è il modo con il quale viene dichiarata la relazione *isa* tra le classi. L'*ereditarietà*, semplice e multipla, è espressa nella descrizione del nome della classe tramite l'operatore di intersezione. Per generalità, si estende la definizione a tutti i nomi di tipo.

Definizione 37 (Ereditarietà) *Dato uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, sia $\mathbf{U}_2$ la chiusura transitiva della relazione ottenuta con le regole da (1.a) a (2) comprese. Dati due nomi di tipo $N, N' \in \mathbf{N}$, diremo che N* eredita da *N', denotato con $N \prec_\sigma N'$, sse $(N, N') \in \mathbf{U}_2$.*

La chiusura riflessiva di $\prec_\sigma$ sarà denotata con $\preceq_\sigma$.

La seconda condizione richiesta per uno schema ben–formato garantisce che la relazione *isa* non contenga cicli. Uno schema σ è *ben–formato sull'ereditarietà* sse $\prec_\sigma$ è un ordine parziale stretto.

Uno schema σ è *ben–formato* sse è ben–formato sui nomi di tipi-valori e ben–formato sull'ereditarietà. Nel seguito si assume sempre che lo schema sia ben–formato.

Esempio di Schema

L'esempio descrive una base di dati riguardante una parte della struttura organizzativa di una società.

Le persone hanno un nome; i dipendenti sono persone che lavorano in un'azienda, percepiscono un salario e sono inquadrate in un livello. I dirigenti sono persone che lavorano e dirigono un'azienda, percepiscono un salario e sono inquadrati in livelli più alti. I settori hanno un nome e un insieme di attività; le aziende hanno un nome. Gli impiegati sono dipendenti che lavorano in un dipartimento; i dipartimenti, a loro volta, sono aziende che impiegano esclusivamente impiegati. I segretari sono dipendenti che lavorano in un ufficio; gli uffici sono aziende e settori che impiegano esclusivamente segretari e hanno un segretario che ricopre il ruolo di dattilografo.

$$
\begin{aligned}
\mathbf{V} &= \{\texttt{Activities}, \texttt{Level}, \texttt{AdvLevel}\} \\
\mathbf{C} &= \{\texttt{Person}, \texttt{Branch}\} \\
\mathbf{D} &= \{\texttt{Manager}, \texttt{Clerk}, \texttt{Department}, \texttt{Sector}, \\
&\quad \texttt{Employee}, \texttt{Secretary}, \texttt{Office}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma(\texttt{Level}) &= 1\text{–}10 \\
\sigma(\texttt{AdvLevel}) &= 8\text{–}10 \\
\sigma(\texttt{Activities}) &= \{\texttt{String}\} \\
\sigma(\texttt{Person}) &= \triangle\Big[\texttt{name}\colon \texttt{String}\Big] \\
\sigma(\texttt{Branch}) &= \triangle\Big[\texttt{name}\colon [\texttt{bname}\colon \texttt{String}]\Big] \\
\sigma(\texttt{Sector}) &= \triangle\Big[\texttt{name}\colon [\texttt{sname}\colon \texttt{String}], \texttt{activity}\colon \texttt{Activities}\Big] \\
\sigma(\texttt{Employee}) &= \texttt{Person} \sqcap \triangle\Big[\texttt{salary}\colon \texttt{Real}, \texttt{worksin}\colon \texttt{Branch}, \\
&\qquad \texttt{level}\colon \texttt{Level}\Big] \\
\sigma(\texttt{Manager}) &= \texttt{Person} \sqcap \triangle\Big[\texttt{salary}\colon \texttt{Real}, \texttt{worksin}\colon \texttt{Branch}, \\
&\qquad \texttt{head}\colon \texttt{Branch}, \texttt{level}\colon \texttt{AdvLevel}\Big] \\
\sigma(\texttt{Clerk}) &= \texttt{Employee} \sqcap \triangle\Big[\texttt{worksin}\colon \texttt{Department}\Big] \\
\sigma(\texttt{Department}) &= \texttt{Branch} \sqcap \triangle\Big[\texttt{employs}\colon \{\texttt{Clerk}\}\Big] \\
\sigma(\texttt{Secretary}) &= \texttt{Employee} \sqcap \triangle\Big[\texttt{worksin}\colon \texttt{Office}\Big] \\
\sigma(\texttt{Office}) &= \texttt{Branch} \sqcap \texttt{Sector} \sqcap \triangle\Big[\texttt{employs}\colon \{\texttt{Secretary}\}, \\
&\qquad \texttt{typing}\colon \texttt{Secretary}\Big]
\end{aligned}
$$

Tabella 4.1: Schema corrispondente alla struttura organizzativa di una società

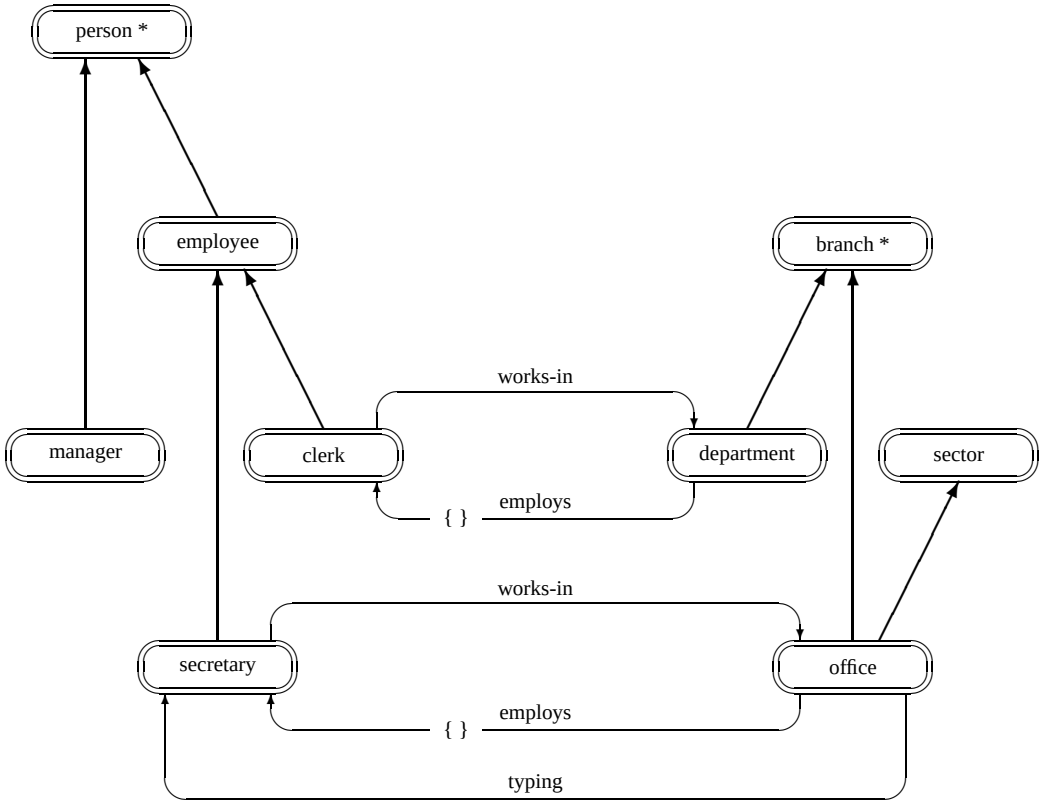

Figura 4.1: Gerarchia delle classi relativa alla struttura organizzativa di una società

Uno schema corrispondente a queste descrizioni è mostrato in tabella 4.1, dove si è assunto che tutte le classi, ad eccezione delle persone e delle aziende, siano classi virtuali. Si può facilmente verificare che questo schema è ben–formato.

La figura 4.1 mostra la relativa gerarchia delle classi; le classi base sono contrassegnate con un asterisco. La figura riporta anche gli attributi che danno origine alle classi virtuali cicliche `Clerk`, `Department`, `Secretary` e `Office`; essi sono indicati con archi orientati con in aggiunta il simbolo $\{\}$ nel caso in cui gli attributi sono multi–valore.

Istanze di uno Schema

In questa sezione viene data una formale semantica al sistema dei tipi. L'approccio seguito è simile alla formale semantica *set–theoretic* per i tipi introdotta da Cardelli [Car84] e adottata nel modello dei dati O_2 [LR89a, LR89b]. Successivamente verranno specificati gli *stati* legali della base di dati tramite la nozione di *istanza possibile* e *istanza legale* di uno schema.

A tale scopo, si definisce il concetto di *interpretazione* come funzione che associa ad ogni tipo un insieme di valori.

Definizione 38 (Interpretazione) *Dato un sistema di tipi* $\mathbf{S}$ *e un dominio* δ*, l'*interpretazione $\mathcal{I}$ *di* $\mathbf{S}$ *su* δ*, è una funzione da* $\mathbf{S}$ *a* $2^{\mathcal{V}(\mathcal{O})}$ *tale che:*

$$
\begin{aligned}
\mathcal{I}[\top] &= \mathcal{V}(\mathcal{O}) \\
\mathcal{I}[B] &= \mathcal{I}_{\mathbf{B}}[B] \\
\mathcal{I}[C] &\subseteq \mathcal{O} \\
\mathcal{I}[D] &\subseteq \mathcal{O} \\
\mathcal{I}[T] &\subseteq \mathcal{V}(\mathcal{O}) - \mathcal{O} \\
\mathcal{I}[\{S\}] &= \left\{ \{v_1, \ldots, v_p\} \middle| v_i \in \mathcal{I}[S], 0 \leq i \leq p \right\} \\
\mathcal{I}[\langle S \rangle] &= \left\{ \langle v_1, \ldots, v_p \rangle \middle| v_i \in \mathcal{I}[S], 0 \leq i \leq p \right\} \\
\mathcal{I}\Big[[a_1\colon S_1, \ldots, a_p\colon S_p]\Big] &= \Big\{ [a_1\colon v_1, \ldots, a_q\colon v_q] \Big| \; p \leq q, v_i \in \mathcal{I}[S_i], 0 \leq i \leq p, \\
&\qquad\qquad v_j \in \mathcal{V}(\mathcal{O}), p+1 \leq j \leq q \Big\}
\end{aligned}
$$

$$\begin{aligned}\mathcal{I}[S \sqcap S'] &= \mathcal{I}[S] \cap \mathcal{I}[S'] \\ \mathcal{I}[\triangle S] &= \Big\{ o \in \mathcal{O} \Big| \delta(o) \in \mathcal{I}[S] \Big\}.\end{aligned}$$

Prima di tutto si noti che per i tipi tupla si adotta una semantica di mondo aperto simile a quella in [Car84]. Ad esempio:

$$\begin{aligned}\Big[\texttt{name}\colon \text{“}\textit{Mark}, \texttt{salary}\colon 8000, \texttt{level}\colon 3\Big] &\in \mathcal{I}\Big[[\texttt{name}\colon \texttt{String}]\Big] \\ \Big[\texttt{name}\colon [\texttt{bname}\colon \text{“}\textit{Research}, \texttt{sname}\colon \text{“}\textit{DB}]\Big] &\in \mathcal{I}\Big[[\texttt{name}\colon [\texttt{bname}\colon \texttt{String}]\Big]\end{aligned}$$

Tramite la nozione di interpretazione sopra definita non si impone che un valore istanziato in un nome di tipo abbia una descrizione corrispondente a quella del nome di tipo stesso. Per i nomi di tipo, la funzione interpretazione si limita a vincolare i nomi delle classi ad un insieme di oid e i nomi di tipo–valore ad un insieme di valori che non siano oid.

Definizione 39 (Istanza Possibile) *Dato uno schema σ su $\mathbf{S}$ e un dominio δ, un interpretazione $\mathcal{I}$ di $\mathbf{S}$ su δ, è detta* istanza possibile *di σ su δ sse δ è* finito *e per ogni $C \in \mathbf{C}, D \in \mathbf{D}, T \in \mathbf{V}$:*

$$\begin{aligned}\mathcal{I}[C] &\subseteq \mathcal{I}[\sigma(C)] \\ \mathcal{I}[T] &= \mathcal{I}[\sigma(T)] \\ \mathcal{I}[D] &= \mathcal{I}[\sigma(D)].\end{aligned}$$

Con la nozione di istanza possibile di uno schema σ si impone che gli oggetti nell'interpretazione di una classe base C siano un sottoinsieme degli oggetti nell'interpretazione della descrizione della classe $\sigma(C)$. Questo significa che gli oggetti *istanziati* in una classe base sono esplicitamente forniti dall'utente e soddisfano la descrizione della classe stessa. Invece, per i nomi di classi virtuali e per i nomi di tipo–valore l'interpretazione è uquale all'interpretazione della descrizione del nome, cioè per tali nomi N l'interpretazione è calcolata sulla base della loro descrizione $\sigma(N)$.

In altri termini, la descrizione $\sigma(N)$ costituisce un insieme di condizioni *necessarie e sufficienti* per i nomi di tipo $N \in \mathbf{D} \cup \mathbf{V}$, mentre costituisce un insieme di condizioni *solamente necessarie* per i nomi di tipo $N \in \mathbf{C}$.

Per concludere questa sezione, vediamo quali siano le conseguenze, a livello di istanze dello schema, di esprimere l'ereditarietà nella descrizione dei nomi di tipo tramite l'operatore di intersezione. Con riferimento allo schema specificato in tabella 4.1, la classe `Office` "eredita" da `Sector` e `Branch` in quanto tali nomi di classe sono congiunti nella descrizione di `Office`, i.e., i valori associati agli oggetti di `Office` devono soddisfare le restrizioni espresse in `Sector` e `Branch`. Questo significa che le restrizioni sull'attributo `name`, che è definito sia in `Sector` che in `Branch`, sono le seguenti:

$$\texttt{name}\colon [\texttt{bname}\colon \texttt{String}] \sqcap [\texttt{sname}\colon \texttt{String}]$$

cioè le restrizioni sull'attributo `name` sono semplicemente combinate congiuntivamente. In altri termini, non è necessario "risolvere i conflitti di ereditarietà" sugli attributi che ereditano un differente tipo da più superclassi come avviene per i linguaggi di definizione degli schemi in O_2 [LR89b] e in LOGIDATA$^+$[Atz93], ma il tipo di un attributo è l'intersezione dei rispettivi tipi in tutte le superclassi. Nello stesso modo, se un attributo è sia ereditato da una o più superclassi sia definito esplicitamente nella classe, il suo tipo è l'intersezione dei rispettivi tipi in tutte le superclassi e del tipo esplicitato nella classe in questione.

4.1.4 Semantica dei nomi ciclici

La nozione di istanza possibile di uno schema impone che l'interpretazione dei nomi di classi virtuali e di tipo–valore coincida con quella della loro descrizione; tale semantica non è immediata in presenza di nomi di tipo ciclici. Per i nomi di classe base il problema non sussiste in quanto l'interpretazione è data esplicitamente.

L'esempio che segue ha lo scopo di evidenziare il problema. La tabella 4.2 riporta un dominio δ relativo allo schema σ di tabella 4.1. La tabella 4.3 mostra alcune istanze possibili dello schema σ che condividono la stessa interpretazione per le classi base `Person` e `Branch` ma con diversa interpretazione per le classi virtuali cicliche. Invece, per le classi virtuali non cicliche (`Employee`, `Manager` e `Sector`) l'interpretazione è la stessa

$$
\begin{aligned}
\mathcal{O} &= \{o_1, o_2, o_3, o_4, o_5, o_6, o_7, o_8, o_9, o_{10}\} \\
\delta(o_1) &= [\texttt{name}: [\texttt{bname}: \text{“}\textit{Administration}], \texttt{employs}: \{o_2, o_3, o_4\}] \\
\delta(o_2) &= [\texttt{name}: \text{“}\textit{Mark}, \texttt{salary}: 8000, \texttt{worksin}: o_1, \texttt{level}: 3] \\
\delta(o_3) &= [\texttt{name}: \text{“}\textit{Andrew}, \texttt{salary}: 8700, \texttt{worksin}: o_1, \texttt{level}: 4] \\
\delta(o_4) &= [\texttt{name}: \text{“}\textit{Andy}, \texttt{salary}: 9000, \texttt{worksin}: o_1, \texttt{head}: o_1, \texttt{level}: 8] \\
\delta(o_5) &= [\texttt{name}: [\texttt{bname}: \text{“}\textit{Development}], \texttt{employs}: \{\}] \\
\delta(o_6) &= [\texttt{name}: \text{“}\textit{Robert}, \texttt{salary}: 9000, \texttt{worksin}: o_5, \texttt{level}: 3] \\
\delta(o_7) &= [\texttt{name}: [\texttt{bname}: \text{“}\textit{Research}, \texttt{sname}: \text{“}\textit{DB}], \texttt{typing}: o_9 \\
&\quad\ \texttt{activity}: \{\text{“}\textit{DB, UserInterfaces}\}, \texttt{employs}: \{o_8, o_9, o_{10}\}] \\
\delta(o_8) &= [\texttt{name}: \text{“}\textit{Peter}, \texttt{salary}: 8500, \texttt{worksin}: o_7, \texttt{level}: 4] \\
\delta(o_9) &= [\texttt{name}: \text{“}\textit{James}, \texttt{salary}: 7500, \texttt{worksin}: o_7, \texttt{level}: 5] \\
\delta(o_{10}) &= [\texttt{name}: \text{“}\textit{Jon}, \texttt{salary}: 14000, \texttt{worksin}: o_7, \texttt{head}: o_7, \texttt{level}: 8]
\end{aligned}
$$

Tabella 4.2: Un esempio di dominio per lo schema

in tutte le istanze possibili mostrate. In altre parole, per questi nomi l'interpretazione è univocamente determinabile a partire da quella dei nomi di classe base e dei tipi atomici.

Si possono delineare due modi di procedere: selezionare come istanza *legale* una *particolare* istanza possibile oppure accettare come istanza legale una *generica* istanza possibile.

Il primo approccio può essere realizzato adottando una *semantica di punto fisso*, cioè determinando le estensioni dei nomi di tipo ciclici tramite opportuni operatori di punto fisso. Questo è l'approccio seguito nella semantica dei Linguaggi di Programmazione in genere e nella semantica dei Linguaggi per Basi di Dati che ammettono la ricorsione.

Nel classico modello di dati basato sulla logica, il DATALOG, si adotta la semantica di minimo punto fisso; un insieme di regole ricorsive ha sempre un "minimo punto fisso" che, proprio come nel caso non ricorsivo, coincide con l'insieme dei fatti derivabili dalla base di dati facendo uso delle regole [Ull89].

	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_3$
`Person`	$\{o_2, o_3, o_4, o_6, o_8, o_9, o_{10}\}$	$\{o_2, o_3, o_4, o_6, o_8, o_9, o_{10}\}$	$\{o_2, o_3, o_4, o_6, o_8, o_9, o_{10}\}$
`Branch`	$\{o_1, o_5, o_7\}$	$\{o_1, o_5, o_7\}$	$\{o_1, o_5, o_7\}$
`Employee`	$\{o_2, o_3, o_4, o_6, o_8, o_9, o_{10}\}$	$\{o_2, o_3, o_4, o_6, o_8, o_9, o_{10}\}$	$\{o_2, o_3, o_4, o_6, o_8, o_9, o_{10}\}$
`Manager`	$\{o_4, o_{10}\}$	$\{o_4, o_{10}\}$	$\{o_4, o_{10}\}$
`Sector`	$\{o_7\}$	$\{o_7\}$	$\{o_7\}$
`Clerk`	$\{o_2, o_3, o_4, o_6, o_8, o_9, o_{10}\}$	$\{o_2, o_3, o_4, o_6\}$	$\{o_6\}$
`Department`	$\{o_1, o_5, o_7\}$	$\{o_1, o_5\}$	$\{o_5\}$
`Secretary`	$\{o_8, o_9, o_{10}\}$	$\{o_8, o_9, o_{10}\}$	$\{\}$
`Office`	$\{o_7\}$	$\{o_7\}$	$\{\}$

Tabella 4.3: Alcune istanze possibili dello schema

Nella maggior parte dei sistemi per la rappresentazione della conoscenza derivati dal modello KL-ONE le descrizioni circolari sono proibite per un duplice motivo: non è ovvio come definire la semantica di un nome ciclico e, anche fissata una certa semantica, non è facile ottenere i corrispondenti algoritmi di inferenza. D'altra parte negli ultimi anni le Logiche Descrittive contenenti cicli sono state investigate a fondo [Baa90, Neb90a, Neb91, Baa91].

Come notato in [AK89, BN94b], per i modelli di dati ad oggetti la semantica del minimo punto fisso non può costituire l'unica scelta possibile, in quanto la minima interpretazione per certi nomi ciclici è necessariamente l'insieme vuoto. Intuitivamente, questo accade quando la descrizione circolare è costruita senza nessun costruttore di insieme o di sequenza, come ad esempio per la classe `Secretary`. Quindi, oltre alla "classica" semantica di minimo punto fisso, anche altre semantiche, quale quella del massimo punto fisso, devono essere prese in considerazione.

La semantica di punto fisso, selezionando una precisa istanza, stabilisce che, come per i nomi non ciclici, l'interpretazione dei nomi virtuali ciclici sia univocamente determinabile a partire da quella dei nomi di classe base e dei tipi atomici. Con il secondo approccio, che chiameremo *semantica descrittiva*, questo *vincolo* viene rilasciato e si accetta come istanza legale una qualsiasi delle istanze possibili.

4.1.5 Semantica di punto fisso

Nella prossima sottosezione vengono brevemente riportate alcune nozioni sulla teoria dei reticoli e dei punti fissi (si veda ad esempio [Llo87, Apt90]; inoltre, un breve riassunto è anche riportato in [Baa90, Neb90a, Neb91]).

Le istanze possibili di uno schema che condividono la stessa interpretazione per le classi base vengono caratterizzare come punti fissi di un certo operatore sulle interpetazioni. Questo permetterà di valutare le estensioni delle espressioni di tipo, e in particolare quella dei nomi ciclici, partendo da una fissata interpretazione per le classi base oltre che, ovviamente, da quella standard per i tipi atomici.

Allo scopo di fissare un'istanza possibile dello schema *rispetto alle sole classi base*, viene fatta, in un primo momento, l'ipotesi che la descrizione di una classe base non faccia riferimento a nomi virtuali; successivamente si vedrà come eliminare o sostituire questa ipotesi con una meno restrittiva.

Formalmente, si considera il sottoinsieme dei tipi di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ costruiti a partire solo da $\mathbf{C}$, cioè $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{C})$ e si assume che, per ogni $C \in \mathbf{C}$, $\sigma(C)$ sia un'espressione di tipo di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{C})$. Sotto questa ipotesi, ad uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ si può associare uno schema $\breve{\sigma}$ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{C})$, tale che $\breve{\sigma}(C) = \sigma(C)$, per ogni $C \in \mathbf{C}$. Un'istanza possibile $\breve{\mathcal{I}}$ di $\breve{\sigma}$ è detta *istanza parziale* dello schema σ; inoltre, si dice che un'istanza possibile $\mathcal{I}$ di σ *estende* l'istanza parziale $\breve{\mathcal{I}}$ di $\breve{\sigma}$ se $\mathcal{I}[C] = \breve{\mathcal{I}}[C]$, per ogni $C \in \mathbf{C}$.

Data un'istanza parziale $\breve{\mathcal{I}}$ dello schema σ sul dominio δ, sia $\Psi_{\breve{\mathcal{I}}}$ l'insieme delle interpretazioni di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ su δ tale che per ogni $\mathcal{I} \in \Psi_{\breve{\mathcal{I}}}$:

$$\mathcal{I}[C] = \breve{\mathcal{I}}[C] \text{ per ogni } C \in \mathbf{C}.$$

e sia $\trianglelefteq$ una relazione su $\Psi_{\breve{\mathcal{I}}}$ tale che per ogni $\mathcal{I}, \mathcal{I}' \in \Psi_{\breve{\mathcal{I}}}$:

$$\mathcal{I} \trianglelefteq \mathcal{I}' \quad \text{iff} \quad \mathcal{I}[N] \subseteq \mathcal{I}'[N] \text{ per ogni } N \in \mathbf{D} \cup \mathbf{V}.$$

Allora $\Psi_{\breve{\mathcal{I}}}$ ordinato in base a $\trianglelefteq$ è un *reticolo completo*. Il minimo elemento di $\Psi_{\breve{\mathcal{I}}}$ verrà denotato con $\mathcal{I}_\perp$ e il massimo elemento di $\Psi_{\breve{\mathcal{I}}}$ verrà denotato con $\mathcal{I}_\top$. Si noti che

$$\begin{array}{lcll} \mathcal{I}_\perp[N] & = & \emptyset & \text{per ogni } N \in \mathbf{D} \cup \mathbf{V} \\ \mathcal{I}_\top[N] & = & \mathcal{O} & \text{per ogni } N \in \mathbf{D} \\ \mathcal{I}_\top[N] & = & \mathcal{V}(\mathcal{O}) \setminus \mathcal{O} & \text{per ogni } N \in \mathbf{V} \end{array}$$

Sia Γ una funzione che mappa interpretationi in interpretationi dell'insieme $\Psi_{\breve{\mathcal{I}}}$:

$$\Gamma : \Psi_{\breve{\mathcal{I}}} \longrightarrow \Psi_{\breve{\mathcal{I}}}$$

tale che, per ogni $N \in \mathbf{N}$

$$\Gamma(\mathcal{I}[N]) = \mathcal{I}[\sigma(N)]$$

Un *punto fisso* di Γ, cioè una interpretatione $\mathcal{I}$ tale che $\Gamma(\mathcal{I}) = \mathcal{I}$, è un'istanza possibile di σ e viceversa.

Proposizione 5 *Dato uno schema σ e un'istanza parziale $\breve{\mathcal{I}}$ di σ su un dominio δ, un'interpretazione $\mathcal{I}$ è un'istanza possibile di σ che estende $\breve{\mathcal{I}}$ se e solo se $\mathcal{I}$ è un punto fisso della funzione Γ su $\Psi_{\breve{\mathcal{I}}}$.*

Pertanto, le istanze possibili di uno schema σ sono state caratterizzate come punti fissi di una funzione Γ su un reticolo completo $\Psi_{\breve{\mathcal{I}}}$. Introducendo le seguenti notazioni: $\Gamma^0(\mathcal{I}) = \mathcal{I}$ e $\Gamma^{i+1}(\mathcal{I}) = \Gamma(\Gamma^i(\mathcal{I}))$, e applicando i risultati richiamati nella precedente sottosezione si ottiene:

Teorema 2 *Dato uno schema σ e un'istanza parziale $\breve{\mathcal{I}}$ di σ su un dominio δ, l'insieme dei punti fissi di Γ forma un reticolo completo con*

$$\begin{aligned} lfp(\Gamma) &= \bigcup_{i \geq 0} \Gamma^i(\mathcal{I}_\perp) \\ gfp(\Gamma) &= \bigcap_{i \geq 0} \Gamma^i(\mathcal{I}_\top) \end{aligned}$$

A questo punto abbiamo tutti gli elementi per definire tre differenti semantiche per uno schema.

Definizione 40 (Istanza Legale) *Dato uno schema σ su $\mathbf{S}$, un dominio δ e un'istanza parziale $\breve{\mathcal{I}}$ di σ su δ, un interpretazione $\mathcal{I}$ di $\mathbf{S}$ su δ è detta*

- istanza *di σ che estende $\breve{\mathcal{I}}$ se $\mathcal{I}$ è un punto fisso di Γ* (semantica descrittiva)*;*
- istanza–lfp *di σ che estende $\breve{\mathcal{I}}$ se $\mathcal{I} = lfp(\Gamma)$* (semantica del minimo punto fisso)*;*
- istanza–gfp *di σ che estende $\breve{\mathcal{I}}$ se $\mathcal{I} = gfp(\Gamma)$* (semantica del massimo punto fisso).

In altre parole, con la semantica descrittiva si ammettono come istanze di uno schema tutte le istanze possibili; con la semantica del minimo punto fisso si ammettono solo quelle che sono il minimo punto fisso di una funzione Γ; con la semantica del massimo punto fisso si ammettono solo quelle che sono il massimo punto fisso di una funzione Γ.

Con riferimento agli oids e la funzione valore specificati in tabella 4.2, fissata come istanza parziale

$$\begin{aligned} \breve{\mathcal{I}}[\texttt{Person}] &= \{o_1, o_3, o_5, o_7\} \\ \breve{\mathcal{I}}[\texttt{Branch}] &= \{o_2, o_4, o_6\} \end{aligned}$$

l'istanza–*lfp* e l'istanza–*gfp* sono rispettivamente la $\mathcal{I}_3$ e la $\mathcal{I}_1$ mostrate in tabella 4.3, mentre la $\mathcal{I}_2$ è un'altra istanza possibile che estende la fissata istanza parziale.

All'inizio della presente sezione è stata fatta l'ipotesi che la descrizione di una classe base non fa riferimento a nomi virtuali; vogliamo analizzare se eliminando tale restrizione o sostituendola con una meno restrittiva quanto detto in precedenza resta valido. A tale scopo, estendiamo alcune definizioni precedenti.

Sia σ un generico schema su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ e $\overline{\mathcal{I}}$ una sua istanza possibile su un dominio δ; si dice che un'istanza possibile $\mathcal{I}$ sul dominio δ *estende* l'istanza $\overline{\mathcal{I}}$ se $\mathcal{I}[C] = \overline{\mathcal{I}}[C]$, per ogni $C \in \mathbf{C}$. Si cuuole individuare, tra le istanze su δ che estendono $\overline{\mathcal{I}}$, quella massima e quella minima; a tale scopo si considera l'insieme $\Psi_{\overline{\mathcal{I}}}$ delle interpretazioni di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ su δ tale che per ogni $\mathcal{I} \in \Psi_{\overline{\mathcal{I}}}$ e per ogni $C \in \mathbf{C}$, si ha che $\mathcal{I}[C] = \overline{\mathcal{I}}[C]$.

Prima di tutto, si individuano i nomi di tipo virtuali per i quali l'estensione è *univocamente determinabile* a partire da quella delle classi base e dei tipi atomici: essi sono i nomi aciclici o con una descrizione circolare che coinvolge *almeno* un nome di classe base. Formalmente, sia $\mathbf{U}_3$ la chiusura transitiva della relazione ottenuta con le regole di pagina 77 esclusa la regole (1.a); sia $\mathbf{N_c}$ il seguente insieme di nomi: $\mathbf{N_c} = \{N \in \mathbf{D} \cup \mathbf{V} \mid \exists N' \in \mathbf{D} \cup \mathbf{V}, (N, N') \in \mathbf{U}_3, (N', N') \in \mathbf{U}_3\}$.

Proposizione 6 *Dato uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, sia $\overline{\mathcal{I}}$ un'istanza possibile di σ su un dominio δ. Allora per ogni $N \in \mathbf{N} \setminus \mathbf{N_c}$, $\mathcal{I}[N] = \mathcal{I}'[N]$, per ogni istanza possibile $\mathcal{I}, \mathcal{I}'$ di σ su δ che estende $\overline{\mathcal{I}}$.*

In altri termini, in uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ con $\mathbf{N_c} = \emptyset$ l'istanza–*lfp* e l'istanza–*gfp* che estendono una fissata istanza possibile $\overline{\mathcal{I}}$ coincidono.

Consideriamo ora i nomi $N \in \mathbf{N_c}$. Intuitivamente, se fissiamo l'estensione di una classe base la cui descrizione contiene un nome virtuale N allora si impone, nell'istanza possibile dello schema, un *estremo inferiore* all'estensione che il nome N può assumere. Ad esempio, sia $C \in \mathbf{C}$ e $D \in \mathbf{D}$, con

$$\begin{aligned} \sigma(C) &= \triangle[a_1 \colon D] \\ \sigma(D) &= \triangle[a_2 \colon D] \end{aligned}$$

quindi, in particolare, $D \in \mathbf{N_c}$. Se viene fissato $\overline{\mathcal{I}}[C]$, allora l'estensione di D deve essere necessariamente più grande dell'insieme $\{o \in \mathcal{O} \mid \exists o' \in \overline{\mathcal{I}}[C], \delta(o') = [a_1 \colon o]\}$. La presenza di questo estremo inferiore comporta che nel calcolo dell'istanza di minimo punto fisso si debba partire da un certa interpretazione e non dall'interpretazione vuota; in altri termini, nella determinazione dell'istanza per la semantica del minimo punto fisso, l'interpretazione di N non può essere calcolata basandosi solo sulla sua descrizione (nell'esempio, la descrizione del nome D non contiene C, ma l'interpretazione di D dipende da quella di C). In questo modo, viene *violata l'idea* che l'interpretazione dei nomi virtuali sia determinabile basandosi esclusivamente sulla loro descrizione. Con la semantica del massimo punto fisso si seleziona l'istanza massima, quindi questo problema non sussiste.

Più precisamente, le istanze possibili dello schema σ non vanno ricercate nell'intero insieme $\Psi_{\overline{\mathcal{I}}}$ ma nel suo sottoinsieme

$$\{\mathcal{I} \in \Psi_{\overline{\mathcal{I}}} \mid \overline{\mathcal{I}}_{\perp} \trianglelefteq \mathcal{I}\}$$

dove il minimo elemento $\overline{\mathcal{I}}_{\perp}$ deve essere determinato sulla base di $\overline{\mathcal{I}}$ e quindi, in generale, non si ha più che $\overline{\mathcal{I}}_{\perp}[N] = \emptyset$, per ogni $N \in \mathbf{D} \cup \mathbf{V}$; il massimo elemento resta invece invariato rispetto a quello di $\Psi_{\check{\mathcal{I}}}$.

In definitiva, per l'istanza di massimo punto fisso, quanto asserito nel teorema 2 vale per un generico schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$. Per l'istanza di minimo punto fisso il teorema 2 resta valido, cioè il minimo punto fisso può essere determinato a partire dall'insieme vuoto, se si vietano i casi simili a quello visto sopra, imponendo che un nome ti tipo

virtuale $N \in \mathbf{N_c}$ non sia utilizzato nella descrizione di una classe base. Formalmente, dato uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, si deve imporre che per ogni $C \in \mathbf{C}$ non esiste $N \in \mathbf{N_c}$ tale che $(C, N) \in \mathbf{U}^+$.

4.1.6 Ereditarietà, Sussunzione e Coerenza

In questa sezione definiamo una relazione di inclusione semantica, detta *relazione di sussunzione*, tra i tipi in uno schema. La relazione di sussunzione, indicata con il simbolo $\sqsubseteq$, in uno schema σ viene definita rispetto alle tre differenti semantiche introdotte nella sezione precedente.

Definizione 41 (Sussunzione) *Dato uno schema σ su $\mathbf{S}$ e due tipi $S, S' \in \mathbf{S}$, si definiscono le relazioni di sussunzione $\sqsubseteq^d_\sigma$, $\sqsubseteq^l_\sigma$ e $\sqsubseteq^g_\sigma$ come segue:*

$$S \sqsubseteq^d_\sigma S' \quad \textit{sse} \quad \mathcal{I}[S] \subseteq \mathcal{I}[S'] \quad \textit{per ogni istanza } \mathcal{I} \textit{ di } \sigma;$$

$$S \sqsubseteq^l_\sigma S' \quad \textit{sse} \quad \mathcal{I}[S] \subseteq \mathcal{I}[S'] \quad \textit{per ogni} \text{ istanza–lfp } \mathcal{I} \textit{ di } \sigma;$$

$$S \sqsubseteq^g_\sigma S' \quad \textit{sse} \quad \mathcal{I}[S] \subseteq \mathcal{I}[S'] \quad \textit{per ogni} \text{ istanza–gfp } \mathcal{I} \textit{ di } \sigma.$$

Nel seguito si denota con $\sqsubseteq_\sigma$ una delle relazioni di sussunzione definite sopra allo scopo di esprimerne proprietà comuni nelle tre semantiche. La relazione $\sqsubseteq_\sigma$ è un preordine (cioè transitivo e riflessivo ma antisimmetrico) che induce una relazione di *equivalenza* $\simeq_\sigma$ sui tipi: $S \simeq_\sigma S'$ sse $S \sqsubseteq_\sigma S'$ e $S' \sqsubseteq_\sigma S$. La relazione di equivalenza $\simeq_\sigma$ permette di definire i tipi S che hanno, nello schema σ, un'interpretazione sempre vuota.

Definizione 42 (Incoerenza) *Dato uno schema σ su $\mathbf{S}$, un tipo $S \in \mathbf{S}$ è detto* incoerente *nello schema σ sse $S \simeq_\sigma \bot$.*

Uno schema σ è detto *coerente* sse per ogni $N \in \mathbf{N}$, $N \not\simeq_\sigma \bot$. Si noti che in uno schema coerente vi possono essere dei tipi incoerenti usati nei tipi set e sequence: infatti i tipi $\{\bot\}$ e $\langle\bot\rangle$ sono coerenti e denotano rispettivamente l'insieme vuoto e la sequenza vuota.

La relazione intuitiva tra ereditarietà e sussunzione è espressa dalla seguente proposizione:

Proposizione 7 *Dato uno schema σ, siano $N, N' \in \mathbf{N}$; se $N \preceq_\sigma N'$ allora $N \sqsubseteq_\sigma N'$.*

In generale il contrario non vale. La principale ragione è la semantica data ai nomi $N \in \mathbf{D} \cup \mathbf{V}$. Un'altra ragione è che un nome di tipo può essere incoerente. Per schemi coerenti possiamo dare un viceversa parziale alla precedente proposizione.

Proposizione 8 *Dato uno schema coerente σ, sia $N \in \mathbf{C} \cup \mathbf{D}$ e $N' \in \mathbf{C}$; allora $N \sqsubseteq_\sigma N'$ sse $N \preceq_\sigma N'$.*

In altri termini, in uno schema coerente le relazioni di sussunzione in cui la *superclasse* è una classe base sono solo quelle stabilite esplicitamente tramite la relazione di ereditarietà.

Tramite la relazione di sussunzione si individuano, per ogni nome N, tutte le sue *generalizzazioni* rispetto all'intera tassonomia, dalle quali è possibile selezionarne le *più specifiche*. Formalmente, dato uno schema σ, per un nome $N \in \mathbf{N}$ si definisce l'insieme $GS(N)$ (*Generalizarions Set*):

$$GS(N) = \{N' \in \mathbf{N} \mid N' \neq N \wedge N \sqsubseteq_\sigma N'\}$$

e l'insieme $MSGS(N)$ (*Most Specialized Generalizarions Set*)

$$MSGS(N) = \{N' \in GS(N) \mid \nexists N'' \in GS(N) \colon N'' \sqsubseteq_\sigma N'\}$$

In uno schema si può introdurre la *classe universale*, denotata con $\top_C$, che rappresenta la classe *più generale* dello schema e che quindi sussume tutte le altre classi, come classe virtuale con descrizione $\sigma(\top_C) = \triangle\top$. Infatti è immediato verificare che, in qualsiasi istanza possibile $\mathcal{I}$, si ha sempre $\mathcal{I}[\top_C] = \mathcal{O}$.

4.2 $\mathcal{OLCD}$

Il formalismo $\mathcal{OLCD}$(Object Language with Complements allowing descriptive cycles) è una logica descrittiva per basi di dati che estende il precedente *ODL*, mediante l'intro-

duzione di nuovi costruttori per l'insieme dei tipi e mediante, [BBB$^+$95], la possibilità di poter descrivere delle *regole d'integrità*, dette *"rule"*, che permettono la formulazione dichiarativa di un insieme rilevante di vincoli di integrità sotto forma di regole *if-then* i cui antecedenti e conseguenti sono espressioni di tipo $\mathcal{OLCD}$. In tale modo, è possibile descrivere correlazioni tra proprietà strutturali della stessa classe, o condizioni sufficienti per il popolamento di sotto-classi di una classe data. In altre parole le *rule* costituiscono uno strumento dichiarativo per descrivere gli oggetti che popolano il sistema.

4.2.1 Unione, complemento, tipi insieme ed path

In $\mathcal{OLCD}$ la descrizione dei tipi può avvenire secondo la seguente regola sintattica astratta:

S	$\rightarrow$	$\top$	
		B	tipo atomico
		N	nome di tipo
		$\neg S$	complemento
		$\{S\}_\forall$	tipo insieme
		$\{S\}_\exists$	tipo insieme esistenziale
		$[a_1 \colon S_1, \ldots, a_k \colon S_k]$	tipo tupla
		$S \sqcup S'$	unione
		$S \sqcap S'$	intersezione
		$\triangle S$	tipo oggetto
		$p \theta d$	restrizione ad un intervallo
		$p \uparrow$	path indefinito

A differenza di *ODL*, nel quale era presente un'unico operatore per tipo insieme, $\mathcal{OLCD}$ considera il tipo insieme universale e il tipo insieme esistenziale. Il primo indica il comune costruttore d'insieme già visto in *ODL*, invece il secondo, $\{S\}_\exists$, denota un insieme in cui almeno un elemento è di tipo S. L'introduzione dei path permette di rappresentare delle sequenze di elementi $p = e_1 \ldots e_n$ tali che $e_n \in \{\Delta, \exists\}$; con θ s'in-

dicano delle relazioni mentre con d si denotano dei valori base. L'insieme di tutti i path è denominato **W**, e per indicare il path vuoto viene utilizzato il simbolo ϵ. Con l'estensione dei path è stato introdotto il path indefinito, $p\uparrow$, mediante il quale si possono specificare gli attributi opzionali, *undefinedness*, nella descrizione di un tipo.

Un'altro degli aspetti importanti del modello è l'introduzione degli operatori di unione e complemento che, unitamente ai path, rendono possibile l'integrazione nello schema delle "*rule*".

Per non appesantire troppo la trattazione si rimanda, per la funzione d'interpretazione $\mathcal{I}$ corrispondente alla descrizione dei tipi di $\mathcal{OLCD}$ e per ulteriori approfondimenti, a [BBLS95].

4.3 Considerazioni sulla semantica adottata

In 2.1.2 sono state introdotte la semantica descrittiva e le semantiche di punto fisso seguendo un metodo più intuitivo che formale, mentre in questo capitolo sono state riprese in modo più rigoroso, concentrando soprattutto l'attenzione sulla semantica di massimo punto fisso, ovvero sulla semantica utilizzata dal modello $\mathcal{OLCD}$. Il fatto che un modello utilizzi, per la rappresentazione dei cicli, un approccio piuttosto che un altro è una questione di scelte, e non ne pregiudica le qualità. Tuttavia la semantica di punto fisso non sempre cattura l'intuizione dell'estensione di un concetto definito in modo ciclico [Neb91]; infatti per molti concetti il minimo punto fisso è l'insieme vuoto, mentre il massimo enfatizza troppo la struttura delle descrizioni, assegnando la stessa estensione anche a concetti intuitivamente distinti. Oltretutto, aumentando l'espressività del linguaggio introducendo, ad esempio, il complemento, la funzione d'estensione può perdere la proprietà di monotonicità. Questo può portare alla non unicità dei punti fissi minimo e massimo, perdendo di fatto l'intuizione dell'assegnamento di un'unica semantica alle definizioni.

La semantica descrittiva abbandona la tecnica costruttiva, per guardare alla terminologia come una descrizione dei modelli ammissibili. Le definizioni sono considerate delle equazioni semantiche che i modelli devono soddisfare. La relazione di sussunzione tra i

concetti in questo caso è più debole rispetto alla definizione tramite punto fisso; perchè, di fatto, si ammette come estensione di un concetto uno qualsiasi dei punti fissi della sua funzione d'estensione. Il risultato è che la struttura della definizione passa in secondo piano, mentre assumono più importanza i nomi che vi compaiono; così due descrizioni strutturalmente simili, ma con nomi di concetti e ruoli diversi, non necessariamente identificano la stessa estensione. D'altronde questo comportamento è perfettamente intuitivo; ad esempio, si considerino le due definizioni cicliche:

$$\begin{aligned} Cat &\equiv Mammiferous \sqcap (\forall hasChild.Cat) \\ Dog &\equiv Mammiferous \sqcap (\forall hasChild.Dog) \end{aligned}$$

La loro struttura è identica, ma questo non è un buon motivo per assumere che l'estensione del concetto *Cat* sia uguale a quello di *Dog*, come invece stabilisce la semantica di punto fisso.

Comunque, a prescindere da queste considerazioni, come verrà discusso nel capitolo 6, nella corrispondenza tra $\mathcal{OLCD}$ e $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ verrà utilizzata la semantica descrittiva, che è appunto quella utilizzata nell'implementazione del sistema Racer.

Capitolo 5

ODB-Tools Engine

5.1 Descrizione

ODB-Tool è un framework integrato per la validazione degli object oriented database (OODB), per la quale preserva la coerenza della tassonomia e compie inferenze di tipo tassonomico, e per l'ottimizzazione semantica di query. ODB-Tool è basato sulla Description Logics OLCD proposta come formalismo comune per esprimere la descrizioni di classi, un insieme rilevante di vincoli di integrità (IC rules) e di query. É un ambiente aperto, compatibile con lo standard ODMG e dotato di un'intefaccia CORBA-2. Come interfaccia verso l'utente esterno adotta la proposta ODMG-93 [Cat96], utilizzando il linguaggio **ODL**[1] (Object Definition Language) per la definizione degli schemi ed il linguaggio OQL (Object Query Language) per la scrittura di query. Entrambi i linguaggi sono stati estesi per la piena compatibilità con il formalismo $\mathcal{OLCD}$. Di recente ODB-Tools supporta $\mathbf{ODL}_{I^3}$.

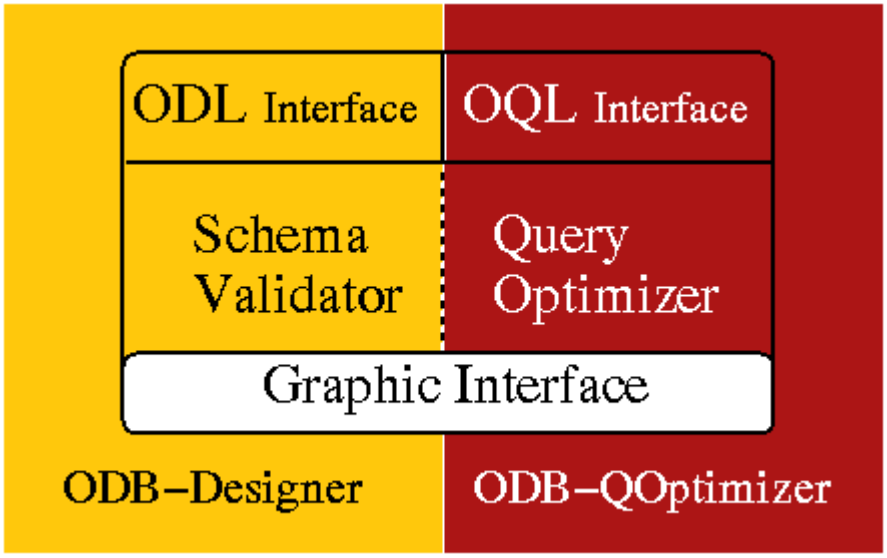

Figura 5.1: Architettura di ODB-tools

5.1.1 Architettura del sistema

L'architettura del sistema, mostrata in figura 5.1 , presenta i vari moduli integrati che definiscono un ambiente user-friendly basato sul linguaggio standard ODMG-93. L'utente inserisce gli schemi in linguaggio ODL e le query in OQL ottenendo come risultato la validazione dello schema, l'ottimizzazione dell'interrogazione (in OQL) e la rappresentazione grafica della gerachia di ereditarietà e di aggregazione dello schema; tutte queste funzionalità sono visibili ed utilizzabili attraverso il sito web:

http://www.dbgroup.unimo.it/ .

Vediamo in dettaglio la descrizione di ciascun modulo:

- **ODL** Interface: è il modulo di input degli schemi. Accetta la sintassi **ODL** e trasforma le classi in descrizioni native del formalismo $\mathcal{OLCD}$.

- OQL Interface: è il modulo di input e di output delle interogazioni. Utilizza il

[1] **ODL** non è *ODL*, la sigla è la stessa ma non ci sono problemi d'inpretazione perchè il loro significato è chiaro dal contesto: **ODL** è un linguaggio, *ODL* una DL.

linguaggio OQL sia per l'input che per l'output della query ottimizzata. I predicati booleani in output sono differenziati a seconda del proprio significato:

  - i fattori introdotti, o modificati, dall'ottimizzazione sono mostrati in colore rosso;
  - i fattori non modificati sono mostrati in colore grigio;
  - i fattori ignorati sono mostrati in colore nero;

  In output all'ottimizzazione non sono visualizzati i fattori ridondanti, cioè quei fattori identici a quelli descritti nelle classi referenziate dalla query.

- Schema Validator: è il modulo di validazione degli schemi, ottenuta dal calcolo delle relazioni di sussunzione e dei tipi incoerenti e dall'espansione semantica dei tipi. Produce come output un insieme di file utilizzati dagli altri moduli per interpretare e rappresentare i risultati.

- Query Optimizer: è il modulo che genera l'ottimizzazione delle interrogazioni. La query viene inserita come descrizione nativa $\mathcal{OLCD}$ dal modulo *OQL Interface* e, tramite l'interazione con lo *Schema Validator*, viene ottimizzata calcolandone l'espansione semantica. La query così ottimizzata viene nuovamente inviata all'*OQL Interface* che genera l'output corretto.

- Graphic Interface: è il modulo per la visualizzazione dello schema. Tale rappresentazione è costituita da un grafo i cui nodi rappresentano le classi e gli archi orientati le relazioni di ereditarietà e di aggregazione (opportunamente distinte); per ciascuna classe è possibile visualizzare i nomi ed i domini degli attributi (sia semplici che complessi). Lo schema contiene anche i vincoli di integrità rappresentati ciascuno tramite due classi che specificano l'antecedente ed il conseguente della regola *if-then*. Durante il processo di ottimizzazione la query entra a far parte dello schema con la dignità di classe e di conseguenza viene automaticamente inserita nella gerarchia di ereditarietà.

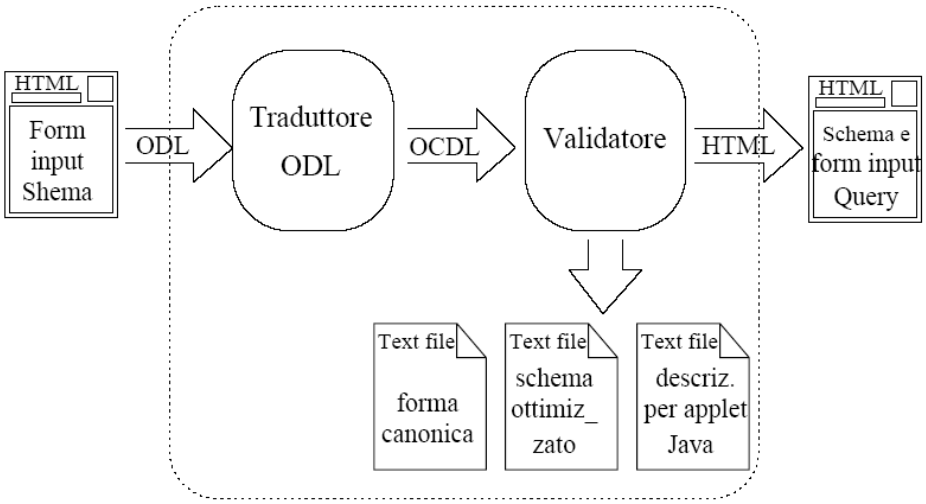

Figura 5.2: Validazione dello schema

5.2 Il linguaggio ODL

Object Definition Language (**ODL**) è il linguaggio per la specifica dell'interfaccia di oggetti e di classi nell'ambito della proposta di standard ODMG-93. Il linguaggio svolge negli ODBMS le funzioni di definizione degli schemi che nei sistemi tradizionali sono assegnate al Data Definition Language. Le caratteristiche fondamentali di ODL, al pari di altri linguaggi basati sul paradigma ad oggetti, possono essere così riassunte:

- definizione di classi e tipi valore;
- distinzione tra intensione ed estensione di una classe di oggetti;
- definizione di attributi semplici e multivalore (set, list, bag);
- definizione di relazioni e relazioni inverse tra classi di oggetti;
- definizione della signature dei metodi.

La sintassi di **ODL** estende quella dell'Interface Definition Language, il linguaggio sviluppato nell'ambito del progetto Common Object Request Broker Architecture (CORBA)[VV.93].

5.2.1 ODL$_{I^3}$

Sviluppato in seguito alla diffusione di I^3 POP[], **ODL**$_{I^3}$ rappresenta l'evoluzione di **ODL**, ed è il linguaggio attualmente utilizzato per la descrizione di schemi eterogenei nel sistema ODB-Tools. Le principali estensioni introdotte dal linguaggio rispetto ad **ODL** sono:

- Operatore di unione (union): esprime delle strutture alternative nella definizione di una **ODL**$_{I^3}$ classe.
- Costruttore opzionale (*): permette di indicare che un attributo è opzionale per un'istanza.
- Regole di integrità: permettono di specificare, in via dichiarativa, le regole di integrità.

  rule <nome-regola> for all <nome-iteratore> in <nome-classe> :
  <condizione-antecedente> then <condizione-conseguente>

  Le condizioni, antecedente e conseguente, hanno la medesima forma e sono costituite da una lista di espressioni booleane in and tra loro; all'interno di una condizione, attributi e oggetti sono identificati mediante la dot notation.
- Relazioni intensionali (SYN, RT, BT, NT): rappresentano delle relazioni terminologiche che esprimono conoscenza iter/intra schema per le sorgenti degli schemi. Sono definite tra classi e attributi.
- Relazioni estensionali (SYN$_{ext}$, RT$_{ext}$, BT$_{ext}$, NT$_{ext}$): rappresentano delle relazioni tra le classi.
- Chiavi esterne (foreing key): servono per preservare le informazioni delle classi(tabelle) proprie degli schemi relazionali.

- Regole di mapping: sono delle speciali regole che permettono di esprimere le relazioni tra le descrizione **ODL**$_{I^3}$ dello schema delle sorgenti di informazione e lo schema **ODL**$_{I^3}$ delle sorgenti originali.

5.2.2 Schema di esempio

Introduciamo ora uno schema che esemplifica la sintassi **ODL**$_{I^3}$ utilizzata da ODB-Tools. L'esempio descrive la realtà universitaria rappresentata dai docenti, gli studenti, i moduli dei corsi e le relazioni che legano tra loro le varie classi. In particolare esiste la classe dei moduli didattici (Section), distinti in moduli teorici (STheory) e di esercitazione (STraining). La popolazione universitaria è costituita da dipendenti (Employee) e studenti (Student). Un sottinsieme dei dipendenti è costituito dai professori (Professor). che uno studente dell'Università. I moduli vengono seguiti da studenti e, se teorici, sono tenuti da professori.

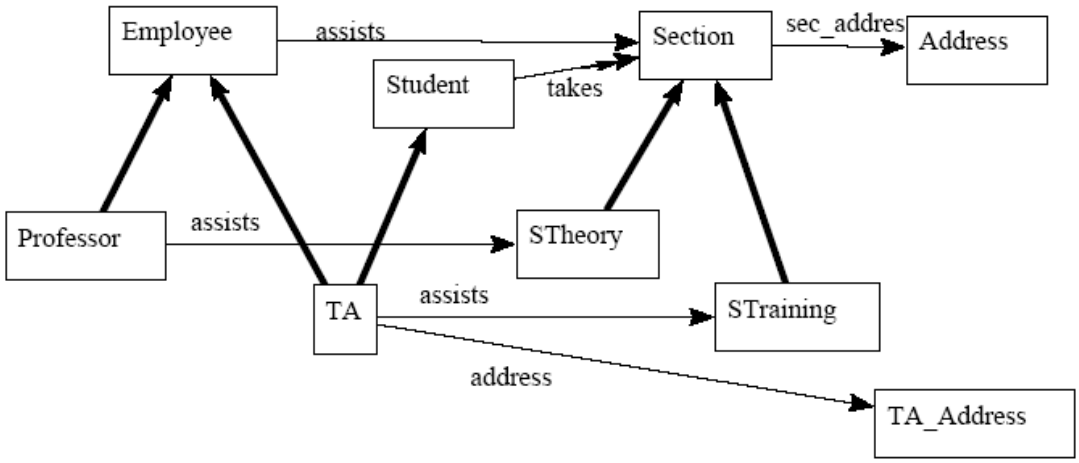

Figura 5.3: Rappresentazione grafica dello schema Università

```
interface Address
{
    attribute string city;
```

```
    attribute string street;
    attribute string zipcode;
};
union Address2
{
    attribute string dummy;
};


interface Section ()
{
    attribute string number;
    attribute Address sec_address;
    attribute String tel_number*;
};


interface STheory : Section()
{
    attribute integer level;
};


interface Student ()
{
    attribute string name;
    attribute integer student_id;
    attribute set<Section> takes;
};
```

```
interface Employee ()
{
    attribute string name;
    attribute real annual_salary;
    attribute string domicile_city;
    attribute Section assists;
};

interface Professor: Employee ()
{
    attribute string rank;
    attribute STheory assists;
};
```

In figura 5.3 viene rappresentato lo schema risultante, in cui le classi sono rappresentate da rettangoli, le relazioni di aggregazione tra classi tramite frecce (semplici per quelli monovalore, doppie per quelli multivalore), e le relazioni di ereditarietà sono rappresentate graficamente da frecce più marcate.

5.2.3 Traduzione da ODL$_{I^3}$ ad $\mathcal{OLCD}$

Vediamo ora le principali regole di traduzione da **ODL**$_{I^3}$ ad $\mathcal{OLCD}$.

- **ODL**$_{I^3}$ classi: la traduzione delle classi C in $\mathcal{OLCD}$ risulta piuttosto semplice: ogni attributo della classe ODL diventa un attributo della corrispondente classe $\mathcal{OLCD}$. Facendo riferimento all'esempio, la traduzione della classe Employee sarà la seguente:

$$\begin{aligned} Employee \;=\; & \Delta[name : String, annual\_salary : Real, \\ & domicile\_city : String, assists : Section] \end{aligned}$$

Le classi base sono introdotte tramite la parola chiave interface, mentre per le classi virtuali si introduce il costrutto view che specifica la classe come virtuale seguendo le stesse regole sintattiche della definizione di una classe base. Ad esempio, la seguente dichiarazione:

view Assistant: Employee, Student ()
{ attribute Address address; };

introduce la classe virtuale Assistant che rappresenta tutti gli oggetti appartenenti sia alla classe studenti che dipendenti e che, in più, hanno un indirizzo rappresentato dalla struttura Address definita dagli attributi via e città. Alcuni aspetti della descrizione $\mathcal{OLCD}$ delle classi non sono tradotti in $\mathcal{OLCD}$, ma verranno usati nel processo di integrazione semantica delle informazioni (ad esempio la chiave *employee_code* della classe Employee).

- Operatore di unione: il costruttore union dell'**ODL**$_{I^3}$ è tradotto in $\mathcal{OLCD}$ usando il costrutto $\sqcup$. Sempre facendo riferimento all'esempio, la traduzione della classe Address sarà:

$$\begin{aligned} Address &= \Delta[dummy : String \sqcup [city : String, \\ &\quad street : String, zipcode : String]] \end{aligned}$$

- Costruttore opzionale: il costruttore $\sqcup$ può inoltre essere usato per tradurre gli attributi opzionali in $\mathcal{OLCD}$. Infatti, un attributo opzionale indica che un valore può o meno esistere per una data instanza. Tutto questo è espresso in $\mathcal{OLCD}$ come l'unione tra la descrizione dell'attributo e l'attributo non definito, denotato tramite l'operatore $\uparrow$. Se consideriamo la descrizione **ODL**$_{I^3}$ della classe Section notiamo la presenza di attributi opzionali (contraddistinti da un *) che vengono così tradotti:

$$\begin{aligned} Section &= \Delta[number : String, sec\_Address : Address] \\ &\quad \sqcap([tel\_number : String \sqcup tel\_number \uparrow) \end{aligned}$$

- Regole di integrità: le regole d'integrità di tipo *if-then* sono integrate in $\mathcal{OLCD}$ usando i costrutti ¬, ⊓, ⊔. Ad esempio, introduciamo la seguente regola di integrità nello schema di riferimento:

  rule rule_1 forall X in Professor:
  X.rank = "Full" then X.annual_salary >= 60000 ;

  La rule 1 stabilisce che un professore di rango Full (cioè un professore ordinario) abbia un salario superiore a 60000; essa viene tradotta in:

$$\begin{aligned} Professor \quad = \quad & Employee \sqcap \Delta([rank : String, assist : STheory] \\ & \sqcup(\neg(\Delta.rank = \text{“}Full''))\sqcup(\Delta.salary > 60000)) \end{aligned}$$

- Relazioni intensionali (SYN, RT, BT, NT): non sono tradotte.

- Relazioni estensionali (SYN_{ext}, RT_{ext}, BT_{ext}, NT_{ext}): Ogni relazione del tipo C1 ISA C2 è espressa in ODL da una rule del tipo:

  rule Rule1 forall X in C1 then X in C2

  ed è integrata nella descrizione della classe C1 usando il costrutto ⊓.

- Regole di mapping: non sono tradotte.

Capitolo 6

Attività di Ragionamento nel linguaggio $\mathbf{ODL}_{I^3}$ tramite il sistema RACER

In questo capitolo verrà discussa la progettazione e l'implementazione di un traduttore dal linguaggio $\mathbf{ODL}_{I^3}$ a quello di RACER, più precisamente dal sottoinsieme del linguaggio $\mathbf{ODL}_{I^3}$ che trova corrispondenza in $\mathcal{OLCD}$, indicato con $\mathbf{ODL}_{I^3}$-$\mathcal{OLCD}$, al linguaggio RLI. L'obiettivo è quello di estendere, all'interno del sistema MOMIS, la capacità di ragionamento dal linguaggio $\mathbf{ODL}_{I^3}$-ODL, supportato da ODB-Tools, al linguaggio $\mathbf{ODL}_{I^3}$-$\mathcal{OLCD}$ che costituisce un sovrainsieme significativo di $\mathbf{ODL}_{I^3}$-ODL in quanto introduce, tra l'altro, unione e negazione. Il capitolo è organizzato come segue. Prima di tutto verranno discusse le differenze tra gli OODMs e le DLs e verrà presentato un modello generale di traduzione da OODMs a DLs proposto in letteratura. Quindi verranno discusse le regole per la traduzione da uno schema espresso in $\mathcal{OLCD}$ ad uno schema equivalente espresso in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$; lo schema ottenuto verrà espresso nel linguaggio RLI e quindi verranno illustrati vari esempi di controllo di consistenza di uno schema $\mathcal{OLCD}$ in RACER. Infine, basandoci su tali regole di traduzione, verra' discussa la traduzione dal linguaggio $\mathbf{ODL}_{I^3}$ a quello di RACER, RLI.

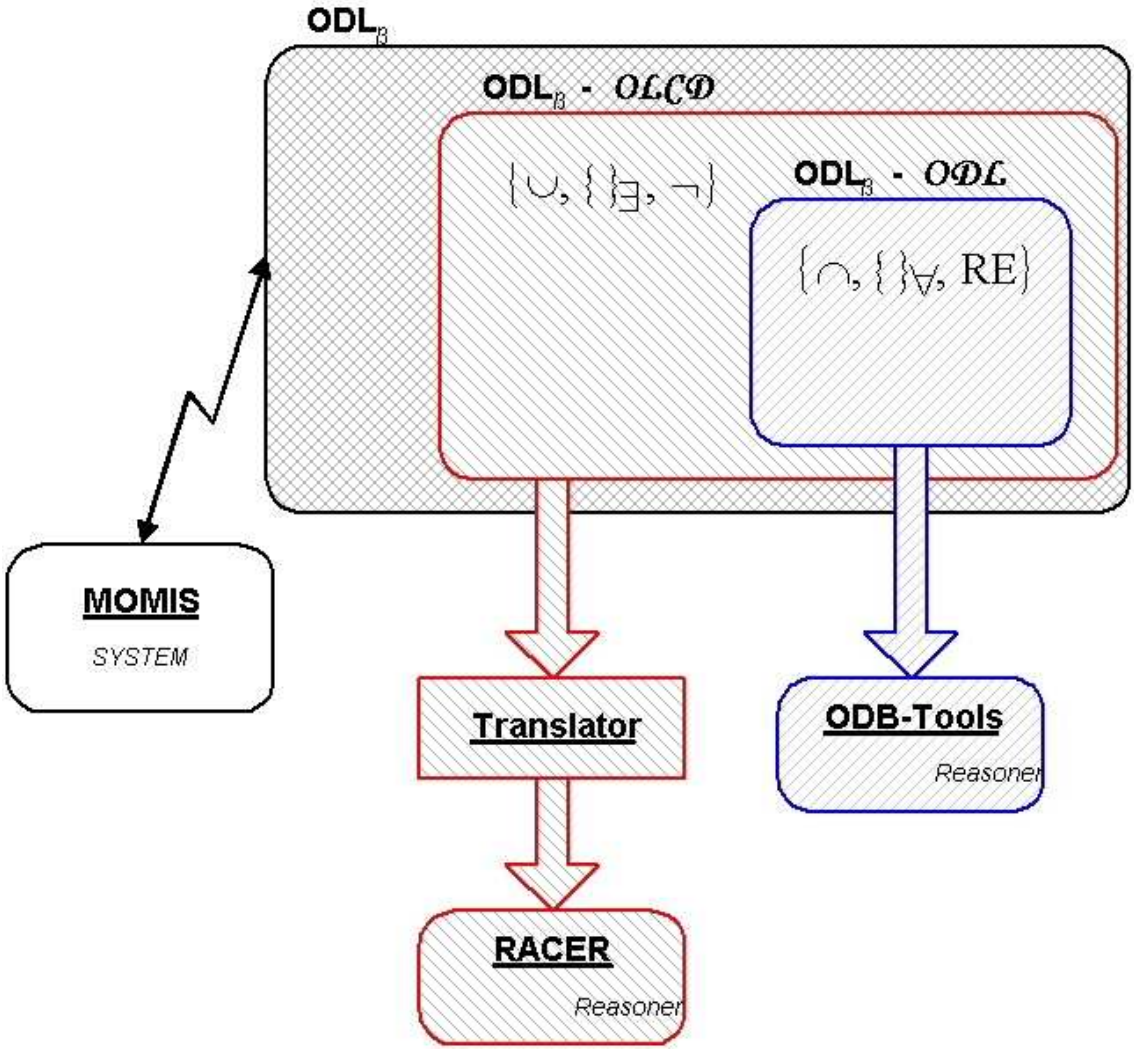

Figura 6.1: Attività di reasoning con il linguaggio **ODL**$_{I^3}$

6.1 Differenze tra il modello OO e le DL

I modelli di dati orientati agli oggetti (OODM: object oriented data model) si basano, a livello estensionale, sulla nozione di object identifier e, a livello intensionale, su quella di classe. Per passare da un OODM ad una logica descrittiva è necessario quindi concentrare l'attenzione sull'aspetto strutturale del modello ad oggetti.

Senza riferirsi ad un particolare formalismo, uno schema object oriented è un insieme finito di dichiarazioni di classe, che impone delle restrizioni sulle istanze delle classi

utilizzate per modellare il dominio di applicazione. In generale, una dichiarazione di classe ha la forma:

class C *is-a* $C_1, \ldots, C_n$ *type-is T*

dove is-a indica l'inclusione delle istanze di C negli insiemi C_1,...,C_n specificati e type-is indica che, la struttura assegnata agli oggetti, istanze di C, deve essere concorde con l'espressione di tipo T. Considerando gli operatori: tipo tupla (record) ed tipo insieme (set-of), possiamo scrivere le seguenti espressioni di tipo:

T	$\rightarrow$	C	nome di tipo
		$[a_1 : T_1, \ldots, a_k : T_k]$	tipo tupla
		$\{T\}$	tipo insieme

e attribuire a ciascuna di esse un'interpretazione semantica in accordo con quanto già scritto in 4.1.3.

Si può notare che il dominio di interpretazione di uno schema object oriented consiste in oggetti strutturati, mentre quello delle logiche descrittive è basato su degli oggetti atomici. Questo significa che per passare da una rappresentazione OO ad una equivalente in DL è neccessario rappresentare in modo esplicito la struttura delle classi.

6.1.1 Modello generale di traduzione da OO a DL

Un modello di traduzione, per passare dal paradigma OO ad una DLs di derivazione KL-ONE, è stato proposto in [CLN94] e ripreso in [AFM03]; tale modello è anche riportato in [SCM03]. Esso si basa sulla costruzione della base di conoscenza a partire da tre concetti di fondamentali a quali viene attribuito un significato particolare:

- *AbtractClass*: è il concetto padre di tutte le classi definite nello schema OO;
- *RecType*: è il concetto padre per tutti i tipi tupla definiti nello schema OO;
- *SetType*: è il concetto padre per tutti i tipi insieme definiti nello schema OO.

Inoltre, il modello di traduzione considera due ruoli speciali:

- *value*: è l'abstract feature che modella l'associazione tra le classi e i tipi oggetto;
- *members*: è il ruolo che permette di specificare gli elementi di un tipo insieme.

Sotto le condizioni che i concetti che rappresentano i tipi, *RecType* e *SetType*, siano tra loro disgiunti, e siano disgiunti da concetto, *AbstractClass*, che modella le classi, si possono scrivere le seguenti asserzioni di inclusione, che devono essere presenti in qualsiasi base di conoscenza ottenuta dalla traduzione di uno schema OO:

$$\begin{aligned} AbstractClass &\sqsubseteq = 1 value \\ RecType &\sqsubseteq \forall value.\bot \\ SetType &\sqsubseteq \forall value.\bot \sqcap \neg RecType \end{aligned}$$

La traduzione deve avvenire mediante una funzione Γ che mappa ciascuna espressione di tipo in un'espressione di concetto secondo quanto segue:

- ciascuna classe C è mappata in un concetto atomico:

$$\Gamma\,(C);$$

- ciascuna espressione di tipo insieme $\{\,T\,\}$ è mappata in:

$$SetType \sqcap \forall member.\Gamma\,(T);$$

- ciascun attributo A è mappato in un ruolo:

$$\Gamma\,(A)$$

- ciascuna espressione di tipo tupla $[a_1 : T_1, \ldots, a_k\colon T_k]$ è mappata in:

$$RecType \sqcap \forall\,\Gamma(a_1).\Gamma(T_1) \sqcap\, =\, 1\,\Gamma(a_1)\, \sqcap \ldots \sqcap \forall\,\Gamma(a_k).\Gamma(T_k) \sqcap\, =\, 1\,\Gamma(a_k).$$

La corrispondenza della base di conoscenza $\Gamma(Schema)$ con lo schema object oriented è ottenuta sostituendo ad ciascuna dichiarazione di classe:

class C *is-a* $C_1, \ldots, C_n$ *type-is* T

un asserzione di inclusione tra concetti nella forma:

$$\Gamma\,(C) \sqsubseteq AbstractClass \sqcap \Gamma(C_1) \sqcap \ldots \sqcap \Gamma(C_n) \sqcap \forall\, value.\Gamma(T).$$

Per stabilire la correttezza della trasformazione, e quindi essere certi che il ragionamento sullo schema object oriented possa essere ricondotto al ragionamento sulla sua traduzione in forma di DLs, è necessario stabilire la corrispondenza tra gli stati legali del database, rispetto allo schema, e i modelli della base di conoscenza ottenuta dalla traduzione. Comunque la base di conoscenza potrebbe avere dei modelli che non corrispondono direttamente allo stato legale del database. In questo caso bisogna tenere presente che i valori hanno una struttura del tipo ad albero, mentre i corrispondenti individui del modello su DL possono essere parte di sotto strutture cicliche. Comunque è dimostrato che è sempre possibile ricondurre il modello ottenuto dalla traduzione in uno che corrisponde direttamente a uno stato legale di database.

6.2 Regole per la traduzione da $\mathcal{OLCD}$ a $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

Per tradurre uno schema $\mathcal{OLCD}$ in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, che risulti inferenziabile da RACER, è necessario dapprima creare la struttura di base che tenga conto, da un lato, della necessita di esplicitare il modello ad oggetti e, dall'altro, delle limitazioni imposte dal sistema. In particolare, è necessario definire delle regole per mappare, nel sistema di rappresentazione della conoscenza, l'insieme dei tipi atomici **B**, l'insieme numerabile di attributi **A** e gli insiemi numerabili dei nomi per i: tipi classe base **C**, tipi classe virtuale **V** e tipi valore **T** del modello $\mathcal{OLCD}$.

6.2.1 Insieme dei tipi atomici

Ricordando che $\mathcal{OLCD}$ definisce il seguente insieme di designatori per i tipi atomici:

$$\mathbf{B} = \{\texttt{Integer}, \texttt{String}, \texttt{Bool}, \texttt{Real}, i_1\text{–}j_1, i_2\text{–}j_2, \ldots, d_1, d_2, \ldots\},$$

chiamiamo i tipi: `Integer`, `String`, `Real`, *tipi primitivi,* e tutti i possibili intervalli i_n–j_n di interi, *tipi range*.

Si può notare che la logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ di RACER supporta, tramite i domini concreti, il ragionamento sui *tipi primitivi*, e quindi, per essi, l'insieme dei designatori può essere costruito mediante la dichiarazione di un insieme di concrete features, tale che l'insieme $\mathcal{A}$ di RACER sia definito come:

$$\mathcal{A} \; = \; \{\; INTEGER,\; REAL,\; STRING \;\},$$

In questo modo si creano dei nomi di concrete features riservati che devono avere l'interpretazione semantica riportata in tabella 6.1.

Nome Concrete Feature	**Interpretazione Semantica**
INTEGER	INTEGER^I : $\Delta^I \rightarrow \Delta^{integer}$
REAL	REAL^I : $\Delta^I \rightarrow \Delta^{real}$
STRING	STRING^I : $\Delta^I \rightarrow \Delta^{string}$

Tabella 6.1: Nomi di concrete features riservati per i tipi primitivi.

Si può effettuare il mapping, tra i tipi primitivi del modello $\mathcal{OLCD}$ e i domini concreti supportati da RACER, secondo lo schema riportato nella tabella 6.2.

Per quanto riguarda i *tipi range*, essi devono essere implementati imponendo delle restrizioni sui valori che può assumere il riempitivo della concrete features INTEGER.
Un generico range:

$$i_n\text{–}j_n,$$

Tipo $\mathcal{OLCD}$		**Sintassi DL**
integer	$\Leftrightarrow$	$\exists$ INTEGER.$\top_{\textit{integer}}$
real	$\Leftrightarrow$	$\exists$ REAL.$\top_{\textit{real}}$
string	$\Leftrightarrow$	$\exists$ STRING.$\top_{\textit{string}}$

Tabella 6.2: Schema di traduzione per i tipi primitivi.

viene tradotto in:

$$(\exists \text{ INTEGER.}\top_{\textit{integer}}) \sqcap (\exists \text{ INTEGER.}min_{i_n}) \sqcap (\exists \text{ INTEGER.}max_{j_n}).$$

Si può notare che, allo stesso modo, è possibile realizzare degli intervalli chiusi, aperti, aperti illimitati e chiusi illimititati, basati sull'insieme dei numeri reali. Infatti, possiamo esprimere un range sulla concrete feature REAL semplicemente come:

$$(\exists \text{ REAL.}\top_{\textit{real}}) \sqcap (\exists \text{ REAL.}>_{i_n}) \sqcap (\exists \text{ REAL. } <_{j_n}).$$

In base a quanto detto, è possibile considerare un concetto "RangeType", definito da:

$$RangeType \quad \sqsupseteq \quad (\exists INTEGER.\top_{\textit{integer}}) \sqcup (\exists REAL.\top_{\textit{real}}) \sqcup (\exists STRING.\top_{\textit{string}})$$

al quale è possibile ricondurre l'insieme dei tipi $\mathbf{B}$ di $\mathcal{OLCD}$.

6.2.2 Rappresentazione della struttura Object-Oriented

Affinchè la base di conoscenza descritta in forma di DL tenga conto delle prerogative del modello OO, prendendo spunto da 6.1.1 e tenendo presenti le considerazioni precedenti, s'introducono nell'insieme $\mathcal{C}$ i nomi di concetto riservati: "ClassType", "RecType", "SetType" e "RangeType" ed "UnionType". Inoltre si considerano i ruoli: "value" e "member" introdotti a pagina 110. Si noti che il ruolo "value" ha la stessa funzione del costruttore Δ del modello $\mathcal{OLCD}$: entrambi modellano l'associazione tra le classi e i tipi oggetti.

Nome Ruolo	**Interpretazione Semantica**
value	value$^I \subseteq \Delta^I \times \Delta^I$
member	member$^I \subseteq \Delta^I \times \Delta^I$

Tabella 6.3: Nomi di ruoli riservati.

Nome Concetto	**Interpretazione Semantica**
ClassType	ClassType$^I \subseteq \Delta^I$
RecType	RecType$^I \subseteq \Delta^I$
SetType	SetType$^I \subseteq \Delta^I$
RangeType	RangeType$^I \subseteq \Delta^I$
UnionType	UnionType$^I \subseteq \Delta^I$

Tabella 6.4: Nomi di concetto riservati.

La corrispondenza tra il modello $\mathcal{OLCD}$ e la logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ viene formalizzata inserendo, nella parte terminologica della base di conoscenza, gli assiomi che indicano le relazioni che esistono tra i concetti sopra citati.
In particolare è necessario mantenere l'ipotesi che i concetti: "ClassType", "RecType", "SetType", "RangeType" risultino tra loro disgiunti; tale ipotesi è ottenibile dalle asserzioni di inclusione:

$$
\begin{aligned}
ClassType &\sqsubseteq (= \ 1 \ value) \\
RecType &\sqsubseteq (\forall value.\bot) \\
SetType &\sqsubseteq (\forall value.\bot) \sqcap (\neg Rectype) \\
RangeType &\sqsubseteq (\forall value.\bot) \sqcap (\neg AllSetType) \sqcap (\neg Rectype) \\
RangeType &\sqsubseteq (\exists INTEGER.\top_{\textit{integer}}) \sqcup (\exists REAL.\top_{\textit{real}}) \sqcup (\exists STRING.\top_{\textit{string}})
\end{aligned}
$$

Il concetto UnionType risulta necessario per esprimere l'unione tra concetti che hanno

radici tra loro disgiunte; esso viene definito tramite l'assioma:

$$UnionType \sqsubseteq ClassType \sqcup RangeType \sqcup SetType$$

Tipo insieme: universale ed esistenziale

$\mathcal{OLCD}$ considera i costruttori per il tipi insieme universale e il tipo insieme esistenziale. Affinchè siano resi distinguibili anche nella KB in logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ si possono considerare i concetti:

- AllSetType: per indicare il tipo insieme universale, definito dall'assioma:

$$AllSetType \sqsubseteq SetType \tag{6.1}$$

- SomeSetType: per indicare il tipo insieme esistenziale, definito dall'assioma:

$$SomeSetType \sqsubseteq SetType \tag{6.2}$$

che denotano due sottoinsiemi di SetType. Per attribuire loro la giusta interpretazione dobbiamo ragionare sul significato di insieme universale e insieme esistenziale. Consideriamo, ad esempio, gli insiemi:

$$\{Cat\}_\forall \quad ed \quad \{Cat\}_\exists,$$

il primo indica un insieme dove gli elementi sono tutti *Cat* oppure l'insieme è vuoto, mentre il secondo indica un insieme dove esiste almeno un elemento *Cat*. Applicando gli stessi operatori al concetto $\top$ otteniamo, intuitivamente, il significato dei concetti *AllSetType* e *SomeSetType*:

$$\begin{aligned} AllSetType &= \{\top\}_\forall \\ SomeSetType &= \{\top\}_\exists \end{aligned}$$

che indicano rispettivamente: un insieme dove esiste almeno un elemento oppure l'insieme è vuoto, e un insieme dove esiste almeno un elemento. In relazione al nostro modello nella logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, tenendo presente che il ruolo *member* è quello designato per denotare gli elementi dei tipi insieme, quest'intuizione si può esprimere mediante gli assiomi:

$$SetType \sqcap (>=\ 0\ member) \quad \sqsubseteq \quad AllSetType. \tag{6.3}$$

$$SetType \sqcap (>=\ 1\ member) \quad \sqsubseteq \quad (AllSetType \sqcap SomeSetType) \tag{6.4}$$

Si può dimostrare che 7.1, 7.2, 7.3, 7.4, si riducono a:

$$SetType \quad \equiv \quad AllSetType \tag{6.5}$$

$$SomeSetType \quad \sqsubseteq \quad AllSetType \tag{6.6}$$

$$AllSetType \sqcap (>=\ 1\ member) \quad \sqsubseteq \quad SomeSetType \tag{6.7}$$

Per cui nel modello è sufficiente sostituire al concetto *SetType* il concetto *AllSetType*, e aggiungere gli assiomi 6.6 e 6.7. La figura 6.2 mostra la tassonomia dei concetti introdotti dagli assiomi precedenti, che rappresentano la struttura di base che verrà utilizzata per la traduzione degli schemi.

6.2.3 Traduzione dello schema

Preparata la struttura di base, per effettuare una traduzione corretta dello schema, occorre definire per ciascun nome di attributo un ruolo e per ciascun nome di classe, sia base che virtuale, e per tutti i nomi dei tipi valore, un concetto atomico: le istanze di classe base e le istanze di classe virtuale devono avere come radice il concetto *Classtype*, mentre le istanze di tipo valore devono avere la radice in uno tra i concetti di *RecType*, *AllSetType*, *SomeSetType*, o *RangeType*.

Le istanze di classi primitive devono essere associate alla loro descrizione tramite l'operatore di inclusione generalizzata ($\sqsubseteq$), invece le istanze di classi virtuali e dei tipi valore devono essere associate alla loro descrizione tramite l'operatore di equivalenza ($\equiv$). Le traduzioni dei tipi devono seguire le regole mostrate nella tabella 6.6.

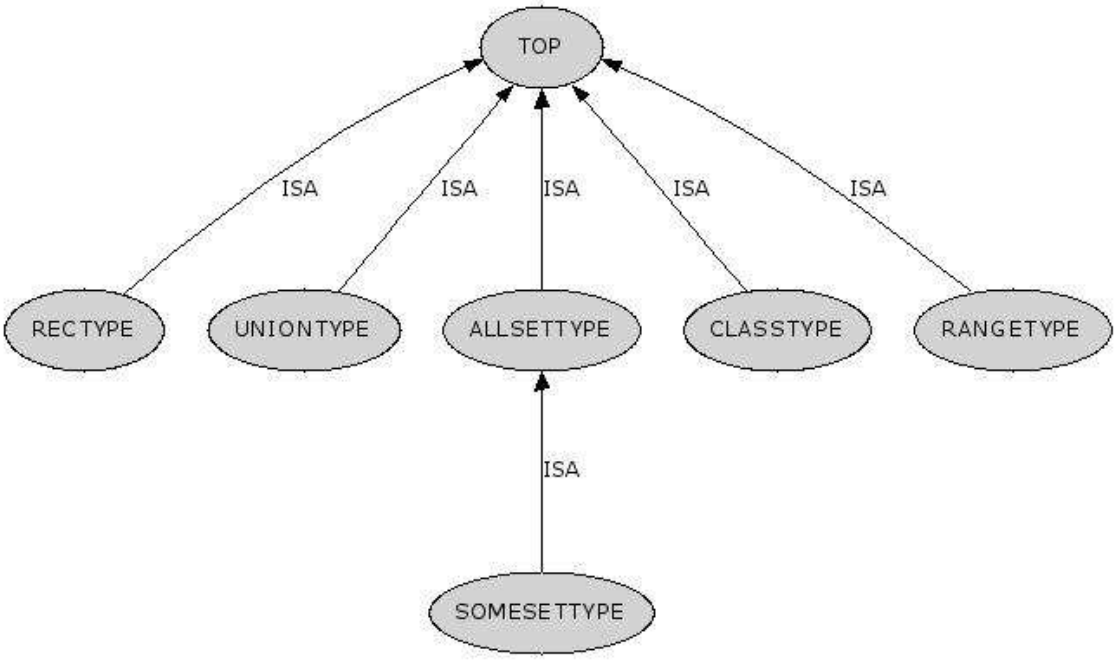

Figura 6.2: Tassonomia della struttura di base

OLCD		**DL**	**Interpretazione Semantica**
$a_j \in \mathbf{A}$	$\Leftrightarrow$	$\Gamma(a_j)$	$\Gamma(a_j)^I : \Delta^I \rightarrow \Delta^I$
$C \in \mathbf{C} \cup \mathbf{V}$	$\Leftrightarrow$	$\Gamma(C)$	$\Gamma(C)^I \subseteq \text{ClassType}^I \subseteq \Delta^I$
$T \in \mathbf{T}$	$\Leftrightarrow$	$\Gamma(T)$	$\Gamma(T)^I \subseteq \text{RecType}^I \cup \text{SetType}^I \cup \text{RangeType}^I \subseteq \Delta^I$
$N \in \mathbf{N}$	$\Leftrightarrow$	$\Gamma(N)$	$\Gamma(N)^I \subseteq \text{ClassType}^I \cup \text{RecType}^I \cup \text{AllSetType}^I \cup \text{RangeType}^I \subseteq \Delta^I$
$range_D \subseteq \mathbf{B}$	$\Leftrightarrow$	$\Gamma(range_D)$	$\Gamma(range_D)^I : \Delta^I \rightarrow \Delta^{\mathcal{D}}$

Tabella 6.5: Mapping degli elementi del modello $\mathcal{OLCD}$ nei corrispondenti elementi in DL.

Sintassi $\mathcal{OLCD}$		**Sintassi DL**	**Descrizione**
$\Gamma(\Delta$ X)	$\Leftrightarrow$	ClassType $\sqcap$ ($\forall$ value.Γ(X))	tipo oggetto
$\Gamma(\neg$ X)	$\Leftrightarrow$	$\neg\ \Gamma$(X)	complemento
Γ(X $\sqcap$ Y)	$\Leftrightarrow$	Γ(X) $\sqcap\ \Gamma$(Y)	intersezione
Γ(X $\sqcup$ Y)	$\Leftrightarrow$	UnionType $\sqcap(\Gamma$(X) $\sqcup\ \Gamma$(Y))	unione
$\Gamma([a_1 : X_1, \ldots,$ $a_n : Xn])$	$\Leftrightarrow$	RecType $\sqcap\ (\forall\ \Gamma(a_1).\Gamma(X_1)) \sqcap (= 1\ \Gamma(a_1))$ $\sqcap \ldots \sqcap\ (\forall\ \Gamma(a_n).\Gamma(X_n)) \sqcap (= 1\ \Gamma(a_n))$	tipo tupla
$\Gamma([a : X] \sqcup a \uparrow)$	$\Leftrightarrow$	RecType $\sqcap\ (\forall\ \Gamma(a).\Gamma(X))$	tipo opzionale
$\Gamma(\{X\}_\forall)$	$\Leftrightarrow$	AllSettype $\sqcap\ (\forall$ member.$\Gamma(X))$	tipo insieme
$\Gamma(\{X\}_\exists)$	$\Leftrightarrow$	SomeSettype $\sqcap\ (\exists$ member.$\Gamma(X))$	tipo insieme esistenziale
$\Gamma(range_D)$	$\Leftrightarrow$	Rangetype $\sqcap\ \Gamma(range_D)$	tipo atomico

Tabella 6.6: Regole di traduzione dalla sintassi $\mathcal{OLCD}$ alla sintassi DL e viceversa.

Traduzione degli attributi opzionali

Il problema inerente la traduzione degli attributi è il seguente: noi vogliamo tradurre gli attributi in modo tale che sia poi possibile risalire alla descrizione di un record tramite delle verifiche di sussunzione. Quest'esigenza nasce dal fatto che, da un lato, il modello $\mathcal{OLCD}$ introduce e permette l'ereditarietà multipla, mentre dall'altro, RACER è in grado di dare risposte solo nei termini di "è vero" oppure "non lo so", quindi per poter ricavare la descrizione del record appartenente ad una classe le strade percorribili sono due:

- ricavare la descrizione dei record in fase di traduzione, realizzando quindi un traduttore "intelligente";
- ricavare la descrizione dei record a posteriori, ovvero dopo che RACER ha inferenziato la tassonomia dello schema, tramite appunto delle verifiche di sussunzione.

Inoltre nella descrizione di un record deve essere possibile distinguere gli attributi "normali" da quelli "opzionali". In 5.2.3, abbiamo visto come vengono espressi gli attributi opzionali nella sintassi $\mathcal{OLCD}$. Analizziamo ora come possono essere tradotti nella sintassi $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ di RACER, con lo scopo di ricavare la descrizione dei record a posteriori.
Dato che, in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, gli attributi vengono mappati su dei ruoli, la traduzione di un attributo normale si ottiene forzandone la presenza all'interno del record, ovvero specificando nella descrizione una restrizione di cardinalità esatta sul ruolo *nome_attributo*:

$$(= \ 1 \ nome\_attributo),$$

e intersecandola con:

$$(\forall nome\_attributo.tipo),$$

per specificare il tipo del riempitivo. Da quanto sopra e dalla semantica dell'operatore $\forall$ delle DLs, la traduzione di un attributo opzionale si ottiene sostituendo la restrizione di cardinalità esatta con una restrizione di cardinalità massima:

$$(\leq 1 nome\_attributo\_opz) \sqcap (\forall nome\_attributo\_opz.tipo).$$

Per verificare che questa traduzione sia adatta allo scopo consideriamo un generico schema che contiene le descrizioni di quattro ipotetici tipi base:

$$\begin{aligned} TipoBase_1 &= [attrib_1 : tipo_1]; \\ TipoBase_2 &= [attrib_2 : tipo_2]; \\ TipoBase_3 &= [attrib_3 : tipo_3]; \\ TipoBase_4 &= TipoBase_1 \sqcap TipoBase_2; \end{aligned}$$

La loro traduzione in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, applicando quanto detto è:

$$\begin{aligned} TipoBase_1 &\sqsubseteq (\forall attrib_1.tipo_1) \sqcap (= 1\ attrib_1), \\ TipoBase_2 &\sqsubseteq (\forall attrib_2.tipo_2) \sqcap (\leq 1\ attrib_2), \\ TipoBase_3 &\sqsubseteq (\forall attrib_3.tipo_3) \sqcap (= 1\ attrib_3), \\ TipoBase_4 &\sqsubseteq TipoBase_1 \sqcap TipoBase_2; \end{aligned}$$

in cui $attrib_1$, $attrib_2$ e $attrib_3$ rappresentano dei nomi di ruolo. Ora, per determinare la presenza o meno di un attributo normale nella descrizione di un concetto è necessario e sufficiente verificare la validità della sussunzione:

$$Concetto \sqsubseteq (= 1 attrib),$$

e, allo stesso modo, per avere la certezza che sia presente un attributo opzionale, è necessario e sufficiente che non sia verificata la sussunzione precedente ma che sia verificata la sussunzione:

$$Concetto \sqsubseteq (\leq 1 attrib).$$

Ovviamente è nessario iterare il procedimento per ciascun attributo rispetto a ciascuna classe. Questo vuol dire che nel nostro caso si deve ottenere, e si ottiene che:

$$\begin{aligned} TipoBase_1 &\sqsubseteq (= 1\ attrib_1)\ \ (VERO) \\ TipoBase_1 &\sqsubseteq (\leq 1\ attrib_1)\ \ (VERO) \\ TipoBase_1 &\sqsubseteq (= 1\ attrib_2)\ \ (FALSO) \\ TipoBase_1 &\sqsubseteq (\leq 1\ attrib_2)\ \ (FALSO) \\ TipoBase_1 &\sqsubseteq (= 1\ attrib_3)\ \ (FALSO) \\ TipoBase_1 &\sqsubseteq (\leq 1\ attrib_3)\ \ (FALSO) \end{aligned}$$

$$\begin{aligned} TipoBase_2 &\sqsubseteq (= 1\ attrib_1)\ \ (FALSO) \\ TipoBase_2 &\sqsubseteq (\leq 1\ attrib_2)\ \ (FALSO) \end{aligned}$$

$$
\begin{aligned}
TipoBase_2 &\sqsubseteq (= 1\ attrib_2)\ \ (FALSO)\\
TipoBase_2 &\sqsubseteq (\leq 1\ attrib_2)\ \ (VERO)\\
TipoBase_2 &\sqsubseteq (= 1\ attrib_3)\ \ (FALSO)\\
TipoBase_2 &\sqsubseteq (\leq 1\ attrib_3)\ \ (FALSO)
\end{aligned}
$$

$$
\begin{aligned}
TipoBase_3 &\sqsubseteq (= 1\ attrib_1)\ \ (FALSO)\\
TipoBase_3 &\sqsubseteq (\leq 1\ attrib_2)\ \ (FALSO)\\
TipoBase_3 &\sqsubseteq (= 1\ attrib_2)\ \ (FALSO)\\
TipoBase_3 &\sqsubseteq (\leq 1\ attrib_2)\ \ (FALSO)\\
TipoBase_3 &\sqsubseteq (= 1\ attrib_3)\ \ (VERO)\\
TipoBase_3 &\sqsubseteq (\leq 1\ attrib_3)\ \ (VERO)
\end{aligned}
$$

$$
\begin{aligned}
TipoBase_4 &\sqsubseteq (= 1\ attrib_1)\ \ (VERO)\\
TipoBase_4 &\sqsubseteq (\leq 1\ attrib_1)\ \ (VERO)\\
TipoBase_4 &\sqsubseteq (= 1\ attrib_2)\ \ (FALSO)\\
TipoBase_4 &\sqsubseteq (\leq 1\ attrib_2)\ \ (VERO)\\
TipoBase_4 &\sqsubseteq (= 1\ attrib_3)\ \ (FALSO)\\
TipoBase_4 &\sqsubseteq (\leq 1\ attrib_3)\ \ (FALSO)
\end{aligned}
$$

Un metodo alternativo per esprimere, nella logica $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, gli attributi, sarebbe stato quello che mappa i nomi degli stessi in abstract features, così che la loro cardinalità massima è definita implicitamente uguale 1. In questo modo la traduzione di un attributo normale è uguale a quella già vista, ma quella di un attributo opzionale si riduce alla sola dichiarazione del tipo di riempitivo ammesso per il ruolo, espressa tramite l'operatore $\forall$:

$$TipoBase_1 \sqsubseteq (\forall attrib_1.tipo_1) \sqcap (= 1\ attrib_1)$$

$$
\begin{aligned}
TipoBase_2 &\sqsubseteq (\forall attrib_2.tipo_2) \\
TipoBase_3 &\sqsubseteq (\forall attrib_3.tipo_3) \sqcap (= 1\ attrib_3), \\
TipoBase_4 &\sqsubseteq TipoBase_1 \sqcap TipoBase_2;
\end{aligned}
$$

dove $attrib_1$, $attrib_2$ e $attrib_3$ rappresentano dei nomi di abstract feature.

Benchè questo metodo risulti più compatto e forse più leggibile non permette di stabilire se, nella descrizione di un record, un attributo è opzionale, o meno. Infatti, in questo caso, le verifiche di sussunzione precedenti danno come risultato:

$$
\begin{aligned}
TipoBase_1 &\sqsubseteq (= 1\ attrib_1)\ \ (VERO) \\
TipoBase_1 &\sqsubseteq (\leq 1\ attrib_1)\ \ (VERO) \\
TipoBase_1 &\sqsubseteq (= 1\ attrib_2)\ \ (FALSO) \\
TipoBase_1 &\sqsubseteq (\leq 1\ attrib_2)\ \ \textbf{(VERO)} \\
TipoBase_1 &\sqsubseteq (= 1\ attrib_3)\ \ (FALSO) \\
TipoBase_1 &\sqsubseteq (\leq 1\ attrib_3)\ \ \textbf{(VERO)}
\end{aligned}
$$

$$
\begin{aligned}
TipoBase_2 &\sqsubseteq (= 1\ attrib_1)\ \ (FALSO) \\
TipoBase_2 &\sqsubseteq (\leq 1\ attrib_1)\ \ \textbf{(VERO)} \\
TipoBase_2 &\sqsubseteq (= 1\ attrib_2)\ \ (FALSO) \\
TipoBase_2 &\sqsubseteq (\leq 1\ attrib_2)\ \ (VERO) \\
TipoBase_2 &\sqsubseteq (= 1\ attrib_3)\ \ (FALSO) \\
TipoBase_2 &\sqsubseteq (\leq 1\ attrib_3)\ \ \textbf{(VERO)}
\end{aligned}
$$

$$
\begin{aligned}
TipoBase_3 &\sqsubseteq (= 1\ attrib_1)\ \ (FALSO) \\
TipoBase_3 &\sqsubseteq (\leq 1\ attrib_1)\ \ \textbf{(VERO)} \\
TipoBase_3 &\sqsubseteq (= 1\ attrib_2)\ \ (FALSO) \\
TipoBase_3 &\sqsubseteq (\leq 1\ attrib_2)\ \ \textbf{(VERO)} \\
TipoBase_3 &\sqsubseteq (= 1\ attrib_3)\ \ (VERO)
\end{aligned}
$$

$$TipoBase_3 \sqsubseteq (\leq 1\ attrib_3)\ (VERO)$$

$$TipoBase_4 \sqsubseteq (= 1\ attrib_1)\ (VERO)$$
$$TipoBase_4 \sqsubseteq (\leq 1\ attrib_1)\ (VERO)$$
$$TipoBase_4 \sqsubseteq (= 1\ attrib_2)\ (FALSO)$$
$$TipoBase_4 \sqsubseteq (\leq 1\ attrib_2)\ (VERO)$$
$$TipoBase_4 \sqsubseteq (= 1\ attrib_3)\ (FALSO)$$
$$TipoBase_4 \sqsubseteq (\leq 1\ attrib_3)\ \textbf{(VERO)}$$

Dal confronto dei risultati ottenuti si evince che, se gli attributi sono mappati in abstract feature non è possibile determinare con certezza se essi sono dichiarati esplicitamente opzionali per una descrizione di concetto o meno. Il vantaggio derivante dall'utilizzo del primo metodo è quello che è possibile creare delle applicazioni che collegandosi al sistema di rappresentazione della conoscenza, in cui è presente la TBox contenente la traduzione dello schema OO, possono, tramite delle interrogazioni mirate cosi come mostrato in figura 6.3, ricevere tutte le informazioni che descrivono i record che formano le classi, in modo da poterle presentare all'utente, o per poter interagire con altre applicazioni che sfruttano tali informazioni.

Traduzione dei path

I path introdotti da $\mathcal{OLCD}$ rappresentano delle restrizioni d'intervallo; questo vuol dire che un generico path, $p\theta d$, vincola il tipo base a cui porta il percorso p ad non assumere un valore qualsiasi, ma lo obbliga ad essere in relazione θ con il tipo base d.
Consideriamo l'esempio:

$$Manager = \Delta[salary : Real] \sqcap (\Delta.salary > 5000.00),$$

in cui *Real* è uno dei tipi base e il path *(salary > 5000.00)* esprime che le istanze di *Manager* devono avere un *salary* superiore a *5000.00* .

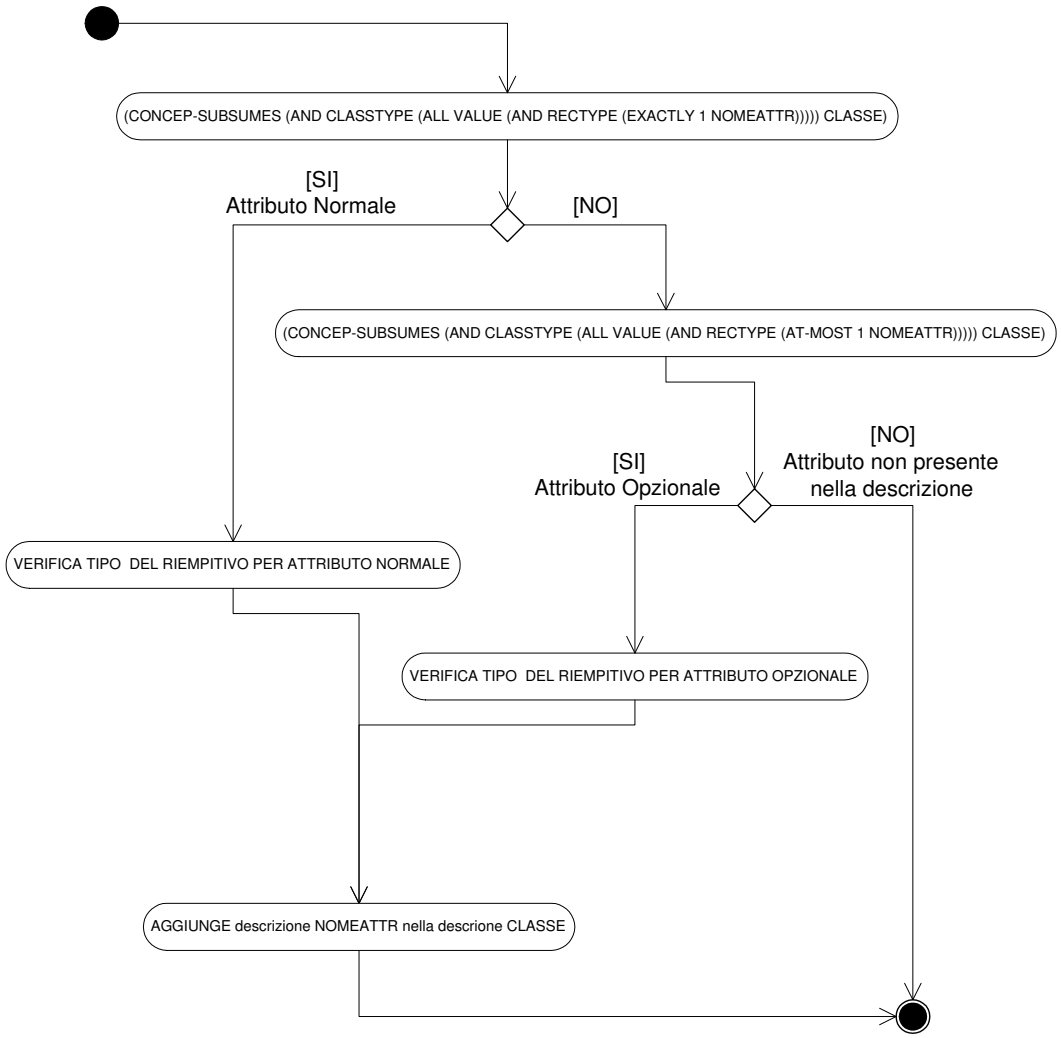

Figura 6.3: Activity Diagram: presenza attributi

Ricordando che l'operatore Δ è quello che permette di distinguere tra i tipi valore ed i tipi oggetto, si può notare che le due scritture:

$$(\Delta.salary > 5000.00),$$
$$\Delta[salary : (\epsilon > 5000.00)],$$

sono equivalenti dal punto di vista semantico, quindi la traduzione dell'esempio nel corrispondente concetto in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ è immediata:

$$\begin{aligned} Manager \quad \sqsubseteq \quad & ClassType \sqcap (\forall value.(RecType \sqcap (= 1\ salary) \sqcap (\forall salary.\Gamma(Real)))) \\ & \sqcap (\forall value(RecType \sqcap (\forall salary.\Gamma(Real_{>5000.00})), \end{aligned}$$

dove $\Gamma(Real_{>5000.00})$ si riconduce alla traduzione di un range secondo quanto mostrato in tabella 6.6.
Allo stesso modo è possibile tradurre path più complessi, come ad esempio:

$$(\Delta.attributo_1.attributo_2.\Delta.attributo_3\theta valore);$$

che equivale alla scrittura:

$$\Delta[attributo_1 : [attributo_2 : \Delta[attributo_3 : (\epsilon\theta valore)]]];$$

e che viene tradotto come segue:

$$\begin{aligned} & ClassType \sqcap (\forall value. \\ & (RecType \sqcap (= 1\ attributo_1) \sqcap (\forall attributo_1. \\ & (RecType \sqcap (= 1\ attributo_2) \sqcap (\forall attributo_2. \\ & (ClassType \sqcap (\forall value. \\ & (RecType \sqcap (= 1\ attributo_3) \sqcap (\forall attributo_3. \\ & \Gamma(tipo_{\theta valore}))))))))). \end{aligned}$$

Traduzione delle regole di integrità

In 5.2.3, abbiamo visto come le regole d'integrità vengano inserite nello schema grazie ai path e agli operatori $\neg$, $\sqcap$, $\sqcup$. In generale, il metodo utilizzato, inserisce direttamente nella

descrizione della classe intereressata, l'antecedente e il conseguente in modo tale che:

$$< classe > \quad = \quad < descrizione > \sqcap (\neg < antecedente > \sqcup < conseguente >).$$

La stessa regola può essere tradotta, in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, semplicemente creando due nuovi concetti: un concetto antecendente e un concetto conseguente tali che:

$$\begin{aligned} < rule\_antecedente > \quad &\equiv \quad < classe > \sqcap < antecedente > \\ < rule\_conseguente > \quad &\equiv \quad < conseguente > \\ < rule\_conseguente > \quad &\sqsubseteq \quad < rule\_antecedente > \end{aligned}$$

6.3 Schema di esempio espresso nella sintassi $\mathcal{OLCD}$

L'esempio di riferimento, considerato in questa sezione, rappresenta la struttura di una azienda. Dallo schema in tabella 6.7 apprendiamo che gli impiegati hanno un nome e percepiscono uno stipendio, che i dirigenti sono impiegati ed hanno un livello composto da una qualifica,che è una stringa, e da un parametro, che è un intero. I depositi hanno una denominazione, che può essere sia una stringa sia una struttura composta dal nome del deposito e dal suo indirizzo, inoltre gli articoli che possono contenere sono dei materiali. Questi ultimi sono contraddistinti da un nome e da un fattore di rischio. I reparti hanno una denominazione, che è una stringa, e a capo di ciascuno c'è un dirigente. I magazzini hanno le caratteristiche sia dei depositi che dei reparti. I fluidi hanno la caratteristica di essere viscosi. I solidi sono tutti i materiali, tranne i fluidi. L'immagazzinamento dei materiali è contraddistinto da un'urgenza.

6.3.1 Traduzione nella sintassi $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

Le tabelle, 6.8, 6.9 e 6.10, riportano la traduzione dell'esempio nella sintassi $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$, ottenuta applicando le regole illustrate in questo capitolo.

Level	=	[qualification: String, parameter: Integer]
Employee	=	Δ [name: String, salary: Integer]
Manager	=	Employee $\sqcap$ Δ [level: Level]
Repository	=	Δ [denomination: String $\sqcup$ [name: String, address: String], stock: $\{\text{Material}\}_{\forall}$]
Departement	=	Δ [denomination: String, manager: Manager]
Warehouse	=	Departement $\sqcap$ Repository
Material	=	Δ [name: String, risk: Integer]
Shipment	=	Δ [urgency: Integer, item: Material]
Fluid	=	Material $\sqcap$ Δ[viscosity: Real]
Solid	=	Material $\sqcap$ $\neg$Fluid

Tabella 6.7: Schema di esempio nella sintassi $\mathcal{OLCD}$

ClassType	$\sqsubseteq$	$(= 1 \text{ value})$
RecType	$\sqsubseteq$	$(\forall \text{ value}.\bot)$
AllSetType	$\sqsubseteq$	$(\forall \text{ value}.\bot) \sqcap (\neg \text{ RecType})$
RangeType	$\sqsubseteq$	$(\forall \text{ value}.\bot) \sqcap (\neg \text{ SetType}) \sqcap (\neg \text{ RecType})$
SomeSetType	$\sqsubseteq$	AllSetType
SomeSetType	$\sqsupseteq$	AllSetType $\sqcap$ $(\geq 1 \text{ member})$
RangeType	$\sqsupseteq$	$(\exists \text{ INTEGER}.\top_{\mathit{integer}}) \sqcup (\exists \text{ REAL}.\top_{\mathit{real}}) \sqcup (\exists \text{ STRING}.\top_{\mathit{string}})$
UnionType	$\sqsubseteq$	Classtype $\sqcup$ Rectype $\sqcup$ AllSetType $\sqcup$ RangeType

Tabella 6.8: Parte comune per tutte le traduzioni.

Level	$\equiv$	(RecType $\sqcap$
		(= 1 qualification) $\sqcap$ ($\forall$ qualification.($\exists$STRING.$\top_{string}$)) $\sqcap$
		(= 1 parameter) $\sqcap$ ($\forall$ parameter.($\exists$INTEGER.$\top_{integer}$)))
Employee	$\sqsubseteq$	ClassType $\sqcap$
		($\forall$value.(RecType $\sqcap$
		(= 1 name) $\sqcap$ ($\forall$ name.($\exists$STRING.$\top_{string}$)) $\sqcap$
		(= 1 salary) $\sqcap$ ($\forall$ salary.($\exists$REAL.$\top_{real}$))))
Manager	$\sqsubseteq$	ClassType $\sqcap$ Employee $\sqcap$
		($\forall$value.(RecType $\sqcap$
		(= 1 level) $\sqcap$ ($\forall$ level.Level))))
Repository	$\sqsubseteq$	ClassType $\sqcap$
		($\forall$value.(RecType $\sqcap$
		(= 1 denomination) $\sqcap$ ($\forall$ denomination.(($\exists$STRING.$\top_{real}$) $\sqcup$
		(RecType $\sqcap$
		(= 1 name) $\sqcap$ ($\forall$ name.($\exists$STRING.$\top_{string}$)) $\sqcap$
		(= 1 address) $\sqcap$ ($\forall$ address.($\exists$STRING.$\top_{string}$)))) $\sqcap$
		(= 1 stock) $\sqcap$ ($\forall$ stock.(AllSetType $\sqcap$ ($\forall$ member.Material)))))
Departement	$\sqsubseteq$	ClassType $\sqcap$
		($\forall$value.(RecType $\sqcap$
		(= 1 denomination) $\sqcap$ ($\forall$ denomination.($\exists$STRING.$\top_{string}$))$\sqcap$
		(= 1 manager) $\sqcap$ ($\forall$ manager.Manager)))
Warehouse	$\sqsubseteq$	Departement $\sqcap$ Repository
Material	$\sqsubseteq$	ClassType $\sqcap$
		($\forall$ value.(RecType $\sqcap$
		(= 1 name) $\sqcap$ ($\forall$ name.($\exists$STRING.$\top_{string}$)) $\sqcap$
		(= 1 risk) $\sqcap$ ($\forall$ risk.($\exists$INTEGER.$\top_{integer}$))))

Tabella 6.9: Traduzione dello schema in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

Shipment	$\sqsubseteq$	ClassType $\sqcap$
		($\forall$value.(RecType $\sqcap$
		(= 1 urgency) $\sqcap$ ($\forall$ urgency.($\exists$INTEGER.$\top_{integer}$)) $\sqcap$
		(= 1 item) $\sqcap$ ($\forall$ item.Material)))
Fluid	$\sqsubseteq$	ClassType $\sqcap$ Material $\sqcap$
		($\forall$value.(RecType $\sqcap$
		(= 1 viscosity) $\sqcap$ ($\forall$ viscosity.($\exists$REAL.$\top_{real}$)))))
Solid	$\sqsubseteq$	Material $\sqcap$ $\neg$ Fluid

Tabella 6.10: Traduzione dello schema in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$

6.4 Test e validazione dello schema

Uno dei problemi principali che il progettista di una base di dati deve affrontare è quello della consistenza delle classi introdotte nello schema. Infatti, molti modelli e linguaggi di definizione dei dati sono sufficientemente espressivi da permettere la rappresentazione di classi inconsistenti, cioè classi che non potranno contenere alcun oggetto della base di dati. Le prove effettuate con RACER, su schemi tradotti seguendo le regole mostrate in questo capitolo, hanno dato esito positivo, nel senso che hanno dimostrato che RACER è in grado di effettuare la validazione dello schema e di rilevare le eventuali incosistenze. In relazione all'esempio 6.7 è riportato il grafico che esprime la gerarchia delle classi; la figura 6.4 mostra in grigio la struttura di base ed in rosso le classi introdotte con l'esempio.

6.4.1 Esempi di inconsistenza

Presentiamo ora degli esempi di inconsistenza che possono presentarsi in uno schema, e che sono stati testati mediante RACER, per valutarne le effettive capacità di ragionamento sul modello proposto. Sempre con riferimento all'esempio 6.7 sono state introdotte le

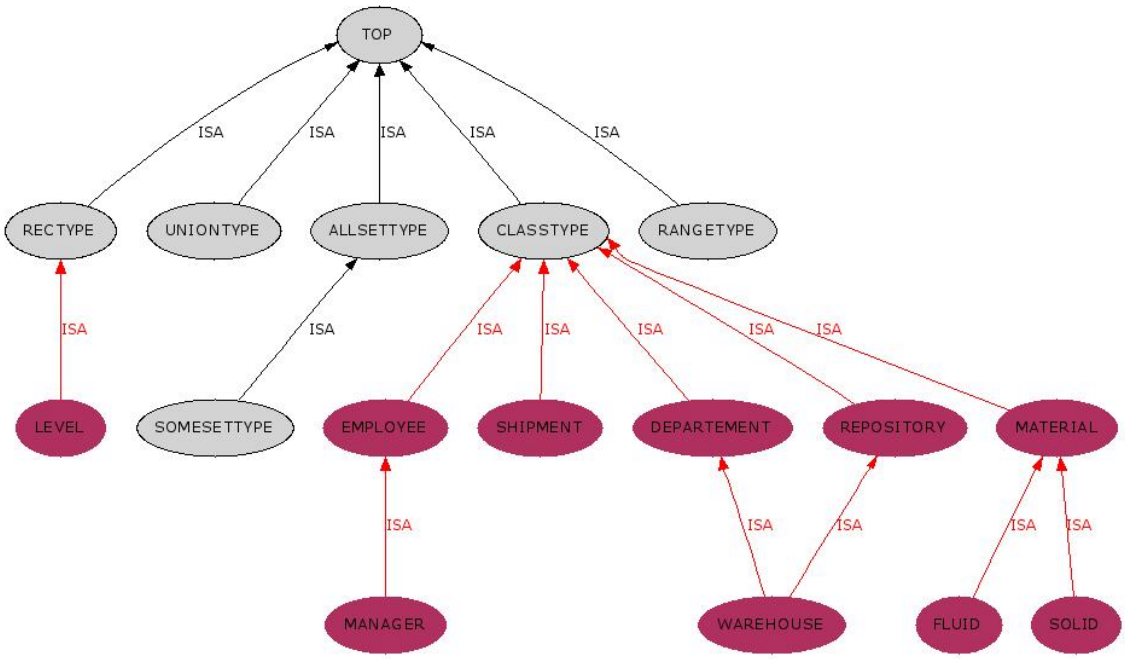

Figura 6.4: Tassonomia relativa all'esempio 6.7

classi:

$$
\begin{aligned}
Storage &= Warehouse \sqcap \Delta[denomination : [name : String, \\
&\quad address : String]] \\
AWarehouse &= Warehouse \sqcap \Delta[stock : \{Fluid\}_{\forall}] \\
BWarehouse &= AWarehouse \sqcap \Delta[stock : \{Solid\}_{\exists}]
\end{aligned}
$$

La classe Storage eredita, da Warehouse, l'attributo denomination, che è di tipo stringa, e lo ridefinisce come una tupla, ma questi sono due tipi incompatibili; questo tipo d'inconsistenza deriva dal fatto che si ha lo stesso attributo vincolato a strutture differenti. La classe AWarehouse può immagazzinare solo oggetti che siano fluidi, il chè non da luogo ad inconsistenze, che però sorgono con nella classe BWarehouse, che stabilisce che tra gli articolo immagazzinati ve ne sia almeno uno di tipo solido, mentre come ricordiamo Solid e Fluid sono state definite disgiunte. Un altro test ha modificato la classe Shipment, dello schema di partenza, introducendo la regola d'integrità che "*se il rischio del mate-*

riale è maggiore di 3 allora la sua urgenza dev'essere superiore di 10"; la sua traduzione in $\mathcal{OLCD}$ è:

$$\begin{aligned} Shipment &= \Delta[urgency : Integer, item : Material] \sqcap \\ &\quad (\neg(item.risk > 3) \sqcup (urgency > 10)) \end{aligned}$$

Inoltre sono state introdotte le classi:

$$\begin{aligned} DangerMaterial &= Material \sqcap (risk > 5) \\ SlowDangerShipment &= Shipment \sqcap \Delta[item : DangerMaterial] \sqcap \\ &\quad (urgency < 3) \end{aligned}$$

La classe SlowDangerShipment è incosistente perchè ciò che immagazzina è un Danger-Material, che ha un rischio superiore a 5 e che implica una urgenza maggiore di 10, e non inferiore di 3 come compare nella descrizione. Un altro esempio di inconsistenza, che deriva dall'intersezione di due intervalli disgiunti, è mostrato nell'esempio seguente:

$$\begin{aligned} HeavyShipment &= Shipment \sqcap (tonnage > 50000) \\ LightShipment &= Shipment \sqcap (tonnage < 20000) \\ NormalShipment &= HeavyShipment \sqcap LightShipment \end{aligned}$$

La classe NormalShipment eredita l'attributo tonnage dalle classi HeavyShipment e LightShipment, quindi il suo dominio diventa l'intersezione degli intervalli specificati nelle classi padre, che essendo disgiunti generano un'inconsistenza. Sono state introdotte anche le classi virtuali:

$$\begin{aligned} SuperFluid &= Fluid \sqcap (viscosity < 10.0) \\ TopFluid &= Fluid \sqcap \neg SuperFluid \sqcap (viscosity < 3.0) \end{aligned}$$

dove la classe virtuale TopFluid è incosistente, perchè benchè abbia una viscosità inferiore a 3, si impone che non appartenga a SuperFluid, al contrario di quanto dovrebbe essere.

6.5 Schemi in linguaggio ODL$_{I^3}$

Finora abbiamo visto come sia possibile passare da uno schema espresso in $\mathcal{OLCD}$ ad uno equivalente in $\mathcal{ALCQHI}_{\mathcal{R}+}(\mathcal{D})^-$ e da questi ottenere il file di input in RLI 3.3. In questa sezione, utilizzando le regole di traduzione introdotte in precedenza, vogliamo implementare un traduttore dal linguaggio **ODL**$_{I^3}$ a quello di RACER, più precisamente dal sottoinsieme del linguaggio **ODL**$_{I^3}$ che trova corrispondenza in $\mathcal{OLCD}$, indicato con **ODL**$_{I^3}$-$\mathcal{OLCD}$, al linguaggio RLI.

L'implementazione del traduttore dal linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ al linguaggio RLI, che chiameremo **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** , deve considerare lo stato attuale (ottobre 2004) dello sviluppo del sistema MOMIS e quindi, in particolare, che:

1. non esiste l'implementazione di un traduttore da **ODL**$_{I^3}$-$\mathcal{OLCD}$ a $\mathcal{OLCD}$, quindi il punto di partenza per l'implementazione di **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** deve essere necessariamente il linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$;

2. il linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ non è certamente minimale, ovvero una stessa descrizione di classe si può ottenere in diversi modi equivalenti; ad esempio:

```
interface C
    {
        a: C1;
    }
union
    {
        a: C2;
    };
```

oppure:

```
interface C
    {
        a: C1 union C2
    };
```

3. le regole sintattiche del linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ sono attualmente in una versione non definitiva.

Di conseguenza, in questa tesi, si è deciso di definire una versione "canonica" del linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$ sul quale effettuare il traduttore **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER**; il termine "canonico" deriva, in particolare, dal fatto che si considera un solo livello di innestamento e che le strutture complesse (con più livelli di innestamento) vengono definite introducendo nomi di classi e di tipi.

Naturalmente la traduzione da **ODL**$_{I^3}$-$\mathcal{OLCD}$ ad RLI di una espressione non canonica potrà essere implementata in due passi:

1. trasformando in forma canonica l'espressione;
2. applicando il traduttore implementato in questa tesi.

Con riferimento all'esempio precedente, l'espressione non canonica:

```
interface C
    {
        a: C1;
    }
union
    {
        a: C2;
    };
```

potrà essere trasformata in **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER**:

interface C
{
a: C1 union C2
};

e quindi trasformata in RLI.

Le seguenti regole sintattiche definiscono il linguaggio canonico **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER**:

<interface_dcl> → <interface_header> {[<interface_body>]} <dot>

<dot> → **;** |
.

<interface_header> → <interface_type> <identifier> <inheritance_spec>

<interface_type>: → **interface** |
view |
typedef

<inheritance_spec> → **:** [**not**] <scoped_name> [**,** <inheritance_spec>]

<interface_body> → **attribute** [<domain_type>] <attribute_name> [*****]**;**
[<interface_body>]

<domain_type> → <base_type> |
<range_expr> |
<set_expr> |
<union_expr> |
[**not**] <identifier>

<base_type> → **string** |
integer |
real

<range_expr> → **range** { <number>, <number> }

<number> → [-]<cifra>[**.**]<cifra> |
[-]**inf**

<cifra> → **0..9** [<cifra>]

<set_expr> → <set_type> **<** <element> **>**

<set_type> → **set** |
exists

<union_expr> → **union <** <element> **,** <element>
[**,** <element>] **>**

<element> → <identifier> |
not <identifier> |
<base_type> |
<range_expr>

Nell'attuale implementazione, **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** non introduce le "regole" e i "path", dato che non aggiungono niente di rilevante alla traduzione; infatti le "regole" corrispondono a semplici assiomi di inclusione, mentre i "path", in $\mathcal{OLCD}$, sono esprimibili in modo equivalente tramite altri costruttori di tipo.

6.5.1 Regole di traduzione da ODL$_{I^3}$-$\mathcal{OLCD}$/RACER a RLI

Le regole di traduzione dal linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** ad RLI vengono ora illustrate tramite esempi significativi.

- Traduzione di Classi e View.

  La traduzione delle classi e delle viste avviene seguendo quanto detto nel paragrafo precedente. Le classi vengono introdotte nello schema mediante la funzione IMPLIES, mentre le viste tramite la funzione EQUIVALENT. Per ciascun nuovo attributo specificato è neccessario inserire la sua dichiarazione.

  Ad esempio, consideriamo la classe Employee:

```
interface Employee
{
    attribute String name;
    attribute Integer salary;
};
```

  La sua traduzione è:

```
(DEFINE-PRIMITIVE-ROLE NAME)
(DEFINE-PRIMITIVE-ROLE SALARY)

(IMPLIES EMPLOYEE (AND CLASSTYPE
    (ALL VALUE (AND RECTYPE
        (EXACTLY 1 NAME)(ALL NAME (A STRING))
        (EXACTLY 1 SALARY) (ALL RISK (A INTEGER)) ))))
```

- Tipi Record definiti dall'utente.

  I tipi record vengono tradotti mediante l'utilizzo della funzione EQUIVALENT, in modo tale che la loro radice sia il concetto RecType. Nell'esempio la classe Level:

```
typedef Level
{
    attribute String qualification;
```

```
    attribute Integer parameter;
};
```

viene tradotta in:

```
(DEFINE-PRIMITIVE-ROLE QUALIFICATION)
(DEFINE-PRIMITIVE-ROLE PARAMETER)

(IMPLIES LEVEL (AND RECTYPE
    (EXACTLY 1 QUALIFICATION) (ALL QUALIFICATION (A STRING))
    (EXACTLY 1 PARAMETER) (ALL PARAMETER (A INTEGER)) ))
```

- Relazioni ISA

  Le relazioni ISA vengono espresse intersecando la classe ereditata con il concetto ClassType. Nel caso della classe Manager, che eredita da Employee abbiamo:

```
interface Manager: Employee
{
    attribute Level level;
};
```

a cui corrisponde il codice in RLI:

```
(DEFINE-PRIMITIVE-ROLE LEVEL)

(IMPLIES MANAGER (AND CLASSTYPE EMPLOYEE
    (ALL VALUE (AND RECTYPE
        (EXACTLY 1 LEVEL) (ALL LEVEL LEVEL) ))))
```

- Operatore not. L'operatore di complemento è applicabile solamente alle classi e alle view, escludendo quindi i tipi definiti tramite il typedef. La classe Solid, si può esprimere mediante la sintassi **ODL**$_{I^3}$:

```
interface Solid: Material, not Fluid
{
};
```

alla quale corrisponde la traduzione:

```
(IMPLIES SOLID (AND CLASSTYPE MATERIAL (NOT FLUID)))
```

- Operatore set

Permette di esprimere il tipo insieme universale. L'esempio mostra la classe AWarehouse:

```
interface AWarehouse: Warehouse
{
    attribute set<Fluid> stock;
};
```

la cui traduzione in RACER genera il seguente codice:

```
(DEFINE-PRIMITIVE-ROLE STOCK)

(IMPLIES AWAREHOUSE (AND CLASSTYPE WAREHOUSE
    (ALL VALUE (AND RECTYPE
        (EXACTLY 1 STOCK)
        (ALL STOCK (AND ALLSETTYPE (ALL MEMBER FLUID))) ))))
```

- Operatore exists. É utilizzato per indicare l'insieme esistenziale. Ad esempio:

```
interface BWarehouse: Warehouse
{
    attribute exists<Solid> stock;
};
```

che in RACER si trasforma in:

```
(DEFINE-PRIMITIVE-ROLE STOCK)

(IMPLIES AWAREHOUSE (AND CLASSTYPE WAREHOUSE
    (ALL VALUE (AND RECTYPE
        (EXACTLY 1 STOCK)
        (ALL STOCK (AND SOMESETTYPE (SOME MEMBER SOLID))) ))))
```

- Operatore union. Permette di specificare l'unione di due o più tipi e viene utilizzato, in genere, per esprimere un'alternativa per il tipo di riempitivo di un attributo.

```
interface Repository
{
    attribute union<String, Address> denomination;
    attribute set<Material> stock;
};

(DEFINE-PRIMITIVE-ROLE DENOMINATION)

(IMPLIES REPOSITORY (AND CLASSTYPE
    (ALL VALUE (AND RECTYPE
        (EXACTLY 1 DENOMINATION)
```

(ALL DENOMINATION (AND UNIONTYPE
(OR (A STRING) ADDRESS)))
. . .))))

6.5.2 Esempio completo in ODL$_{I^3}$

Si riporta, in linguaggio **ODL**$_{I^3}$, l'esempio introdotto in 6.3, esteso con gli esempi di incosistenza mostrati in 6.4.1.

```
typedef Level
{
    attribute String qualification;
    attribute Integer parameter;
};

typedef Denomination
{
    attribute String name;
    attribute String address;
};

interface Employee
{
    attribute String name;
    attribute Integer salary;
};

interface Manager: Employee
{
    attribute Level level;
```

```
};

interface Repository
{
    attribute union<String, Denomination> denomination;
    attribute set<Material> stock;
};

interface Departement
{
    attribute String denomination;
    attribute Manager manager;
};

interface Warehouse: Departement, Repository
{};

interface Material
{
    attribute String name;
    attribute Integer risk;
};

interface Shipment
{
    attribute Integer urgency ;
    attribute Material item;
};
```

```
interface Fluid: Material
{
    attribute Real viscosity;
};

interface Solid: Material, not Fluid
{};

interface Storage: Warehouse
{
    attribute Denomination denomination;
};

interface AWarehouse: Warehouse
{
    attribute set<Fluid> stock;
};

interface BWarehouse: AWarehouse
{
    attribute exists<Solid> stock;
};

interface DangerMaterial: Material
{
    attribute range{6, +inf} risk;
};

interface SlowDangerShipment: Shipment
```

```
{
    attribute DangerMaterial item;
    attribute range{-inf, 2} urgency;
};

interface HeavyShipment: Shipment
{
    attribute range{50001, inf} tonnage;
};

interface LightShipment: Shipment
{
    attribute range{-inf, 19999} tonnage;
};

interface NormalShipment: HeavyShipment, LightShipment
{};

view SuperFluid: Fluid
{
    attribute range{-inf, 10.0} viscosity;
};

view TopFluid: SuperFluid
{
    attribute range{-inf, 3.0} viscosity;
};
```

6.5.3 Traduzione dell'esempio in RLI

La traduzione dell'esempio precedente è riportata di seguito.

```
(IN-TBOX COMPANY)

(DEFINE-PRIMITIVE-ROLE VALUE)
(DEFINE-PRIMITIVE-ROLE MEMBER)

(DEFINE-CONCRETE-DOMAIN-ATTRIBUTE STRING :TYPE STRING)
(DEFINE-CONCRETE-DOMAIN-ATTRIBUTE INTEGER :TYPE INTEGER)
(DEFINE-CONCRETE-DOMAIN-ATTRIBUTE REAL :TYPE REAL)

(IMPLIES CLASSTYPE (EXACTLY 1 VALUE))
(IMPLIES RECTYPE (ALL VALUE BOTTOM))
(IMPLIES ALLSETTYPE (AND(ALL VALUE BOTTOM) (NOT RECTYPE)))
(IMPLIES RANGETYPE (AND(ALL VALUE BOTTOM)
    (NOT ALLSETTYPE) (NOT RECTYPE)))
(IMPLIES (OR (A INTEGER) (A STRING) (A REAL)) RANGETYPE)
(IMPLIES SOMESETTYPE ALLSETTYPE)
(IMPLIES (AND (AT-LEAST 1 MEMBER) ALLSETTYPE) SOMESETTYPE)
(IMPLIES UNIONTYPE (OR CLASSTYPE RECTYPE
    ALLSETTYPE RANGETYPE))

(DEFINE-PRIMITIVE-ROLE LEVEL)
(DEFINE-PRIMITIVE-ROLE NAME)
(DEFINE-PRIMITIVE-ROLE SALARY)
(DEFINE-PRIMITIVE-ROLE QUALIFICATION)
(DEFINE-PRIMITIVE-ROLE PARAMETER)
(DEFINE-PRIMITIVE-ROLE DENOMINATION)
```

```
(DEFINE-PRIMITIVE-ROLE STOCK)
(DEFINE-PRIMITIVE-ROLE ADDRESS)
(DEFINE-PRIMITIVE-ROLE MANAGER)
(DEFINE-PRIMITIVE-ROLE RISK)
(DEFINE-PRIMITIVE-ROLE VISCOSITY)
(DEFINE-PRIMITIVE-ROLE URGENCY)
(DEFINE-PRIMITIVE-ROLE ITEM)
(DEFINE-PRIMITIVE-ROLE WORKS)

(EQUIVALENT LEVEL (AND RECTYPE
    (EXACTLY 1 QUALIFICATION) (ALL QUALIFICATION (A STRING))
    (EXACTLY 1 PARAMETER) (ALL PARAMETER (A INTEGER))))

(EQUIVALENT DENOMINATION (AND RECTYPE
    (EXACTLY 1 NAME) (ALL NAME (A STRING))
    (EXACTLY 1 ADDRESS) (ALL ADDRESS (A STRING))))

(IMPLIES EMPLOYEE (AND CLASSTYPE
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 NAME) (ALL NAME (A STRING))
    (AT-MOST 1 SALARY) (ALL SALARY ( A REAL))))))

(IMPLIES MANAGER (AND CLASSTYPE EMPLOYEE
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 LEVEL) (ALL LEVEL LEVEL)))))

(IMPLIES REPOSITORY (AND CLASSTYPE
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 DENOMINATION) (ALL DENOMINATION
```

```
        (AND UNIONTYPE (OR (A STRING) DENOMINATION)))
    (EXACTLY 1 STOCK) (ALL STOCK
        (AND ALLSETTYPE (ALL MEMBER MATERIAL)))))))

(IMPLIES DEPARTEMENT (AND CLASSTYPE
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 DENOMINATION) (ALL DENOMINATION (A STRING))
    (EXACTLY 1 MANAGER) (ALL MANAGER MANAGER) ))))

(IMPLIES WAREHOUSE (AND CLASSTYPE DEPARTEMENT REPOSITORY))

(IMPLIES MATERIAL (AND CLASSTYPE
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 NAME) (ALL NAME (A STRING))
    (EXACTLY 1 RISK) (ALL RISK (A INTEGER))))))

(IMPLIES FLUID (AND CLASSTYPE MATERIAL
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 VISCOSITY) (ALL VISCOSITY (A REAL))))))

(IMPLIES SOLID (AND CLASSTYPE MATERIAL (NOT FLUID)))

(IMPLIES SHIPMENT (AND CLASSTYPE
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 URGENCY) (ALL URGENCY (A INTEGER))
    (EXACTLY 1 ITEM) (ALL ITEM MATERIAL)))))

(IMPLIES STORAGE (AND CLASSTYPE WAREHOUSE
    (ALL VALUE (AND RECTYPE
```

```
    (EXACTLY 1 DENOMINATION) (ALL DENOMINATION DENOMINATION)))))

(IMPLIES AWAREHOUSE (AND CLASSTYPE
    (ALL VALUE(AND RECTYPE
    (EXACTLY 1 STOCK) (ALL STOCK
        (AND ALLSETTYPE (ALL MEMBER FLUID)))))))

(IMPLIES BWAREHOUSE (AND CLASSTYPE AWAREHOUSE
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 STOCK) (ALL STOCK
        (AND SOMESETTYPE (SOME MEMBER SOLID)))))))

(IMPLIES DANGERMATERIAL (AND CLASSTYPE MATERIAL
    (ALL VALUE (AND RECTYPE
    (ALL RISK (MIN INTEGER 5))))))

(IMPLIES SLOWDANGERSHIPMENT (AND CLASSTYPE SHIPMENT
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 ITEM) (ALL ITEM DANGERMATERIAL)
    (ALL URGENCY (MAX INTEGER 5))))))

(EQUIVALENT SUPERFLUID (AND CLASSTYPE FLUID
    (ALL VALUE (AND RECTYPE
    (EXACTLY 1 VISCOSITY) (ALL VISCOSITY (<= REAL 10)) ))))

(EQUIVALENT TOPFLUID (AND CLASSTYPE FLUID (NOT SUPERFLUID)
    (ALL VALUE (AND RECTYPE
    (ALL VISCOSITY (< REAL 3)) ))))
```

Capitolo 7

Conoscenza estensionale di un OODB in RACER

7.1 Rappresentazione delle istanze del modello

Come visto nel capitolo 3, RACER supporta l'ABox, permettendo quindi di fornire una rappresentazione estensionale della conoscenza. Partendo dal fatto che l'estensione delle classi primitive è definita solo dall'utente, nel senso che è l'utente che deve specificare la classe base di appartenenza per ciascuna istanza inserita, e tenendo presente che RACER adotta l'OWA, in questo capitolo vogliamo discutere come rappresentare le istanze di uno schema OODB in RACER in modo da ottenere "gli stessi risultati", che si ottengono nell'OODB dove viene utilizzata la CWA; Informalmente, si vuole "simulare" la CWA in RACER in modo da ottenere un comportamento simile a quello di un OODB; intuitivamente questo significa che si devono completare le informazioni "importanti" mancanti, in modo tale da eliminare quell'assenza di informazioni che altrimenti genererebbe delle "lacks of knowledge". Dopo aver individuato come effettuare tale completamento, allo scopo di effettuarlo in modo automatico, verrà proposto un componente software, che consente (tramite un'interfaccia) di inserire istanze e di completarle automaticamente. Prendendo spunto da [BB03], introduciamo un esempio generale per spiegare il problema

in modo intuitivo, a prescindere dalle considerazioni e dai risultati del capitolo precedente. Supponiamo di avere uno schema OODB che definisce due tipi base:

$$\begin{aligned} \sigma(Book) &= [\, author :\ \{Person\}_{\forall}\,]\ ; \\ \sigma(EnglishMan) &= Person\ ; \end{aligned}$$

Introduciamo i seguenti valori:

$$\begin{aligned} \nu_1 &\rightarrow [\, author :\ \{V_{11}, V_{12}\}\,]\ ; \\ \nu_2 &\rightarrow [\, author :\ \{V_{13}, V_{14}\}\,]\ ; \end{aligned}$$

e consideriamo le istanze:

$$\begin{aligned} I[\, Book\,] &\rightarrow \{\nu_1, \nu_2\}; \\ I[\, Person\,] &\rightarrow \{V_{11}, V_{12},\ V_{13},\ V_{14}\}; \\ I[\, EnglishMan\,] &\rightarrow [\, author :\ \{V_{11},\ V_{12},\ V_{14}\}. \end{aligned}$$

Ora consideriamo l'interrogazione (tipo virtuale):

$$\sigma(Query) = Book \sqcap [\, author :\ \{EnglishMan\}_{\forall}\,]\ ;$$

la cui istanza (risposta all'interrogazione) è:

$$I[\, Query\,] \rightarrow \{\nu_1\};$$

Adesso, dopo aver rappresentato il precedente schema (tipi base e virtuali) OODB in RLI come segue:

(IMPLIES BOOK

(ALL AUTHOR PERSON))
(IMPLIES ENGLISHMAN PERSON)
(EQUIVALENT QUERY (AND BOOK
(ALL AUTHOR ENGLISHMAN)))

vogliamo rappresentare le istanze dell'ABox, $\mathcal{A}$, in modo tale che i problemi di instance check:

$$\mathcal{A} \models QUERY(\nu_1)?$$
$$\mathcal{A} \models QUERY(\nu_2)?$$

diano, rispettivamente, le risposte vero e falso, ovvero si ottenga "lo stesso risultato" ottenuto nell'OODB. Una prima rappresentazione dell'ABox potrebbe essere data dalle seguenti asserzioni:

(INSTANCE ν_1 BOOK)
(RELATED ν_1 V_{11} AUTHOR)
(RELATED ν_1 V_{12} AUTHOR)

(INSTANCE ν_2 BOOK)
(RELATED ν_2 V_{13} AUTHOR)
(RELATED ν_2 V_{14} AUTHOR)

(INSTANCE V_{11} ENGLISHMAN)
(INSTANCE V_{12} ENGLISHMAN)
(INSTANCE V_{14} ENGLISHMAN)
(INSTANCE V_{13} PERSON)

Queste asserzioni non sono sufficienti a garantire le risposte che cerchiamo, infatti, i problemi di instance check:

$$\mathcal{A} \models QUERY(\nu_1)? \; FALSO$$

$$\mathcal{A} \models QUERY(\nu_2)?\ FALSO$$

non restituiscono i risultati attesi. Per ottenere la CWA, è necessario specificare la "chiusura" sul numero di riempitivi effettivamente presenti per il ruolo "AUTHOR" sulle istanze ν_1 ed ν_2; in altre parole bisogna esplicitare il fatto che non esistono altri riempitivi di "AUTHOR" per le istanze ν_1 ed ν_2 oltre a quelli definiti. Questo si può esprimere dicendo che ν_1 ed ν_1 sono dei *"BOOK che hanno esattamente due riempitivi per il ruolo AUTHOR"*:

(INSTANCE ν_1 (AND BOOK
    (EXACTLY 2 AUTHOR)))

(INSTANCE ν_2 (AND BOOK
    (EXACTLY 2 AUTHOR)))

. . .. . .

(INSTANCE V_{13} PERSON)

Un'altra faccia dello stesso problema è il fatto che non viene messa in discussione l'appartenenza di un individuo ad un concetto. Questo vuol dire che è possibile asserire l'appartenenza di un individuo ad un determinato concetto senza specificarne le proprietà. Ridefiniamo il tipo base *Book* nel modo seguente:

$$\sigma(Book) = [\ isbnNr : ISBN\ ];$$

e consideriamo i nuovi valori:

$$\nu_1 \rightarrow [\ isbnNr :\ V_{21}]\ ;$$
$$\nu_2 \rightarrow [\ ]\ .$$

dove:

$$I[\ ISBN\ ] \rightarrow \{\ V_{21}\ \};$$

Viste le definizioni cui sopra, una condizione necessaria affinchè un'istanza appartenga a *Book* è che abbia un *isbnNr* di tipo *Isbn*, quindi ci aspettiamo che ν_2 non sia un'istanza valida per *Book*, in quanto, per essa, non viene specificata nemmeno la presenza dell'attributo *Isbn*. Traducendo, in RLI, lo schema d'esempio:

(IMPLIES BOOK (AND
    (ALL ISBNNR ISBN)
    (EXACTLY 1 ISBNNR)))

e le istanze come segue:

(INSTANCE ν_1 BOOK)
(INSTANCE V_{21} ISBN)
(RELATED ν_1 V_{21} ISBNNR)

(INSTANCE ν_2 BOOK)

si ottiene il risultato che entrambe le istanze: ν_1, ν_2, sono considerate valide ed entrambe appartengono ad $I[\ BOOK\ ]$. Anche in questo caso, come nel caso precedente, per ottenere la CWA è necessario specificare la presenza, o meno, degli attributi tramite la chiusura sul numero di riempitivi: Notando che dal punto di vista semantico le due scritture:

$$(=\ 0\ isbnnr) \quad ,$$
$$(\forall\ isbnnr.\bot) \quad ,$$

sono equivalenti, quindi si può scrivere:

(INSTANCE ν_1 (AND BOOK

(EXACTLY 1 ISBNNR))
(INSTANCE V_{21} ISBN)
(RELATED ν_1 V_{21} ISBNNR)

(INSTANCE ν_2 (AND BOOK
(ALL ISBNNR BOTTOM))

Al meglio della mia conoscenza questi problemi, anche se discussi in modo intuitivo in [BCM+03], non sono stati affrontati in maniera esaustiva e formale in alcun articolo presente in letteratura. Pertanto nel seguito viene presentata, in modo informale ma completo, una soluzione al problema.

7.1.1 Esempio di ABox

Per come è stato sviluppato il modello di rappresentazione dello schema, un generico oggetto, istanza di una classe base, è rappresentato da un object-identifier (oid) associato alla sua descrizione tramite il ruolo *value*. Con riferimento alla TBox dell'esempio 6.3, supponiamo di voler inserire nell'ABox l'oggetto:

$$OBJ_1 \quad \rightarrow \quad [name : Liz, salary : 50000];$$

come istanza di *Employee*. La prima asserzione da fare, nel linguaggio RLI 3.3, è:

(INSTANCE OBJ_1 EMPLOYEE),

che crea l'oid OBJ_1 appartenente al concetto *Employee*. In accordo con l'OWA, per il sistema l'individuo OBJ_1 è un *Employee*, anche se non ne abbiamo ancora specificato la descrizione.

É necessario quindi creare un individuo, *REC_*OBJ_1 di *RecType*, per poter descrivere il record associato all'oid.

Dato che, nel caso in esame, la TBox 6.3 definisce i ruoli:

{ LEVEL, NAME, SALARY, QUALIFICATION, PARAMETER, DENOMINATION, STOCK, ADDRESS, MANAGER, RISK, VISCOSITY, URGENCY, ITEM, WORKS, TONNAGE };

e che nel record che vogliamo descrivere compaiono solamente gli attributi *NAME* ed *SALARY*, in base alle considerazioni precedenti, è necessario fare la seguente asserzione:

(INSTANCE *REC*_OBJ_1
(AND RECTYPE
(EXACTLY 1 NAME)
(EXACTLY 1 SALARY)
(ALL LEVEL BOTTOM)
(ALL QUALIFICATION BOTTOM)
...
(ALL WORKS BOTTOM)
(ALL TONNAGE BOTTOM)))

Con l'istruzione:

(RELATED
OBJ_1 *REC*_OBJ_1 VALUE),

asseriamo che *REC*_OBJ_1 è il riempitivo del ruolo *value* di OBJ_1, legando così l'oid alla propria descrizione.

Dato che per l'attributo *name* viene fornito in ingresso un valore di tipo stringa, è necessario generare un individuo *NAME_REC*_OBJ_1 associato, tramite la concrete feature STRING, all'oggetto[1] *SNAME_REC*_OBJ_1:

(CONSTRAINED
*NAME_REC*_OBJ_1 *SNAME_REC*_OBJ_1 STRING),

[1] in questo caso il termine oggetto fa riferimento alla terminologia introdotta con RACER nel cap 4.

alla quale deve corrispondere il valore stringa "*Liz*":

(CONSTRAINTS
    (STRING= *SNAME_REC_*OBJ_1 |Liz|)).

Infine è necessario asserire che *REC_*OBJ_1 è collegato a *NAME_REC_*OBJ_1 tramite il ruolo *name*:

(RELATED
    *REC_*OBJ_1 *NAME_REC_*OBJ_1 NAME).

Lo stesso procedimento va iterato per l'attributo *salary*:

(CONSTRAINED
    *SALARY_REC_*OBJ_1 *ISALARY_REC_*OBJ_1 INTEGER)
(CONSTRAINTS
    (EQUAL *ISALARY_REC_*OBJ_1 5000))
(RELATED
    *REC_*OBJ_1 *SALARY_REC_*OBJ_1 SALARY)

É molto importante indicare il tipo: string, real o integer, nelle asserzioni riguardanti le concrete features (constrained), e conseguentemente utilizzare l'operatore adatto nelle asserzioni dei predicati sui domini concreti (constraints). Il tipo dev'essere determinato, o determinabile, dal record che vogliamo inserire. La figura 7.1 rappresenta in forma grafica l'oggetto OBJ_1.

Con lo stesso metodo possiamo definire le istanze: OBJ_2, OBJ_3 della classe *Solid* e OBJ_4 della classe *Fluid* :

$$\begin{aligned} OBJ_2 &\rightarrow [name : steel, risk : 1]; \\ OBJ_3 &\rightarrow [name : iron, risk : 2]; \\ OBJ_4 &\rightarrow [name : water, risk : 5, viscosity : 4.0]; \end{aligned}$$

alle quali corrisponde il seguente codice in RLI:

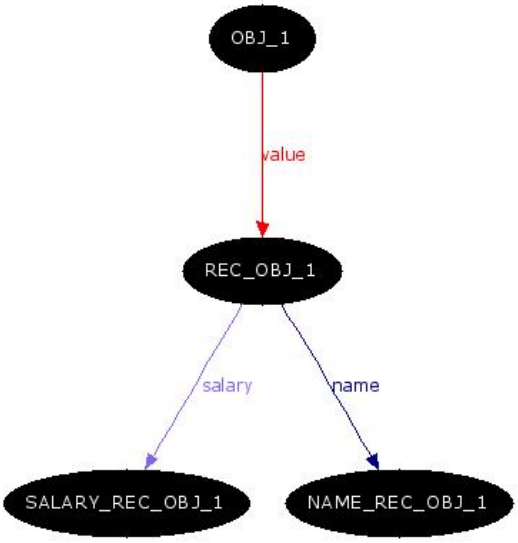

Figura 7.1: Rappresentazione grafica dell'oggetto OBJ_1

(INSTANCE OBJ_2 MATERIAL)

(INSTANCE *REC*_OBJ_2

(AND RECTYPE

(EXACTLY 1 NAME)

(EXACTLY 1 RISK)

(ALL LEVEL BOTTOM)

(ALL QUALIFICATION BOTTOM)

. . .

(ALL WORKS BOTTOM)

(ALL TONNAGE BOTTOM)))

(RELATED

OBJ_2 *REC*_OBJ_2 VALUE)

(CONSTRAINED

*NAME_REC*_OBJ_2 *SNAME_REC*_OBJ_2 STRING)

(CONSTRAINTS

(STRING= *SNAME_REC*_OBJ_2 |steel|))

(RELATED

*REC*_OBJ_2 *NAME_REC*_OBJ_2 NAME)

(CONSTRAINED

*NAME_REC*_OBJ_2 *IRISK_REC*_OBJ_2 INTEGER)

(CONSTRAINTS

(EQUAL *IRISK_REC*_OBJ_2 2))

(RELATED

*REC*_OBJ_2 *RISK_REC*_OBJ_2 RISK)

. . . . . .

e la rappresentazione mostrata in figura 7.2.

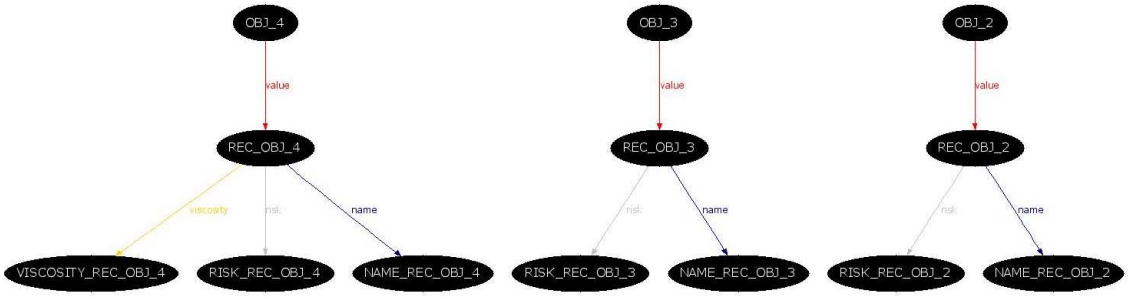

Figura 7.2: Rappresentazione grafica degli oggetti: OBJ_2, OBJ_3, OBJ_4

Per l'inserimento di un tipo insieme riferiamoci all'istanza OBJ_5 della classe *Repository*:

$$OBJ_5 \;\rightarrow\; [denomination : WestRepository, stock : OBJ_2,\ OBJ_3];$$

(INSTANCE OBJ_5 REPOSITORY)

(INSTANCE *REC*_OBJ_5

(AND RECTYPE

(EXACTLY 1 DENOMINATION)

(EXACTLY 1 STOCK)

(ALL NAME BOTTOM)

(ALL LEVEL BOTTOM)
(ALL QUALIFICATION BOTTOM)
...
(ALL WORKS BOTTOM)
(ALL TONNAGE BOTTOM)))
(RELATED
OBJ_5 *REC_*OBJ_5 VALUE)

(CONSTRAINED
*DENOMINATION_REC_*OBJ_5 *SDENOMINATION_REC_*OBJ_5 STRING)
(CONSTRAINTS
(STRING= *SDENOMINATION_REC_*OBJ_5 |West Repository|))
(RELATED
*REC_*OBJ_5 *DENOMINATION_REC_*OBJ_5 DENOMINATION)

Come già visto, per "chiudere" un tipo insieme, sia universale che esistenziale, è necessario e sufficiente fare un'asserzione di concetto su AllSetType, specificando il numero di elementi che formano l'insieme, ovvero il numero di riempitivi per il ruolo *member*; dato che, nel nostro caso, l'insieme dei riempitivi per l'attributo *item* è formato da due elementi, si scrive:

(INSTANCE *SET_ITEM_REC_*OBJ_5 (AND ALLSETTYPE (EXACTLY 3 MEMBER)));

e si completa la descrizione dell'insieme asserendo i riempitivi del ruolo *member* per *SET_ITEM_REC_*OBJ_5.

(RELATED
*SET_ITEM_REC_*OBJ_5 OBJ_2 MEMBER)
(RELATED
*SET_ITEM_REC_*OBJ_5 OBJ_3 MEMBER)

Infine, con:

(RELATED

*REC_*OBJ_5 *SET_ITEM_REC_*OBJ_5 STOCK)

si lega il record all'insieme tramite l'attributo item. La figura 7.3 riporta la rappresentazione grafica degli oggetti inseriti.

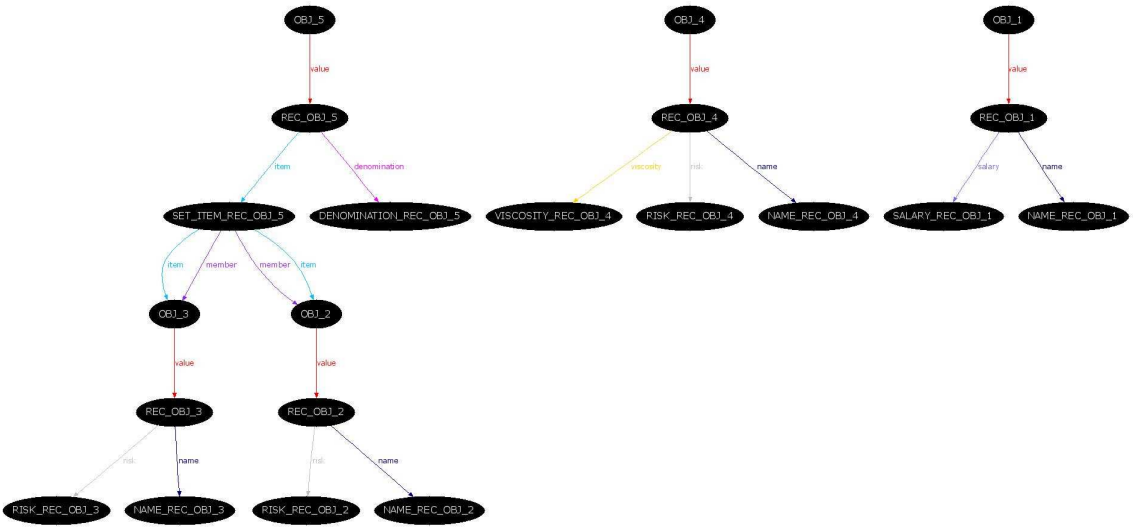

Figura 7.3: Rappresentazione grafica degli oggetti inseriti.

7.1.2 Considerazioni sulla rappresentazione estensionale

L'esempio precedente mette in evidenza che, per effettuare la rappresentazione, è neccessario:

- conoscere le classi primitive, sulle quali è possibile fare l'inserimento;
- conoscere gli attributi definiti nella parte estensionale della base di conoscenza.
- avere la conoscenza completa del record da inserire.

Le asserzioni nell'ABox devono utilizzare gli stessi concetti e ruoli atomici definiti nella terminologia, in modo tale che gli individui vengano collocati in un dominio strutturato del formalismo terminologico.

Solo sulla base di questo è possibile:

- chiudere le asserzioni sugli individui di RecType.
- chiudere le asserzioni sugli individui di AllSetType.
- selezionare l'operatore da utilizzare per i valori sui domini concreti.

Con riferimento alla struttura di base utilizza per la traduzione, si ottiene come risultato che:

- gli individui che popolano il concetto ClassType sono solo oid;
- gli individui che popolano il concetto RecType sono solo tipi record;
- gli individui che popolano il concetto AllSetType sono solo tipi insieme (universale o esistenziale);
- gli individui che popolano il concetto RangeType sono solo oggetti[2] che hanno un corrispettivo in un dominio concreto.

7.2 Ragionamento nell'ABox

7.2.1 Verifica della consistenza nell'ABox

Un oggetto viene costruito mediante un'insieme di asserzioni atomiche, quindi, qualora una di esse dovesse generare un'inconsistenza l'intero oggetto è da considerarsi inconsistente.

In questa sezione vengono riassunte ed analizzate le possibili cause di inconsistenza in relazione all'inserimento di un'istanza.

- Incosistenza rilevata sull'inserimento dell'oid

  La causa di incosistenza è da ricercarsi nel fatto che l'oid appartiene già ad un

[2] in questo caso il termine oggetto fa riferimento alla terminologia introdotta con RACER nel cap 4.

altro concetto che è disgiunto da quello rispetto al quale vogliamo fare l'asserzione. Questo caso si può verificare solamente se l'oid è stato introdotto come riempitivo di un attributo di un'altra istanza, senza esser quindi stato definito; infatti in questo caso il sistema gli attribuisce una classe di appartenza concorde con la descrizione del tipo dell'istanza di cui fa parte.

- Incosistenza rilevata sull'inserimento del record
  In questo caso, l'inconsistenza è da attribuirsi al fatto che nel record non sono specificati uno o più attributi essenziali per le istanze della classe alla quale appartiene l'oggetto che si sta inserendo.

- Inconsistenza rilevata sull'inserimento del riempitivo di un attributo
  Si verifica quando il tipo di riempitivo non è coerente rispetto a quello specificato nella descrizione dell'attributo della classe di appartenenza dell'oggetto che si vuole inserire.

- Inconsistenza rilevata sull'inserimento di un tipo insieme
  Il numero di riempitivi per il ruolo *member* è superiore al numero specificato nella dichiarazione dell'insieme, oppure la causa dell'inconsistenza è da ricondurre al caso precedente.

7.2.2 Altre forme di ragionamento

Da quanto detto finora è chiaro che la rappresentazione estensionale delle istanze appartenti alle classi di uno schema, tradotto in modo tale che questi sia inferenziabile da RACER, oltre ad essere complicata è anche molto limitata. Infatti l'unica forma di ragionamento possibile è l'estrazione di individui appartenenti ad un concetto fissato; questo è un ragionamento meno generale dell'interrogazione, perchè consente di ottenere solamente insiemi d'individui invece di insiemi di n-ple. Inoltre risulta molto difficile, se non impossibile, mettere in relazione diversi oggetti per comporre i risultati. Infatti utilizzando l'estrazione non è possibile formulare in modo semplice delle interrogazioni del tipo:

"quali sono tutti i *Repository* e cosa contengono", ma è necessario comporre singolarmente i singoli risultati. Tuttavia, implementando gli oggetti secondo quanto specificato precedentemente, è possibile implementare semplici algoritmi che sfruttano il sistema RACER per svolgere i seguenti compiti di ragionamento:

- instance check: determinare l'appartenenza o meno di oggetto ad una determinata classe;
- retrieval problem: determinare tutti gli oggetti appartenenti ad una data classe.

restituendo dei risultati attendibili. É inoltre possibile sfruttare le capacità di RACER per dare una rappresentazione grafica della conoscenza estensionale. In definitiva RACER è utilizzabile come validatore e classificatore per gli oggetti.

7.3 Linguaggio di riferimento per l'inserimento delle istanze

Viene ora introdotto il linguaggio d'interfaccia, abbastanza intuitivo, che, riprendendo il formalismo utilizzato in questo capitolo, verrà utilizzato per permettere l'inserimento, da parte dell'utente, degli oggetti appartenenti alle classi base.
La sua BNF è:

<oid>**(**<class_name>**)** = **[**<attrib_name>**:** <value> [, <attrib_name>: <value>] **];**

<value> → <single_value> |
<set_value>

<set_value> → **{**<single_value> [, <single_value>]**}** |

<single_value> → **"**(a-b)|(A-Z)**"** | (string)
(0-9) | (integer)
(0-9).(0-9) | (real)
<oid> | (oid)

Capitolo 8

OOSWiR

8.1 Presentazione del sistema

OOSWiR (**O**bject **O**riented **S**chema **wi**th **R**acer) è il prototipo di uno strumento di supporto, realizzato in Java, per la progettazione di schemi minimali e consistenti per Basi di Dati orientate agli oggetti.

Basato sul modello presentato ed analizzato nel capitolo 6, utilizza RACER per applicare diverse forme di ragionamento, ed adotta come interfaccia il linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** presentato in 6.5.

La figura 8.1 mostra la sua architettura, nella quale si possono individuare i componenti principali:

- ODL Interface
  É l'interfaccia con l'esterno che permette all'utente di inserire schemi espressi in **ODL**$_{I^3}$-$\mathcal{OLCD}$.

- Translator Module
  Questo componente traduce lo schema dal linguaggio **ODL**$_{I^3}$-$\mathcal{OLCD}$/**RACER** al linguaggio RLI.

- RACER Interface

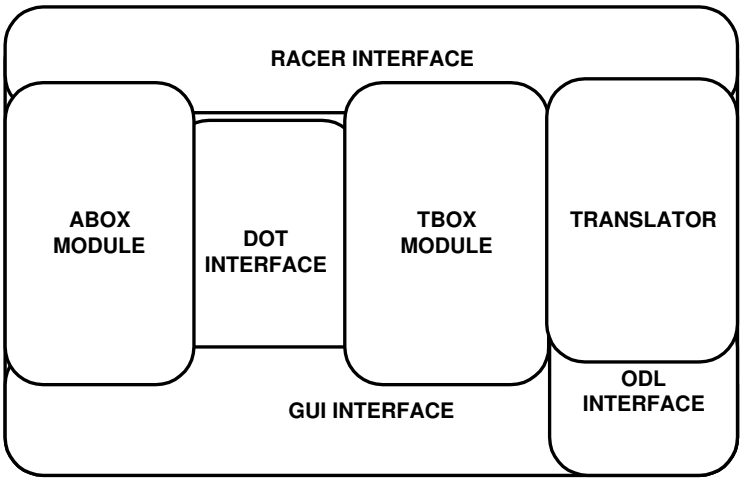

Figura 8.1: Architttetura di OOSWiR

É l'interfaccia che permette lo scambio di informazioni tra l'applicazione e il ragionatore RACER.

- TBox Module

  É il modulo che ha il compito di interagire con RACER per ricavare le informazioni riguardanti la parte intesionale dello schema. Interrogando RACER, tramite delle query mirate, ricava tutte le informazioni inerenti la validazione, le relazioni tra le classi e la loro descrizione, e ne gestisce la rappresentazione, implementando al suo interno opportune strutture dati.

- ABox Module

  Questo componente gestisce la parte estensionale. Secondo le modalità descritte in sezione 7.1.1 consente di inserire delle istanze da parte dell'utente, ed interagisce con RACER per verificarne la consistenza e ricarvarne la classificazione. Si occupa anche di gestire la loro rappresentazione (grafica).

- Dot Interface

  Quest'interfaccia è utilizzata per interagire con Graphviz, un'applicazione di libero utilizzo, sviluppata da Emden Gansner, Eleftherios Koutsofios, Stephen North, (ATT) che permette la creazione file grafici che rappresentano tassonomie.

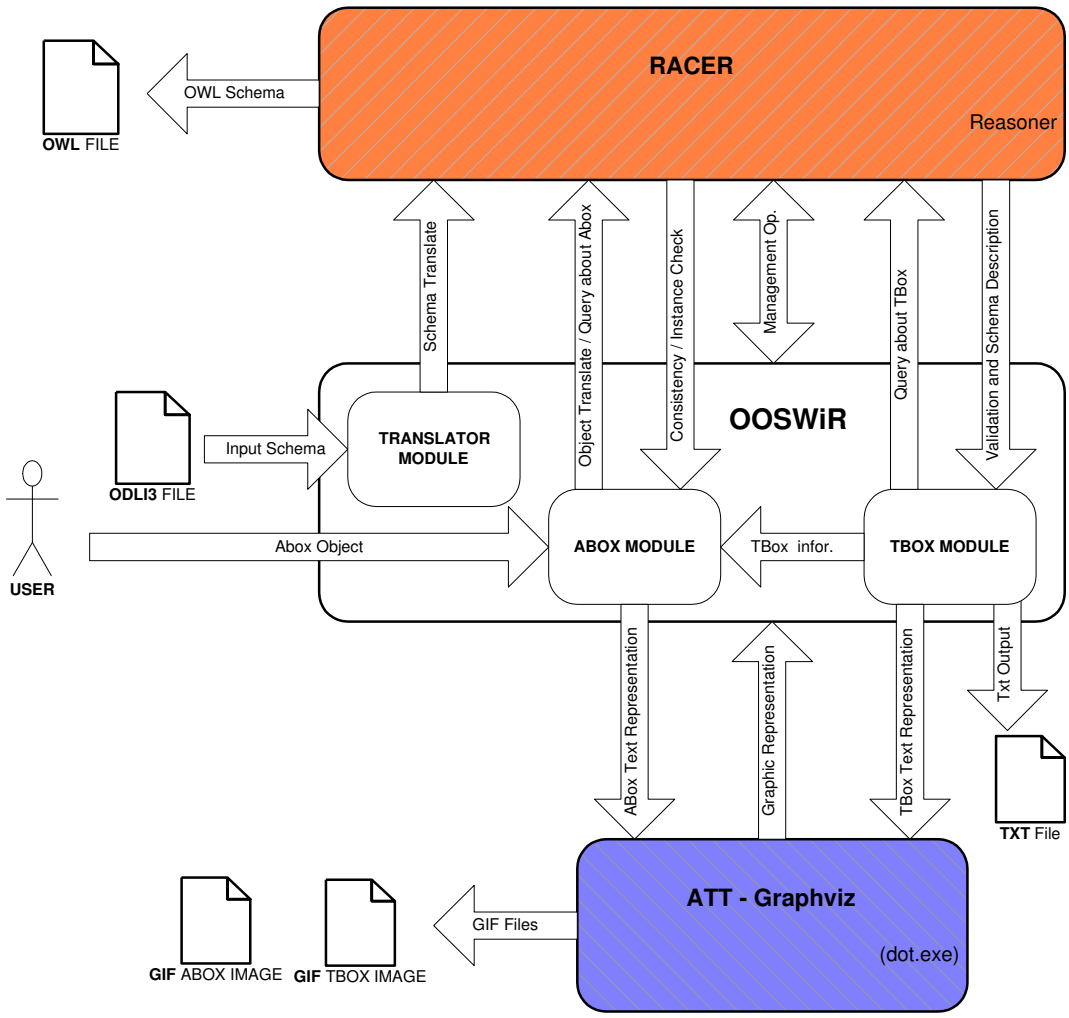

Figura 8.2: Schema generale dell'interazione tra i componenti

- GUI Interface

  É l'interfaccia grafica che il sistema implementa per ricevere e presentare le informazioni all'utente.

8.2 Requisiti del Software

8.2.1 Requisiti Funzionali

Requisito #1

- Introduzione. Acquisizione e traduzione dello schema.
- Input. File in formato di testo.
- Processing. Analisi del testo, traduzione mediante applicazione di regole.
- Output. Messaggio di notifica risultato dell'operazione e file temporaneo in formato testo contenente schema in linguaggio RACER.

Requisito #2

- Introduzione. Validazione dello schema.
- Input. File temporaneo in formato testo contenente schema in linguaggio RACER.
- Processing. Invio dello schema al server RACER, interrogazione al server RACER.
- Output. Messaggio risultato interrogazione, lista delle classi inconsistenti.

Requisito #3

- Introduzione. Informazioni relazioni ISA tra le classi.
- Input. Lista delle classi consistenti.
- Processing. Interrogazioni (sussunzione tra concetti) al server RACER.
- Output. Tabella relazioni ISA tra le classi.

Requisito #4

- Introduzione. Informazioni relazioni EQU tra le classi.
- Input. Lista delle classi consistenti.
- Processing. Interrogazioni (equivalenza tra concetti) al server RACER.
- Output. Tabella relazioni EQU tra le classi.

Requisito #5

- Introduzione. Informazioni relazioni DISJOINT tra le classi.
- Input. Lista delle classi consistenti.
- Processing. Interrogazioni (disgiunzione tra concetti) al server RACER.
- Output. Tabella relazioni DISJOINT tra le classi.

Requisito #6

- Introduzione. Informazioni descrizione delle classi.
- Input. Lista delle classi consistenti, lista degli attributi definiti dallo schema, lista di tutti i tipi definiti dallo schema.
- Processing. Interrogazioni (basate sulla sussunzione) al server RACER.
- Output. Descrizione di ciascuna classe.

Requisito #7

- Introduzione. Grafico delle classi.
- Input. Lista delle classi consistenti. Relazioni ISA ed relazioni EQU.

- Processing. Creazione dello stream di input per l'applicazione Graphviz. Esecuzione di Graphviz.
- Output. Visualizzazione del formato grafico creato da Graphviz.

Requisito #8

- Introduzione. Inserimento oggetti relativi alle classi
- Input. Lista delle classi base consistenti. Descrizione oggetto da inserire.
- Processing. Traduzione della descrizione dell'oggetto. Invio delle informazioni al server RACER. Verifica della consistenza dell'ABox.
- Output. Messaggio di notifica risultato dell'operazione.

Requisito #10

- Introduzione. Classificazione di oggetti.
- Input. Lista degli oggetti inseriti, lista delle classi consistenti.
- Processing. Interrogazione (instance checking) al server RACER.
- Output. Tabella relazioni di appartenenza degli oggetti alle classi.

Requisito #11

- Introduzione. Grafico della descrizione degli oggetti.
- Input. Lista degli oggetti inseriti, lista degli attributi definiti dallo schema.
- Processing. Interrogazione (relazione tra individui) al server RACER. Creazione dello stream di input per l'applicazione Graphviz. Esecuzione di Graphviz.
- Output. Visualizzazione del formato grafico creato da Graphviz.

8.2.2 Requisiti delle interfacce esterne

8.2.3 User Interface

Le interfacce per gli utenti devono essere il più possibile “user friendly”. Ad ogni form compilato ed inviato deve corrispondere una risposta dal server e, nei limiti del possibile, ad ogni operazione deve corrispondere un solo form.

8.2.4 Hardware

Non è richiesto nessun requisito hardware particolare.

8.2.5 Software

Sono richieste le applicazioni software: RACER, ver(1.7.23) e Graphviz ver(1.6).

8.3 Organizzazione del software

La scelta, riguardo il linguaggio di programmazione da utilizzare per l’implentazione di OOSWiR, è ricaduta su JAVA, in quanto oltre ad essere un linguaggio OO, e quindi offrire un alto livello di modularità, permette un facile interfacciamento con RACER ed offre una serie di caratteristiche, che rendono pratica la gestione delle stringhe, utili nella realizzazione del traduttore.

Il progetto del software ha previsto la realizzazione di una struttura organizzata in packages: un package permette di raggruppare classi Java per scopo o per ereditarietà e consente una loro maggiore protezione. Nelle fasi di sviluppo di OOSWiR sono stati individuati nove package distinti, mostrati in figura 8.3.

- Package: ”translator”. Il package chiamato translator implementa il motore del traduttore.La struttura di questo package è rappresentata in 8.4. Il package translator è attualmente formato da quattro classi: TranslatorEngine, ODLParser, LineTranslator, MakeLabel. La classe principale è TranslatorEngine, che fornisce i metodi per

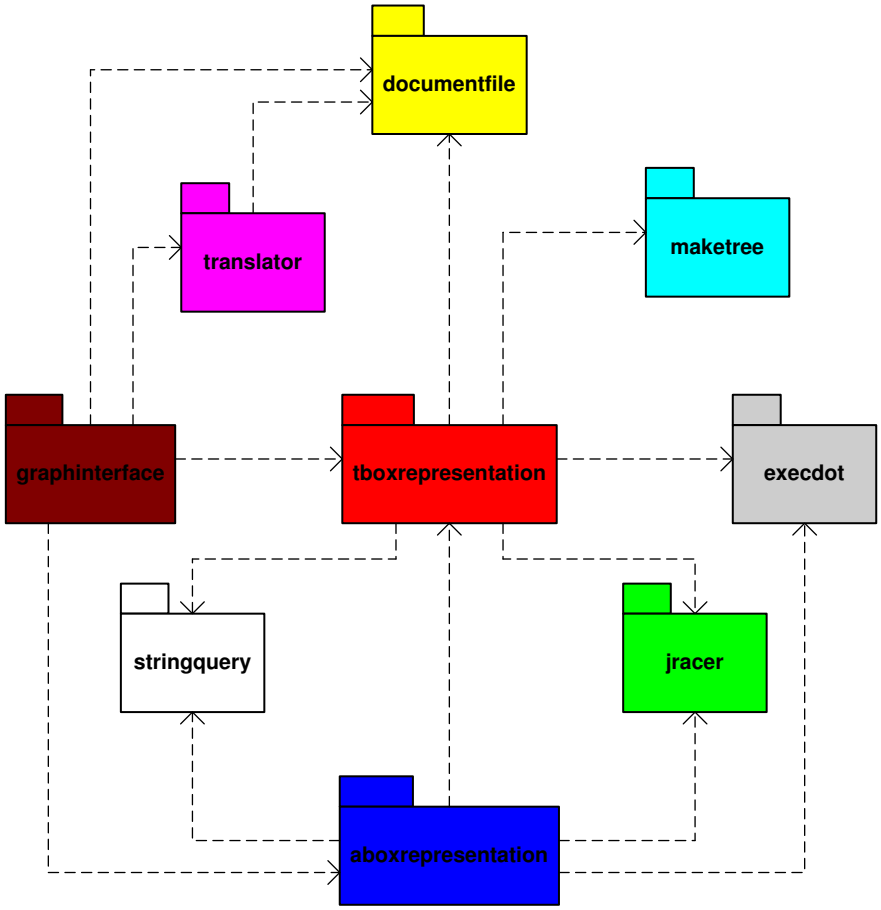

Figura 8.3: OOSWiR Packages Diagram

ottenere la traduzione dello schema. ODLParser è la classe che analizza le righe di codice **ODL**$_{\mathcal{I}^3}$-$\mathcal{OLCD}$/**RACER**, e stabilisce se la loro sintassi è corretta, mentre la classe LineTranslator implementa l'automa e gli algoritmi utilizzati per la traduzione. MakeLabel è una classe che genera delle label; tali label sono utilizzate per identificare i tipi range ed union, definiti nello schema originale.

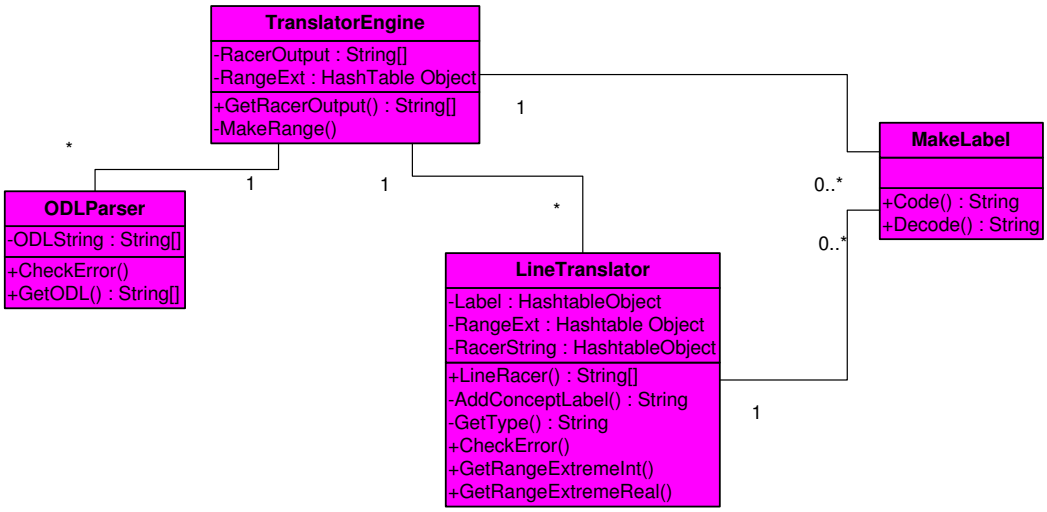

Figura 8.4: Class Diagram package translator

- Package: "tboxrepresentation". Questo package contiene le classi che realizzano le funzionalità principali riguardanti la rappresentazione intensionale. Delle cinque classi che lo compongono la principale è Taxonomy che permette di verificare la validazione di uno schema e fornisce l'accesso alle informazioni riguardanti le classi. La classe ClassDescription mette a disposizione i metodi per il recupero delle informazioni circa la descrizione delle classi, ottenute sfruttando la classe CheckAttribute che implementa un algoritmo di interrogazione per RACER. La classe CheckConcept dispone dei metodi per ricavare la tassonomia dei concetti e permette di recuperare i nomi dei concetti, suddividendoli per tipo di appartenenza(ClassType, RecType, AllSetType, etc). Infine, ClassRelationship permette di ricavare le relazioni esistenti tra le classi, implementando in tal senso un algoritmo di interrogazione per RACER.

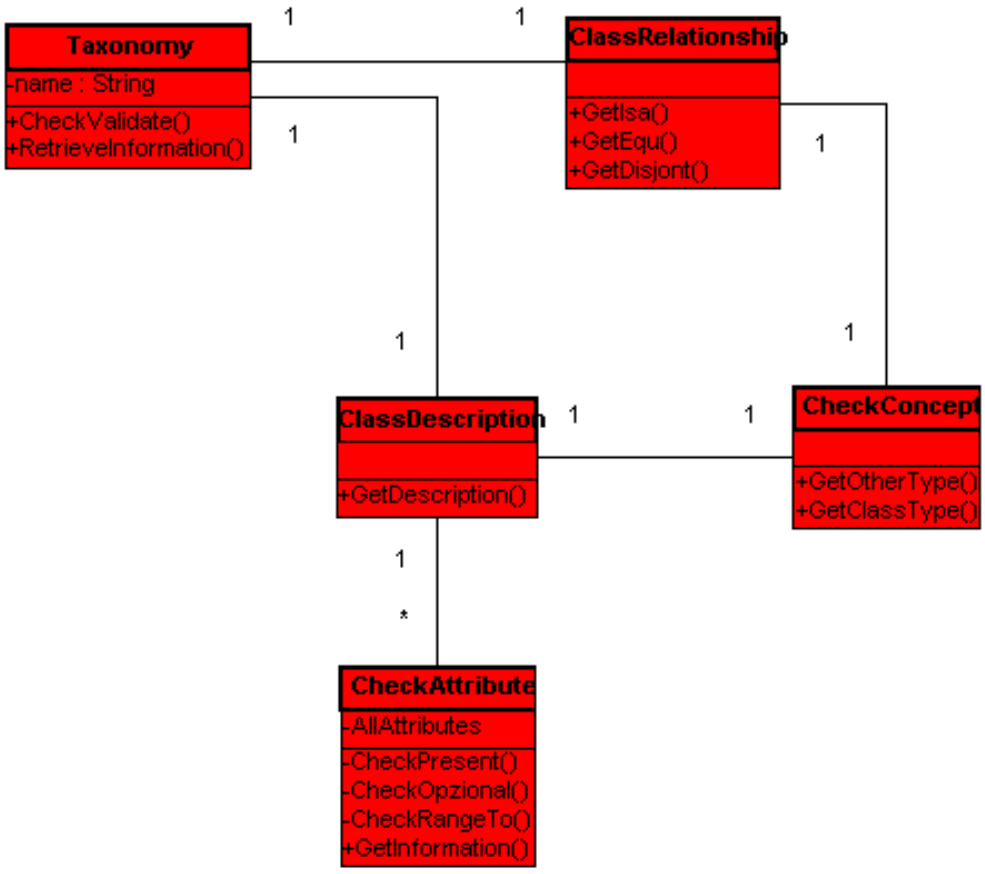

Figura 8.5: Class Diagram package tboxrepresentation

- Package: “aboxrepresentation”. Questa libreria mette a disposizione gli strumenti

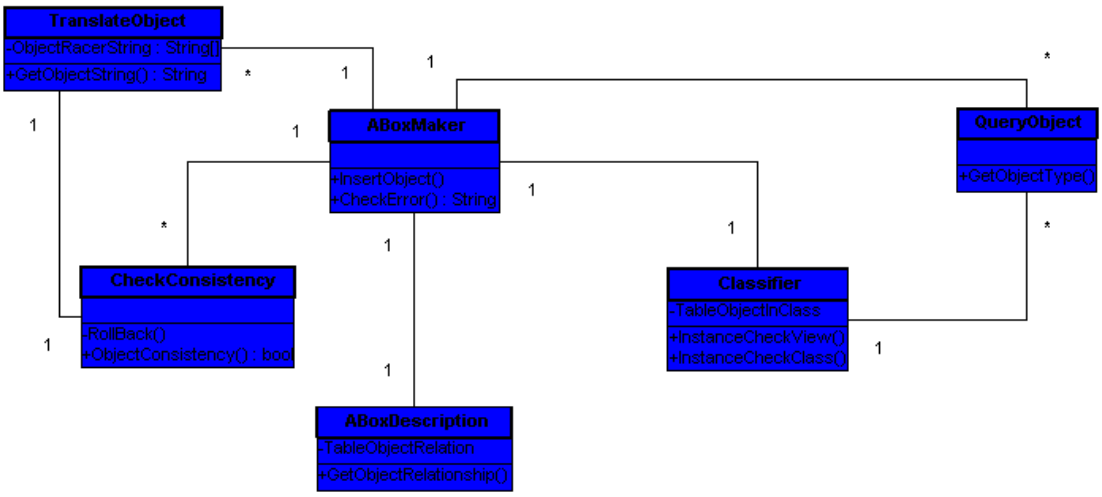

Figura 8.6: Class Diagram package aboxrepresentation

per la gestione della rappresentazione estensionale. ABoxMaker permette l’inserimento di oggetti, nella parte relativa all’ABox, in RACER, occupandosi di verificarne la consistenza, grazie alle classi TranslateObject e CheckConsistency. In particolare la classe CheckConsistency, oltre a verificare la consistenza di un oggetto, implementa un meccanismo di rollback per eliminare gli oggetti inconsistenti dall’ABox di RACER. La classe Classifier contiene le informazioni di classificazione per gli oggetti inseriti, mentre QueryObject permette di ricavare il tipo per l’oggetto inserito. ABoxDescription astrae le informazioni riguardanti la rappresentazione grafica degli oggetti, utilizzando un’apposita struttura per memorizzare i legami tra gli elementi che formano gli oggetti stessi.

- Package “graphinterface”. Contiene le classi che implementano l’interfaccia grafica mediante la quale l’utente è in grado d’interagire con il sistema. La classe principale è MainFrame, che crea la finestra dell’applicazione. Le altre classi mettono a disposizione i componenti utilizzati per la creazione dei pannelli di dialogo.

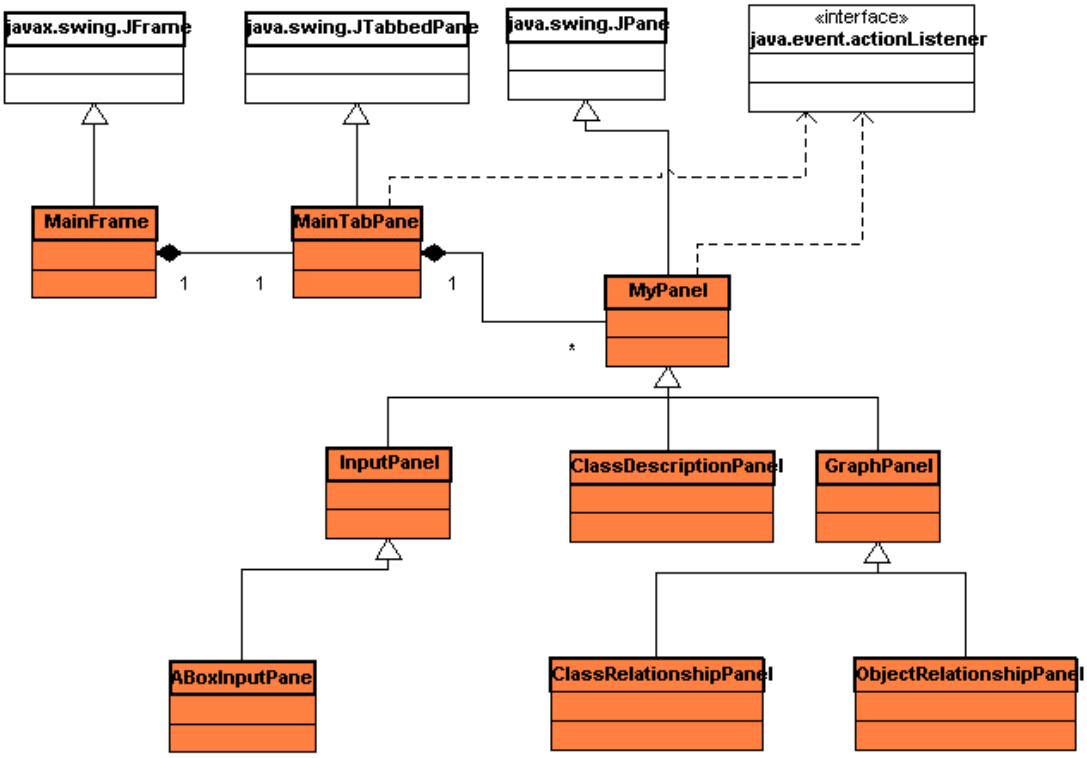

Figura 8.7: Class Diagram package graphinterface

- Package: “maketree”. Questo package fornisce le classi che implementano la struttura dati interna, sulla quale lavora OOSWiR, che viene creata sulla base delle informazioni riguardanti la parte intensionale, ricevute da RACER. La classe TreeOfClassRelations permette di rappresentare l’albero contenente la tassonomia delle classi, mentre TreeOfClassDescription è utilizzato per rappresentare l’albero della descrizione delle classi.

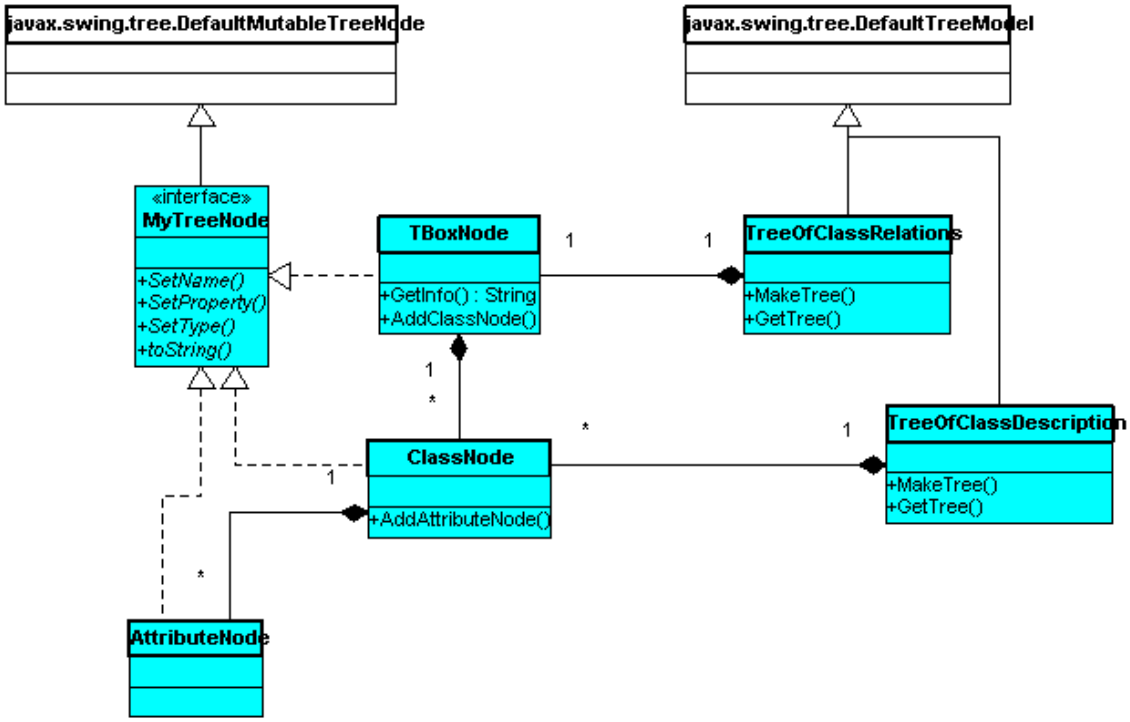

Figura 8.8: Class Diagram package maketree

- Package: “stringquery”.

  Raggruppa le classi che permettono la manipolazione delle stringhe di comunicazione, interrogazione e management che vengono utilizzate per interagire con RACER.

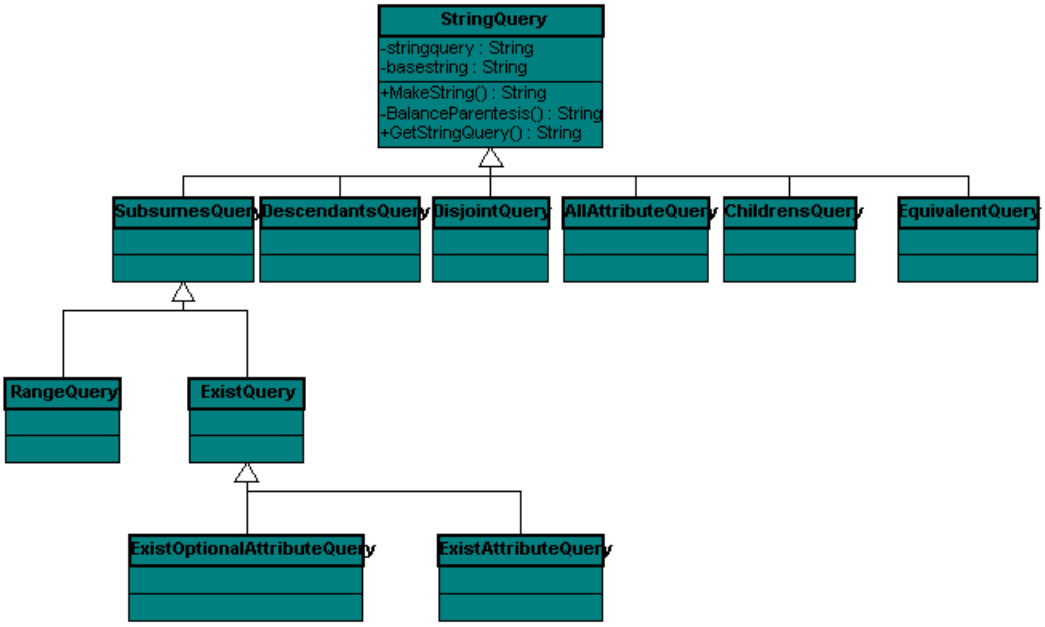

Figura 8.9: Class Diagram package stringquery

- Package: “jracer”.

  All’interno di questo package si trovano le classi e i metodi utilizzati per instaurare il canale, TCP-IP, di comunicazione tra OOSWiR e RACER.
  La classe RacerSocket implementa i metodi per inizializzare ed aprire una connessione persistente con RACER, chiudere la connessione, verificare lo stato della connessione. La classe Command gestisce l’invio dei comandi di management, mentre la classe Query gestisce l’invio delle query al server RACER, implementando un metodo sendQuery di tipo sincrono, che ritorna la risposta del server. ResultParser contiene i metodi che effettuano il parser delle risposte ricevute.

- Package: “execdot”.

  Realizza le funzionalità d’interfaccia con Graphviz, permettendo di utilizzare quest’ultima applicazione per la creazione dei formati grafici utilizzati da OOSWiR e disponibili anche come output in forma di file.

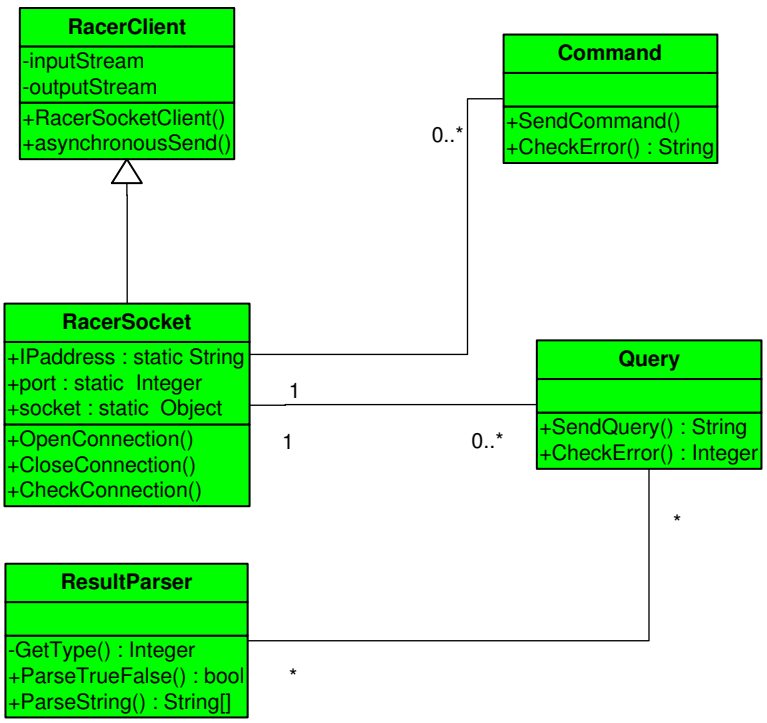

Figura 8.10: Class Diagram package jracer

- Package: “documentfile”.

  Raggruppa le classi utilizzate per l’accesso in lettura e/o scrittura delle informazioni sul disco rigido. InputGIFFile permette l’accesso ai file grafici in formato GIF, mentre InputODLFile è utilizzato per effettuare la lettura, per righe, dei file in input contenenti il codice **ODL**$_{I^3}$-$\mathcal{OLCD}$/RLI. Le classi di Output implementano i metodi per scrivere i diversi formati previsti in output.

8.4 Realizzazione del Translator Module

Il diagramma 8.13 mostra l’interazione tra le classi che concorrono a formare la parte relativa al modulo di traduzione implementato in OOSWiR. Tale modulo soddisfa il requisito #1.

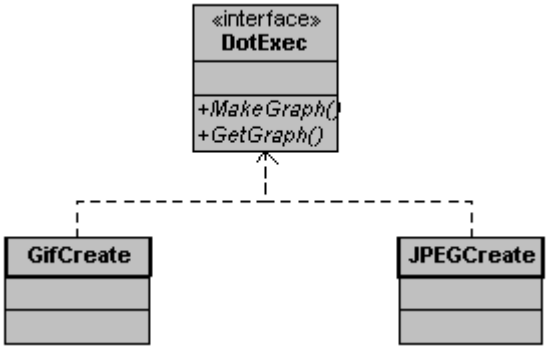

Figura 8.11: Class Diagram package execdot

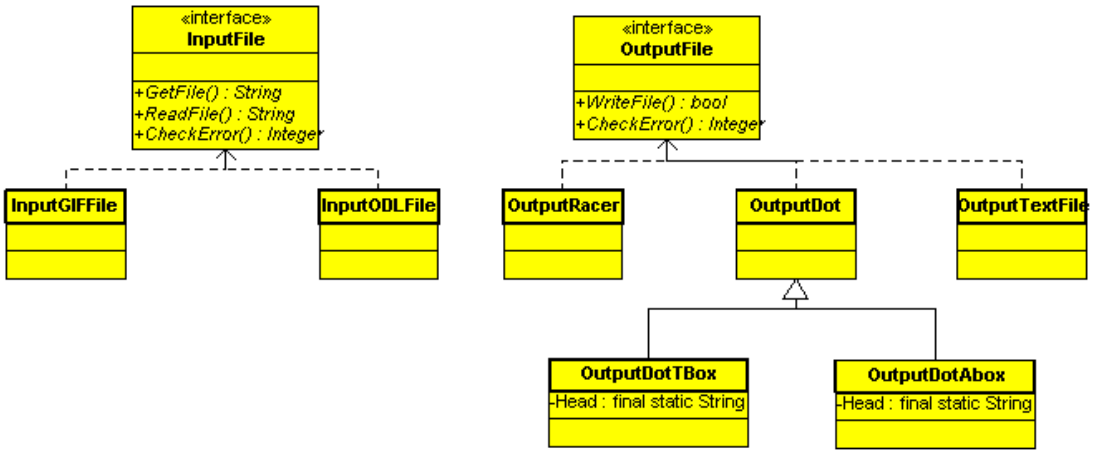

Figura 8.12: Class Diagram package documentfile

8.4.1 Funzionamento

La traduzione avviene applicando le regole viste in 6.5.1 e altre ottimizzazioni. Per spiegarne il funzionamento consideriamo per esempio il codice **ODL**$_{I^3}$-$\mathcal{OLCD}$:

```
interface ClasseA {
    attribute range{1, 34} attrib_range;
    attribute union<TipoA, TipoB> attrib_union;
};
```

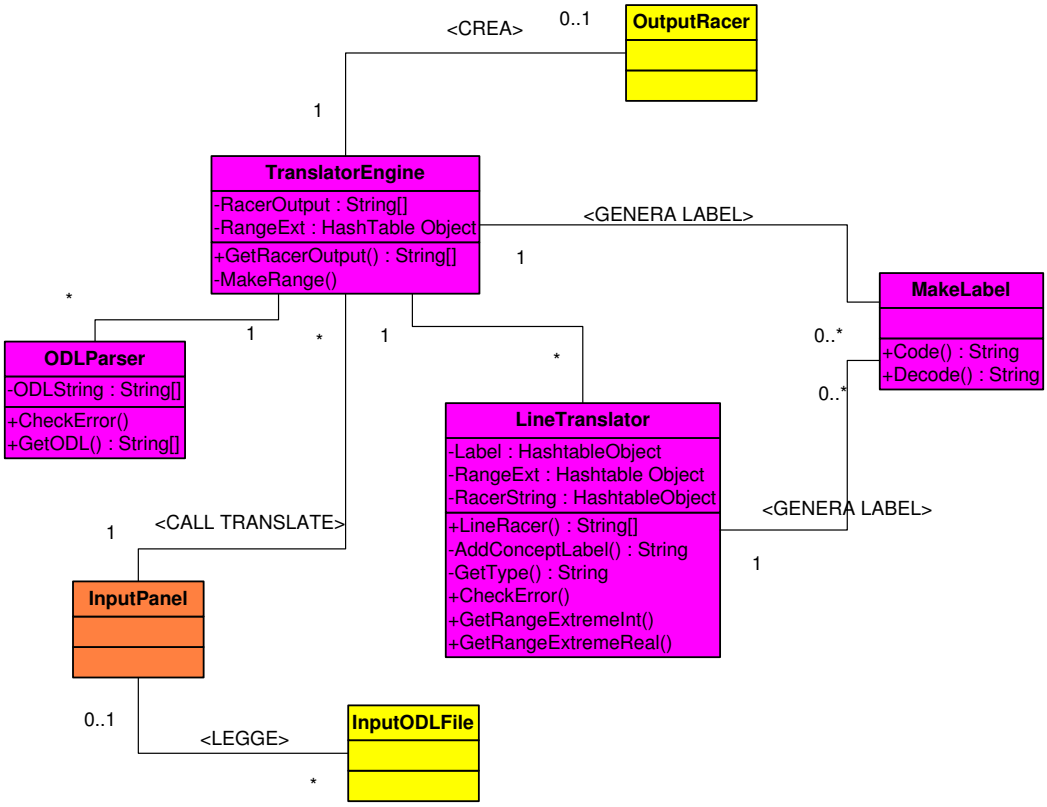

Figura 8.13: Class Diagram: realizzazione del modulo traduttore

```
interface ClasseB {
attribute range7, 59 attrib_range;
attribute set<TipoB> attrib_set;
};

interface ClasseC: ClasseA, ClasseB {
};
```

ODLParser effettua l'analisi sintattica del testo, converte i caratteri alfabetici in maiuscolo e crea un array di stringhe, formato di tanti elementi quante sono le classi da tradurre. La traduzione di ciascuna classe viene fatta, in modo indipendente rispetto alle altre, da LineTranslator. LineTranslator implementa un automa che ogni qualvolta incontra un tipo range sostituisce alla sua descrizione una label, mentre quando incontra un tipo union, allsettype oppure somesetype, oltre a sostituire alla sua descrizione una label, crea un concetto equivalente identificato dalla label stessa. Le label sono codificate secondo quanto riportato in tabella 8.1.

Alla traduzione della *ClasseA*, effettuata da LineTranslator, corrisponde il vettore di stringhe:

```
Stringa[0]=
    (IMPLIES CLASSEA (AND CLASSTYPE (AND VALUE (AND RECTYPE
    (EXACTLY 1 ATTRIB_RANGE) (ALL ATTRIB_RANGE R_I_1_34)
    (EXACTLY 1 ATTRIB_UNION) (ALL ATTRIB_UNION U#TIPOA#TIPOB) ))))
Stringa[1]=
(EQUIVALENT U_TIPOA_TIPOB (AND UNIONTYPE
    (OR TIPOA TIPOB)))
```

Il metodo GetRangeExtremeInt() di LineTranslator restituisce gli estremi degli intervalli di interi definiti dalla descrizione della classe (GetRangeExtremeReal() fa lo stesso ma per gli intervalli di reali). In questo caso abbiamo: {1, 2}.

Tipo	Modello Label Base	Esempio
Range(Interger)	R_I_<nr>_<nr>	range{1, 34}→ R_I_1_34
Range(Real)	R_R_<nr>d<nr>_<nr>d<nr>	range{3.5, 21}→ R_R_3d5_21d0
Set	A_tipo	set<TipoB>→ A_TipoB
Exists	S_tipo	exists<TipoD>→ S_TipoD
Union	U#tipo$_1$_. . .#tipo$_n$	union<TipoB, range{1, 2}>→ U#TipoB#R_I_1_3

Tabella 8.1: Codifica delle label

Allo stesso modo la traduzione della *ClasseB*:

```
String[0]=
    (IMPLIES CLASSEB (AND CLASSTYPE (AND VALUE (AND RECTYPE
    (EXACTLY 1 ATTRIB_RANGE) (ALL ATTRIB_RANGE R_I_7_59)
    (EXACTLY 1 ATTRIB_SET) (ALL ATTRIB_SET A_TIPOB) ))))
    (EXACTLY 1 ATTRIB_UNION) (ALL ATTRIB_UNION U#TIPOA#TIPOB) ))))
String[1]=
(EQUIVALENT A_TIPOB (AND ALLSETTYPE
    (ALL MEMBER TIPOB)))
String[2]=
(EQUIVALENT U_TIPOA_TIPOB (AND UNIONTYPE
    (OR TIPOA TIPOB)))
GetRangeExtremeInt={7,59}
```

e della classeC:

```
String[0]=
```

(IMPLIES CLASSEC (AND CLASSTYPE CLASSEA CLASSEB))
GetRangeExtremeInt{}

TranslatorEngine coordina le operazioni: si preoccupa di mettere insieme i risultati delle singole traduzioni, facendo in modo che non compaiano più definizioni per lo stesso ruolo o concetto, e aggiunge la parte comune; inoltre sfrutta i valori ritornati da GetRangeExtremeInt (GetRangeExtremeReal()) per definire tutti i possibili intervalli ottenibili dalla combinazione degli estremi restituiti. Considerando l'esempio, i concetti relativi agli intervalli, sulla base degli estremi 1, 34, 7, 59, sono:

```
(EQUIVALENT R_I_1_1 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 1) (MAX INTEGER 1)))
(EQUIVALENT R_I_1_7 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 1) (MAX INTEGER 7)))
(EQUIVALENT R_I_1_34 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 1) (MAX INTEGER 34)))
(EQUIVALENT R_I_1_59 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 1) (MAX INTEGER 59)))

(EQUIVALENT R_I_7_7 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 7) (MAX INTEGER 7)))
(EQUIVALENT R_I_7_34 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 7) (MAX INTEGER 34)))
(EQUIVALENT R_I_7_59 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 7) (MAX INTEGER 59)))

(EQUIVALENT R_I_34_34 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 34) (MAX INTEGER 34)))
(EQUIVALENT R_I_34_34 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 34) (MAX INTEGER 59)))
```

```
(EQUIVALENT R_I_59_59 (AND RANGETYPE
    (A INTEGER) (MIN INTEGER 59) (MAX INTEGER 59)))
```

L'introduzione di questi nuovi concetti permetterà di risalire in modo più agevole alla descrizione delle classi, che avviene sfruttando delle interrogazioni basate sulla sussunzione. In particolare sarà possibile risalire alla descrizione di classi che ereditano degli attributi basati sui range, come la *ClasseC* dell'esempio, dove ci si aspetta che *attrib_range* abbia come riempitivo il range dei numeri interi compresi tra 7 e 34, non esplicitamente definito, ma implicitamente aggiunto dal traduttore. Senza la definizione di questi intervalli l'unica alternativa, per ricavare queste informazioni, sarebbe quella di "provare tutti i numeri", ovvero quella di formulare una query per ciascun numero compreso tra 1 e 59. Tale metodo oltre ad essere oneroso dal punto di vista del numero di interrogazioni, è sicuramente improponibile nel caso di range sui reali.

8.5 Realizzazione del TBox Module

Il Tbox Module svolge le funzionalità richieste dai requisiti: #2...#7. La sua realizzazione utilizza le classi mostrate in figura 8.14.

8.5.1 Funzionamento

Il funzionamento del modulo TBox è basato sull'interazione con il server RACER, che riceve come input iniziale la TBox preparata dal modulo di traduzione. Successivamente viene fatta un'interrogazione per sapere se la TBox è consistente e, in caso di risposta negativa, vengono richiesti i nomi delle classi inconsistenti. Si procede effettuando le interrogazioni per il recupero di tutte le informazioni di carattere generale:

1. recupero dei concetti appartenenti a ClassType;

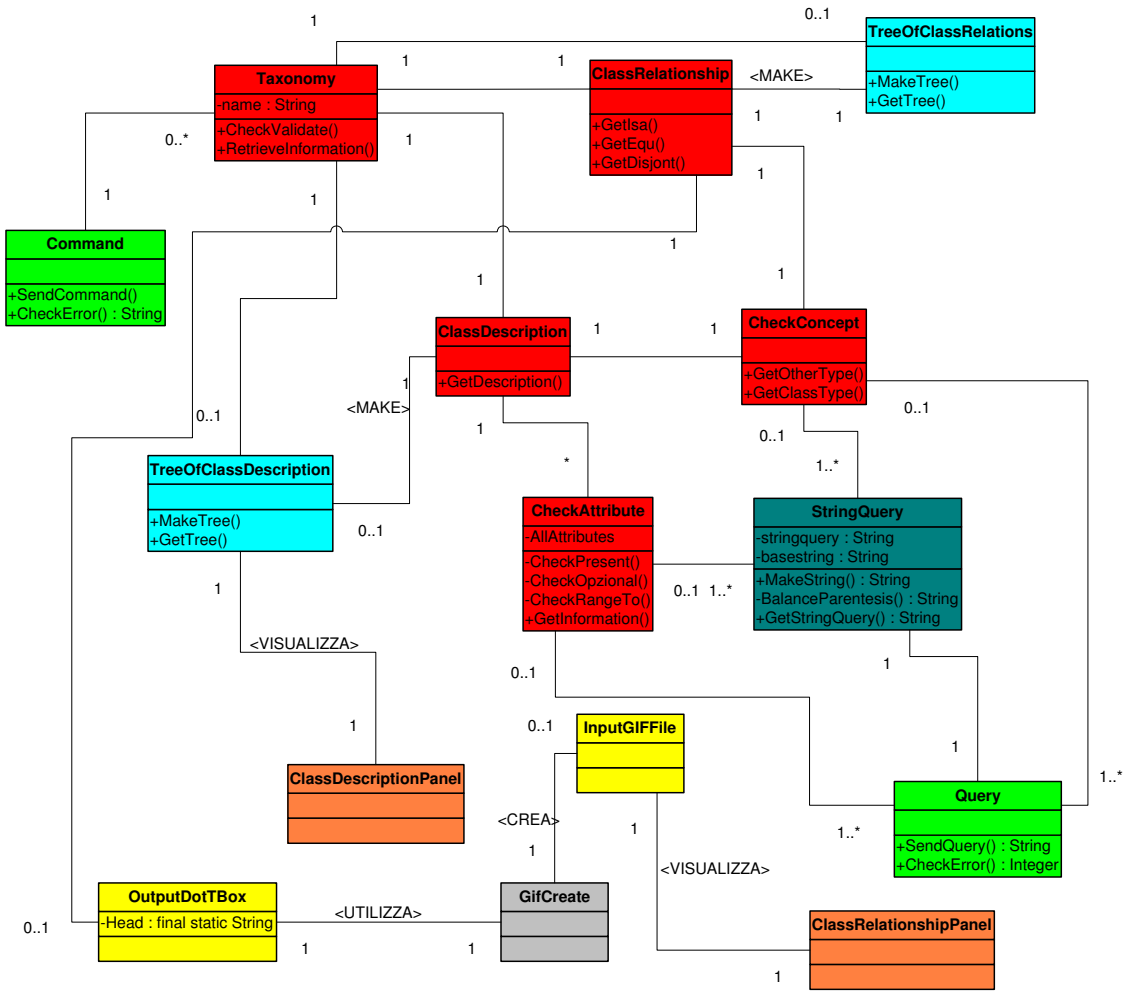

Figura 8.14: Class Diagram: realizzazione del modulo TBox

2. recupero dei concetti appartenenti a RecType;
3. recupero dei concetti appartenenti a UnionType;
4. recupero dei concetti appartenenti a RangeType;
5. recupero dei concetti appartenenti a AllSetType;
6. recupero dei concetti appartenenti a SomeSetType;
7. recupero degli attributi definiti dalla TBox;

ClassRelationship si preoccupa di ricavare le relazioni tra le classi; per ciascuna coppia di concetti appartenenti a ClassType viene interrogato RACER per sapere se:

1. i due concetti sono equivalenti;
2. uno dei due sussume l'altro;
3. sono tra loro disgiunti;

seguendo l'algoritmo riportato in figura 8.15.

ClassDescription ricava le informazioni riguardo la descrizione di ciascuna classe. Utilizzando la lista degli attributi definiti nella TBox effettua, per ciascuno di questi, delle interrogazioni al server RACER basate sulla sussunzione, alle quali quest'ultimo risponde con un True oppure un Nil. L'algoritmo è stato implementato sulla base di quanto detto nel capitolo 6 ed è mostrato in figura 6.3. Una volta che è stata stabilita la presenza di un attributo nella descrizione di una classe, viene prima recuperato il tipo di appartenenza per il riempitivo ammesso, e poi il concetto più specifico che sussume il riempitivo. Anche questo procedimento sfrutta la sussunzione ed è mostrato in 8.16.

Tutti i dati ricavati vengono inseriti in un'apposita struttura ad albero.

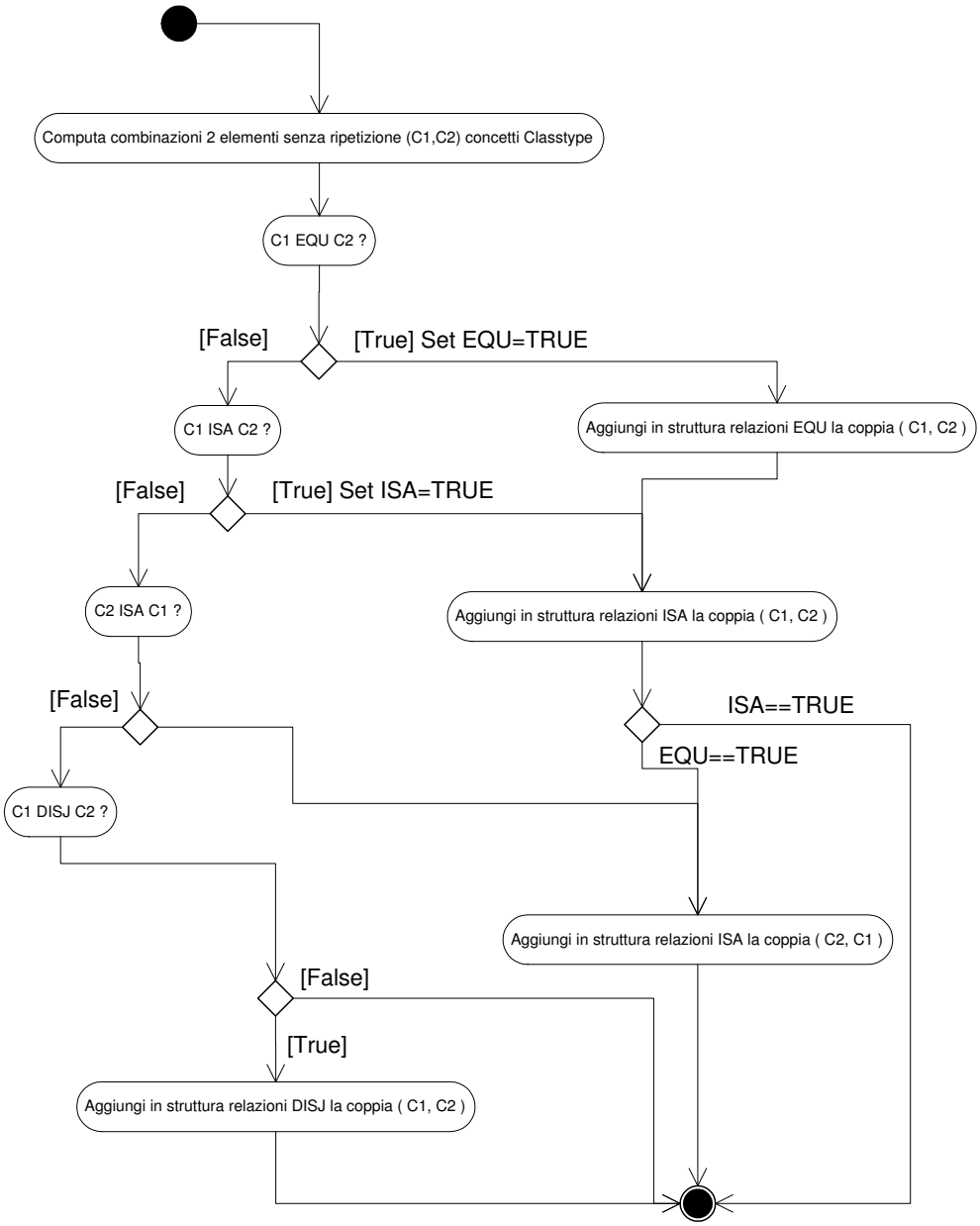

Figura 8.15: Activity Diagram: recupero relazioni tra le classi

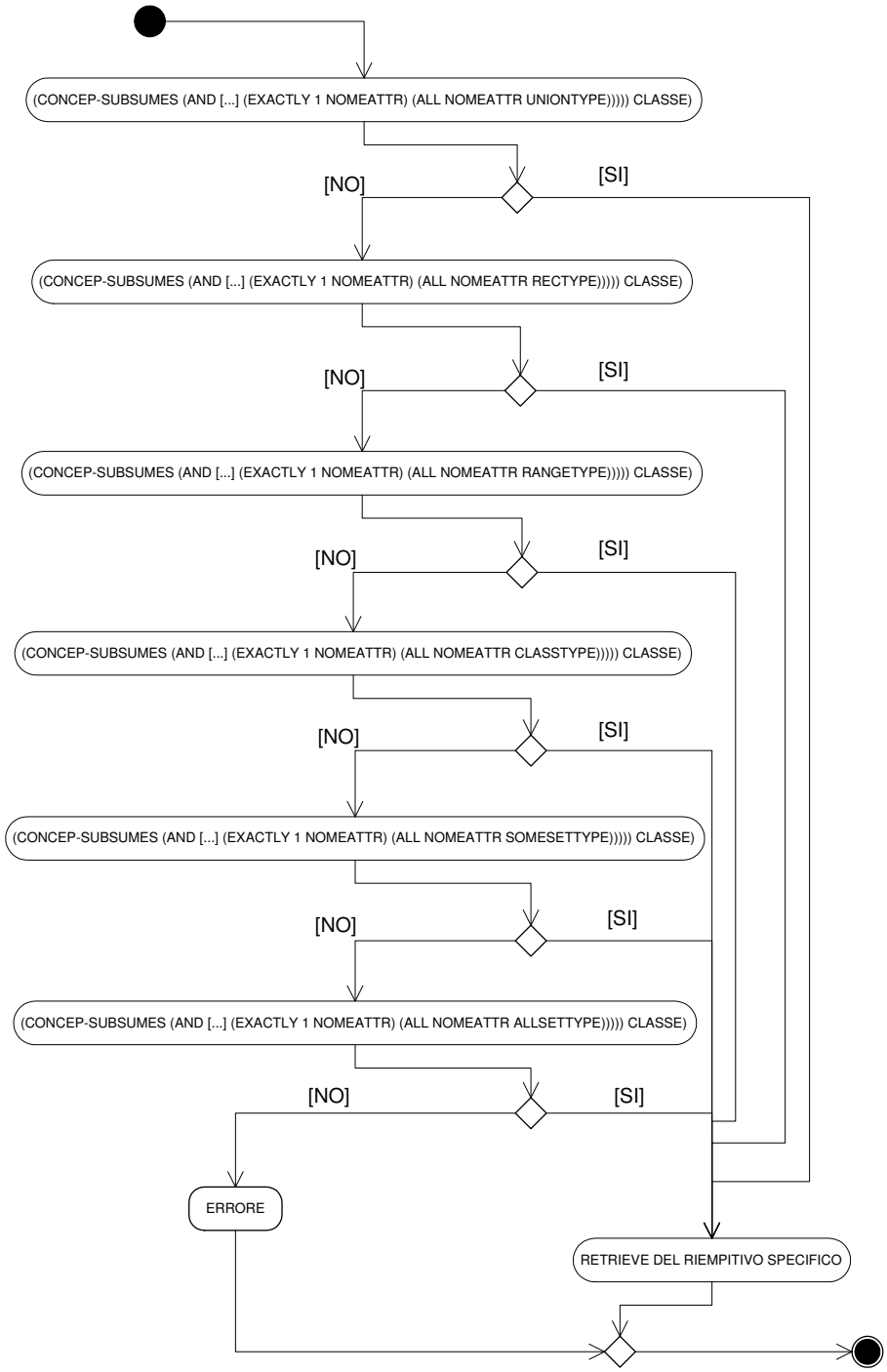

Figura 8.16: Activity Diagram: verifica riempitivo

8.6 ABox Module

Le funzionalità espletate dal modulo relativo la parte estensionale sono quelle richieste dai requisiti #8. . .#11. Il diagramma in figura 8.17 riporta le classi coinvolte per la sua realizzazione.

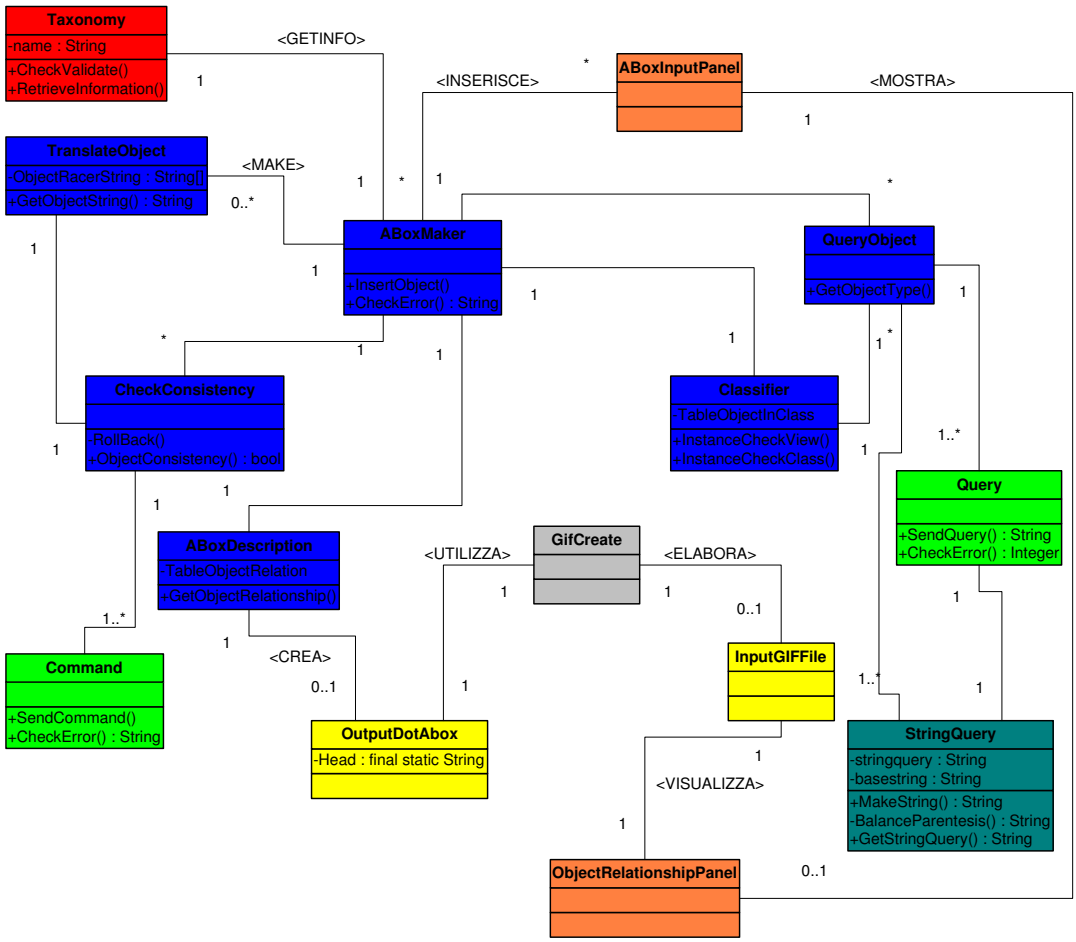

Figura 8.17: Class Diagram: realizzazione del modulo ABox

8.6.1 Funzionamento

La realizzazione di questa parte è basata sul linguaggio d'interfaccia introdotto in 7.3. Il funzionamento di questo modulo è riassunto nel diagramma 8.18. A livello implementa-

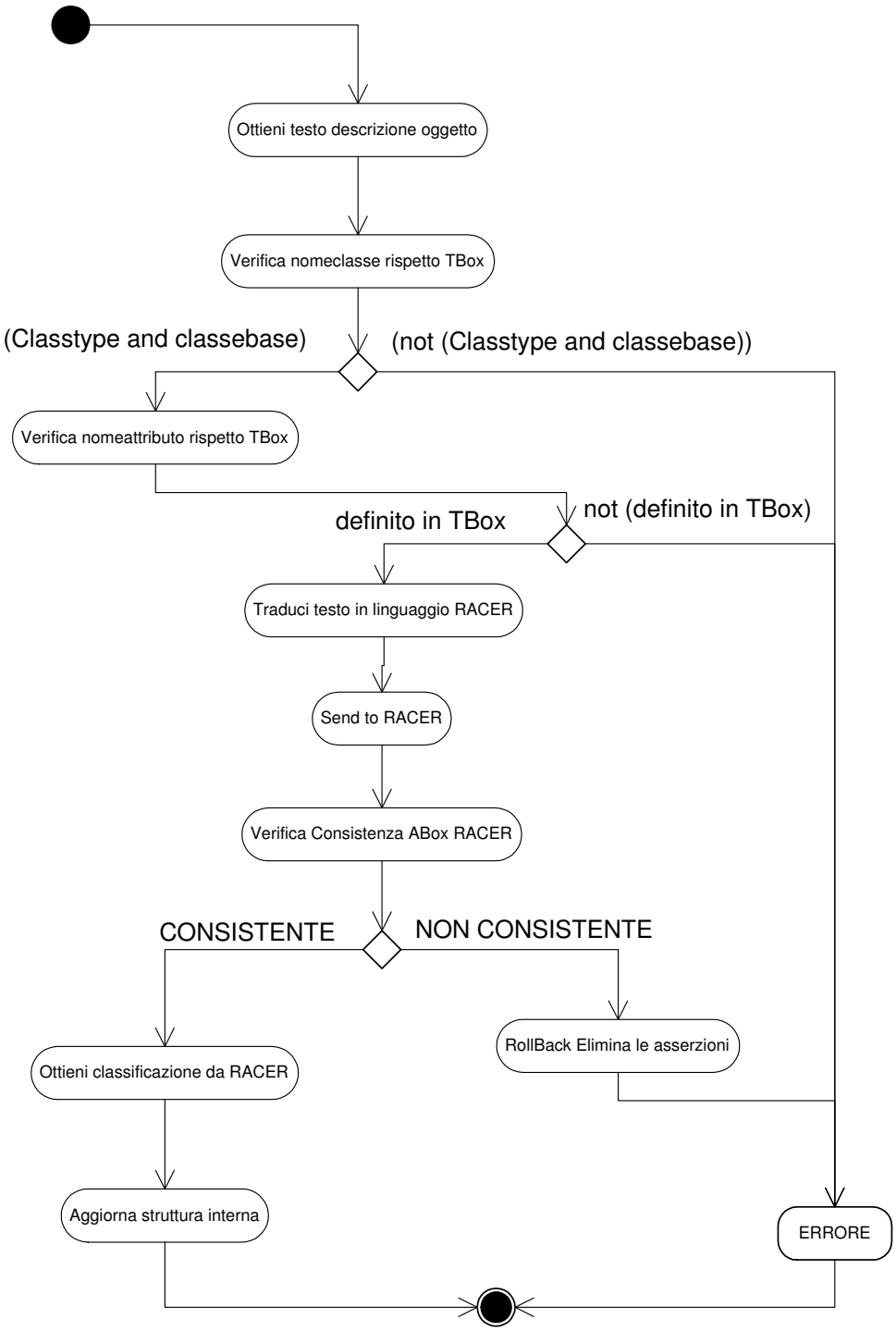

Figura 8.18: Activity Diagram funzionamento ABox module

tivo è stato realizzato un automa che, nella traduzione in RLI, applica le regole viste nel capitolo 7 per realizzare una CWA “locale”. A tal proposito si occupa quindi di:

- esplicitare chiaramente quali sono gli attributi presenti, e non, nella descrizione di un record;
- contare gli elementi che formano i tipi set;
- utilizzare gli operatori corretti in relazione al tipo di valore (integer, real, string, oggetto) specificato come riempitivo per l’attributo. Le funzioni relative alla classificazione delle istanze sono basate esclusivamente su RACER.

Conclusioni

Il problema generale della traduzione è stato studiato in letteratura, in questa tesi è stato approfondito ed in particolare riferito a due precisi modelli di riferimento, ovvero **ODL**$_{I^3}$, usato in MOMIS/SEWASIE, e RACER uno dei KBRS tra i più completi e maggiormente diffusi. Questo ha comportato sia un lavoro teorico, per considerare gli aspetti caratteristici dei due modelli in questione, sia un lavoro implementativo per la realizzazione del software nel quale è stato indispensabile uno studio approfondito del sistema RACER.

Le funzionalità del prototipo sviluppato sono state illustrate con vari esempi. In particolare, dal punto di vista intensionale, le funzionalità di calcolo della sussunzione tra i tipi di uno schema, la consistenza e minimalità di uno schema. Queste funzionalità come detto negli obiettivi della tesi sono fondamentali nello sviluppo del progetto MOMIS/SEWASIE, in quanto forniscono il supporto al ragionamento ad un insieme significativo del linguaggio **ODL**$_{I^3}$ utilizzato in MOMIS/SEWASIE. Dal punto di vista estensionale, sono state illustrate le funzionalità di riconoscimento delle istanze, riferite ad istanze con struttura semplice che in qualche modo possono essere considerate canoniche, alle quali, quindi, si possono ricondurre anche altre istanze con struttura più complessa.

Queste funzionalità non sono al momento direttamente sfruttabili nel sistema MOMIS/SEWASIE, in quanto in quest'ultimo il ragionamento è limitato sulla parte intensionale. D'altra parte, nel contesto dell'integrazione delle informazioni è importante anche riferirsi e ragionare sulle istanze, ad esempio per rilevare mappings tra le sorgenti da integrare, come viene effettuato ad esempio in CLIO, [HMH$^+$01, cli01], impiegando un paradigma "mapping-byexample". Per questo motivo, tali funzionalità saranno probabilmente sfruttabili in una fase successiva dello sviluppo di MOMIS/SEWASIE.

Il prototipo è stand-alone ma essendo sviluppato con la stessa tecnologia (JAVA) utilizzata nel progetto MOMIS/SEWASIE la sua integrazione in tale progetto dovrebbe essere immediata; da questo punto di vista la parte più rlevante da realizzare è sicuramente la parte di interfaccia che si deve in particolare preoccupare di trasformare una espressione **ODL**$_{I^3}$ generica in una espressione nell'**ODL**$_{I^3}$ canonico introdotto nella tesi.

SINTASSI ODL$_{I^3}$

Si presenta la sintassi BNF del linguaggio **ODL**$_{I^3}$, comprendente sia gli elementi sintattici del linguaggio standard $\mathcal{ODL}$ che le estensioni:

⟨OdlSpecification⟩ ::= ⟨Definition⟩ ⟨DOT⟩
| ⟨Definition⟩ ⟨SEMI⟩
| ⟨Definition⟩ ⟨SEMI⟩ ⟨OdlSpecification⟩

⟨Definition⟩ ::= ⟨TypeDcl⟩
| ⟨ConstDcl⟩
| ⟨ExceptDcl⟩
| ⟨Interface⟩
| ⟨RuleDcl⟩
| ⟨ExtRuleDcl⟩
| ⟨ThesaurusRelation⟩
| ⟨Module⟩
| ⟨Source⟩
| ⟨WnAnnotation⟩
| ⟨error⟩

⟨TypeDcl⟩ ::= **typedef** ⟨TypeDeclarator⟩
| ⟨ConstrTypeSpec⟩

⟨TypeDeclarator⟩ ::= ⟨TypeSpec⟩ ⟨Declarators⟩

⟨Declarators⟩ ::= ⟨Declarator⟩
| ⟨Declarators⟩ ⟨COMMA⟩ ⟨Declarator⟩

⟨Declarator⟩ ::= ⟨Identifier⟩
| ⟨Identifier⟩ ⟨ArraySizeList⟩

⟨ArraySizeList⟩ ::= ⟨FixedArraySize⟩
| ⟨ArraySizeList⟩ ⟨FixedArraySize⟩

⟨TypeSpec⟩ ::= ⟨SimpleTypeSpec⟩
| ⟨ConstrTypeSpec⟩

⟨SimpleTypeSpec⟩ ::= ⟨BaseTypeSpec⟩
| ⟨TemplateTypeSpec⟩
| ⟨Identifier⟩

⟨BaseTypeSpec⟩ ::= ⟨FloatingPtType⟩
| ⟨IntegerType⟩
| ⟨CharType⟩
| ⟨BooleanType⟩
| ⟨OctetType⟩
| ⟨RangeType⟩
| ⟨DateType⟩
| ⟨AnyType⟩

⟨FloatingPtType⟩ ::= **float**
| **double**

⟨DateType⟩ ::= ⟨DATE⟩
| ⟨TIMESTAMP⟩

⟨IntegerType⟩ ::= ⟨LongIntType⟩
| **short**
| **unsigned long**
| **unsigned short**

⟨LongIntType⟩ ::= **integer**
| **int**
| **long**

⟨CharType⟩ ::= **char**

⟨BooleanType⟩ ::= **boolean**

⟨OctetType⟩ ::= **octet**

⟨RangeType⟩ ::= **range** ⟨LPAR⟩ ⟨RangeSpecifier⟩ ⟨RPAR⟩

⟨RangeSpecifier⟩ ::= ⟨SignedIntegerLiteral⟩ ⟨COMMA⟩
| ⟨SignedIntegerLiteral⟩ ⟨SignedIntegerLiteral⟩ ⟨COMMA⟩ ⟨PLUS⟩
| **infinite** ⟨MINUS⟩ **infinite** ⟨COMMA⟩ ⟨SignedIntegerLiteral⟩

⟨AnyType⟩ ::= **any**

⟨TemplateTypeSpec⟩ ::= ⟨ArrayType⟩
| ⟨StringType⟩
| ⟨CollectionType⟩

⟨ArrayType⟩ ::= **array** ⟨LEFT⟩ ⟨SimpleTypeSpec⟩ ⟨COMMA⟩ ⟨IntegerLiteral⟩ ⟨RIGHT⟩

⟨$$1⟩ ::=

⟨ArrayType⟩ ::= **array** ⟨LEFT⟩ ⟨SimpleTypeSpec⟩ ⟨RIGHT⟩ ⟨$$1⟩ **sequence** ⟨LEFT⟩ ⟨SimpleTypeSpec⟩ ⟨COMMA⟩ ⟨IntegerLiteral⟩ ⟨RIGHT⟩

⟨StringType⟩ ::= **string** ⟨LEPAR⟩ ⟨IntegerLiteral⟩ ⟨REPAR⟩
| **string**

⟨CollectionType⟩ ::= ⟨AttrCollectionSpecifier⟩ ⟨LEFT⟩
⟨SimpleTypeSpec⟩ ⟨RIGHT⟩

⟨AttrCollectionSpecifier⟩ ::= **set**
| **list**
| **bag**

⟨ConstrTypeSpec⟩ ::= ⟨StructType⟩
| ⟨UnionType⟩
| ⟨EnumType⟩

⟨StructType⟩ ::= **struct** ⟨Identifier⟩ ⟨LPAR⟩ ⟨MemberList⟩ ⟨RPAR⟩

⟨MemberList⟩ ::= ⟨Member⟩
| ⟨MemberList⟩ ⟨Member⟩

⟨Member⟩ ::= ⟨TypeSpec⟩ ⟨Declarators⟩ ⟨SEMI⟩

⟨UnionType⟩ ::= **union** ⟨Identifier⟩ **switch** ⟨LRPAR⟩
⟨SwitchTypeSpec⟩ ⟨RRPAR⟩ ⟨LPAR⟩
⟨SwitchBody⟩ ⟨RPAR⟩

⟨SwitchTypeSpec⟩ ::= ⟨IntegerType⟩
| ⟨CharType⟩
| ⟨BooleanType⟩
| ⟨EnumType⟩
| ⟨ScopedName⟩
| ⟨RangeType⟩

⟨SwitchBody⟩ ::= ⟨Case⟩
| ⟨Case⟩ ⟨SwitchBody⟩

⟨Case⟩ ::= ⟨CaseLabelList⟩ ⟨ElementSpec⟩ ⟨SEMI⟩

⟨CaseLabelList⟩ ::= ⟨CaseLabel⟩
| ⟨CaseLabel⟩ ⟨CaseLabelList⟩

⟨CaseLabel⟩ ::= **case** ⟨ConstExp⟩ ⟨COLON⟩
| **default** ⟨COLON⟩

⟨ElementSpec⟩ ::= ⟨TypeSpec⟩ ⟨Declarator⟩

⟨EnumType⟩ : : = **enum** ⟨Identifier⟩ ⟨LPAR⟩ ⟨EnumeratorList⟩ ⟨RPAR⟩

⟨EnumeratorList⟩ : : = ⟨Enumerator⟩

| ⟨EnumeratorList⟩ ⟨COMMA⟩ ⟨Enumerator⟩

⟨Enumerator⟩ : : = ⟨Identifier⟩

⟨ConstExp⟩ : : = ⟨OrExpr⟩

⟨OrExpr⟩ : : = ⟨XOrExpr⟩

| ⟨OrExpr⟩ ⟨VERT⟩ ⟨XOrExpr⟩

⟨XOrExpr⟩ : : = ⟨AndExpr⟩

| ⟨XOrExpr⟩ ⟨HAT⟩ ⟨AndExpr⟩

⟨AndExpr⟩ : : = ⟨ShiftExpr⟩

| ⟨AndExpr⟩ ⟨AMPER⟩ ⟨ShiftExpr⟩

⟨ShiftExpr⟩ : : = ⟨AddExpr⟩

| ⟨ShiftExpr⟩ ⟨DOUBLE_RIGHT⟩ ⟨AddExpr⟩

| ⟨ShiftExpr⟩ ⟨DOUBLE_LEFT⟩ ⟨AddExpr⟩

⟨AddExpr⟩ : : = ⟨MultExpr⟩

| ⟨AddExpr⟩ ⟨PLUS⟩ ⟨MultExpr⟩

| ⟨AddExpr⟩ ⟨MINUS⟩ ⟨MultExpr⟩

⟨MultExpr⟩ : : = ⟨UnaryExpr⟩

| ⟨MultExpr⟩ ⟨TIMES⟩ ⟨UnaryExpr⟩

| ⟨MultExpr⟩ ⟨SLASH⟩ ⟨UnaryExpr⟩

| ⟨MultExpr⟩ ⟨PERCENT⟩ ⟨UnaryExpr⟩

⟨UnaryExpr⟩ : : = ⟨Signes⟩ ⟨PrimaryExpr⟩

| ⟨PrimaryExpr⟩

⟨PrimaryExpr⟩ ::= ⟨Identifier⟩
| ⟨IntegerLiteral⟩
| ⟨FloatingPtLiteral⟩
| ⟨LRPAR⟩ ⟨OrExpr⟩ ⟨RRPAR⟩
| ⟨LRPAR⟩ ⟨error⟩ ⟨RRPAR⟩

⟨ConstDcl⟩ ::= **const** ⟨StringType⟩ ⟨Identifier⟩ ⟨EQUAL⟩
⟨StringLiteral⟩
| **const** ⟨CharType⟩ ⟨Identifier⟩ ⟨EQUAL⟩
⟨CharacterLiteral⟩
| **const** ⟨IntegerType⟩ ⟨Identifier⟩ ⟨EQUAL⟩
⟨ConstExp⟩
| **const** ⟨FloatingPtType⟩ ⟨Identifier⟩ ⟨EQUAL⟩
⟨ConstExp⟩

⟨SignedIntegerLiteral⟩ ::= ⟨IntegerLiteral⟩
| ⟨Signes⟩ ⟨IntegerLiteral⟩

⟨SignedFloatingPtLiteral⟩ ::= ⟨Signes⟩ ⟨FloatingPtLiteral⟩
| ⟨FloatingPtLiteral⟩

⟨Signes⟩ ::= ⟨MINUS⟩
| ⟨PLUS⟩

⟨ExceptDcl⟩ ::= **exception** ⟨Identifier⟩ ⟨LPAR⟩ ⟨OptMemberList⟩
⟨RPAR⟩

⟨OptMemberList⟩ ::=
| ⟨MemberList⟩

⟨ScopedName⟩ ::= ⟨Identifier⟩
| ⟨DOUBLE_COLON⟩ ⟨Identifier⟩
| ⟨ScopedName⟩ ⟨DOUBLE_COLON⟩ ⟨Identifier⟩

⟨Interface⟩ ::= ⟨InterfaceDcl⟩

⟨IntView⟩ ::= **interface** ⟨InterfaceName⟩
| **view** ⟨InterfaceName⟩

⟨InterfaceName⟩ : : = ⟨Identifier⟩

⟨InterfaceDcl⟩ : : = ⟨IntView⟩ ⟨COLON⟩ ⟨InheritanceSpec⟩
⟨OptTypePropertyList⟩ ⟨OptPersistenceDcl⟩
⟨SingleInterfaceBody⟩
| ⟨IntView⟩ ⟨COLON⟩ ⟨InheritanceSpec⟩
⟨OptTypePropertyList⟩ ⟨OptPersistenceDcl⟩
⟨SingleInterfaceBody⟩ ⟨InterfaceBodyUnionList⟩
| ⟨IntView⟩ ⟨OptTypePropertyList⟩
⟨OptPersistenceDcl⟩ ⟨SingleInterfaceBody⟩
| ⟨IntView⟩ ⟨OptTypePropertyList⟩
⟨OptPersistenceDcl⟩ ⟨SingleInterfaceBody⟩
⟨InterfaceBodyUnionList⟩

⟨InheritanceSpec⟩ : : = ⟨Identifier⟩
| ⟨InheritanceSpec⟩ ⟨COMMA⟩ ⟨Identifier⟩

⟨OptTypePropertyList⟩ : : =
| ⟨LRPAR⟩ ⟨OptExtentSpec⟩ ⟨OptKeySpec⟩
⟨OptCandKeySpec⟩ ⟨OptForKeySpec⟩
⟨OptJoinAttribute⟩ ⟨RRPAR⟩

⟨OptExtentSpec⟩ : : =
| **extent** ⟨ListExtent⟩

⟨ListExtent⟩ : : = ⟨Identifier⟩
| ⟨ListExtent⟩ ⟨COMMA⟩ ⟨Identifier⟩

⟨OptKeySpec⟩ : : =
| **key** ⟨Key⟩

⟨OptJoinAttribute⟩ : : =
| ⟨JOIN⟩ **attribute** ⟨OptJoinAttribute_type⟩
⟨LPAR⟩
⟨OptJoinAttribute_joinAttributeFunction⟩
⟨RPAR⟩

⟨OptJoinAttribute_type⟩ : = ⟨SimpleTypeSpec⟩

⟨OptJoinAttribute_joinAttributeFunction⟩ ::= ⟨DottedName⟩ ⟨COLON⟩ ⟨StringLiteral⟩
| ⟨OptJoinAttribute_joinAttributeFunction⟩ ⟨COMMA⟩ ⟨OptJoinAttribute_joinAttributeFunction⟩

⟨OptCandKeySpec⟩ ::=
| ⟨CandKeySpecList⟩

⟨CandKeySpecList⟩ ::= ⟨CandKeySpec⟩
| ⟨CandKeySpecList⟩ ⟨CandKeySpec⟩

⟨CandKeySpec⟩ ::= **candidate_key** ⟨Identifier⟩ ⟨Key⟩

⟨OptForKeySpec⟩ ::=
| ⟨ForKeySpecList⟩

⟨ForKeySpecList⟩ ::= ⟨ForKeySpec⟩
| ⟨ForKeySpecList⟩ ⟨ForKeySpec⟩

⟨ForKeySpec⟩ ::= **foreign_key** ⟨Identifier⟩ ⟨LRPAR⟩ ⟨ForeignKeyList⟩ ⟨RRPAR⟩ **references** ⟨Identifier⟩ ⟨OptRefKeyList⟩

⟨OptRefKeyList⟩ ::=
| ⟨LRPAR⟩ ⟨ForeignKeyList⟩ ⟨RRPAR⟩

⟨ForeignKeyList⟩ ::= ⟨Identifier⟩
| ⟨ForeignKeyList⟩ ⟨COMMA⟩ ⟨Identifier⟩

⟨Key⟩ ::= ⟨LRPAR⟩ ⟨PropertyList⟩ ⟨RRPAR⟩

⟨PropertyList⟩ ::= ⟨PropertyName⟩
| ⟨PropertyList⟩ ⟨COMMA⟩ ⟨PropertyName⟩

⟨PropertyName⟩ ::= ⟨Identifier⟩

⟨OptPersistenceDcl⟩ ::=
| **persistent**
| **transient**

⟨InterfaceBodyUnionList⟩ ::= ⟨InterfaceBodyUnion⟩
| ⟨InterfaceBodyUnionList⟩ ⟨InterfaceBodyUnion⟩

⟨InterfaceBodyUnion⟩ ::= **union** ⟨InterfaceBodyUnionName⟩ ⟨SingleInterfaceBody⟩

⟨InterfaceBodyUnionName⟩ ::= ⟨Identifier⟩

⟨SingleInterfaceBody⟩ ::= ⟨LPAR⟩ ⟨InterfaceBody⟩ ⟨RPAR⟩

⟨InterfaceBody⟩ ::=
| ⟨Export⟩ ⟨SEMI⟩
| ⟨Export⟩ ⟨SEMI⟩ ⟨InterfaceBody⟩

⟨Export⟩ ::= ⟨TypeDcl⟩
| ⟨ConstDcl⟩
| ⟨ExceptDcl⟩
| ⟨AttrDcl⟩
| ⟨RelDcl⟩
| ⟨OpDcl⟩

⟨AttrDcl⟩ ::= ⟨AttrDclNoMapping⟩ ⟨OptMappingRuleDcl⟩

⟨AttrDclNoMapping⟩ ::= ⟨OptReadonly⟩ **attribute** ⟨DomainType⟩ ⟨AttributeName⟩ ⟨OptFixedArraySize⟩

⟨OptReadonly⟩ ::=
| **readonly**

⟨AttributeName⟩ ::= ⟨Identifier⟩
| ⟨Identifier⟩ ⟨QUESTION⟩

⟨DomainType⟩ ::= ⟨SimpleTypeSpec⟩
| ⟨StructType⟩
| ⟨EnumType⟩

⟨OptFixedArraySize⟩ ::=
| ⟨FixedArraySize⟩

⟨FixedArraySize⟩ ::= ⟨LEPAR⟩ ⟨IntegerLiteral⟩ ⟨REPAR⟩

⟨OptMappingRuleDcl⟩ : : =

| **mapping** ⟨LRPAR⟩ ⟨StringLiteral⟩ ⟨OptMappingRuleDclLocalOpt⟩ ⟨RRPAR⟩

⟨OptMappingRuleDclLocalOpt⟩

| ⟨OptMappingRuleDclLocal⟩

⟨OptMappingRuleDclLocal⟩ ⟨OptMappingRuleDclLocalOpt⟩ ⟨COMMA⟩ ⟨OptMappingRuleDclLocalInterfaceName⟩ ⟨OptMappingRuleDclLocalOptLocalResolutionFunction⟩

⟨OptMappingRuleDclLocalInterfaceName⟩ ⟨DottedName⟩

⟨OptMappingRuleDclLocalOptLocalResolutionFunction⟩

| ⟨COLON⟩ ⟨StringLiteral⟩

⟨LocalAttributeName⟩ : : = ⟨LocalClassName⟩ ⟨DOT⟩ ⟨Identifier⟩

⟨LocalClassName⟩ : : = ⟨Identifier⟩ ⟨DOT⟩ ⟨Identifier⟩

⟨RelDcl⟩ : : = **relationship** ⟨TargetOfPath⟩ ⟨Identifier⟩ **inverse** ⟨InverseTraversalPath⟩ ⟨OptOrderBy⟩

⟨TargetOfPath⟩ : : = ⟨Identifier⟩

| ⟨RelCollectionType⟩ ⟨LEFT⟩ ⟨Identifier⟩ ⟨RIGHT⟩

⟨RelCollectionType⟩ : : = **set**

| **list**

⟨InverseTraversalPath⟩ : : = ⟨Identifier⟩ ⟨DOUBLE_COLON⟩ ⟨Identifier⟩

⟨OptOrderBy⟩ : : =

| ⟨LPAR⟩ **order_by** ⟨ScopedNameList⟩ ⟨RPAR⟩

⟨ScopedNameList⟩ : : = ⟨ScopedName⟩

| ⟨ScopedNameList⟩ ⟨COMMA⟩ ⟨ScopedName⟩

⟨OpDcl⟩ : : = ⟨OptOneway⟩ ⟨OperTypeSpec⟩ ⟨Identifier⟩ ⟨ParameterDcls⟩ ⟨OptRaisesExpr⟩ ⟨OptContextExpr⟩

⟨OptOneway⟩	::=	
	|	**oneway**
⟨OperTypeSpec⟩	::=	⟨SimpleTypeSpec⟩
	|	**voi**
⟨ParameterDcls⟩	::=	⟨LRPAR⟩ ⟨ParamDclList⟩ ⟨RRPAR⟩
	|	⟨LRPAR⟩ ⟨RRPAR⟩
⟨ParamDclList⟩	::=	⟨ParamDcl⟩
	|	⟨ParamDclList⟩ ⟨COMMA⟩ ⟨ParamDcl⟩
⟨ParamDcl⟩	::=	⟨ParamAttribute⟩ ⟨SimpleTypeSpec⟩ ⟨Declarator⟩
⟨ParamAttribute⟩	::=	
	|	**in**
	|	**out**
	|	**inout**
⟨OptRaisesExpr⟩	::=	
	|	**raises** ⟨LRPAR⟩ ⟨ScopedNameList⟩ ⟨RRPAR⟩
⟨OptContextExpr⟩	::=	
	|	**context** ⟨LRPAR⟩ ⟨StringLiteralList⟩ ⟨RRPAR⟩
⟨StringLiteralList⟩	::=	⟨StringLiteral⟩
	|	⟨StringLiteralList⟩ ⟨COMMA⟩ ⟨StringLiteral⟩
⟨ExtRuleDcl⟩	::=	**extrule** ⟨Identifier⟩ ⟨ExtRuleSpec⟩
⟨ExtRuleSpec⟩	::=	⟨ForAll⟩ ⟨Identifier⟩ **in** ⟨ExtRuleType⟩
⟨ExtRuleType⟩	::=	⟨ExtRuleIsa⟩
	|	⟨ExtRuleBottom⟩
⟨ExtRuleBottom⟩	::=	⟨LRPAR⟩ ⟨LocalClassName⟩ **and** ⟨LocalClassName⟩ ⟨RRPAR⟩ **then** ⟨Identifier⟩ **in** **bottom**
⟨ExtRuleIsa⟩	::=	⟨LocalClassName⟩ **then** ⟨Identifier⟩ **in** ⟨LocalClassName⟩
⟨RuleDcl⟩	::=	**rule** ⟨Identifier⟩ ⟨RuleSpec⟩

⟨RuleSpec⟩ : : = ⟨ForAll⟩ ⟨Identifier⟩ **in** ⟨Identifier⟩ ⟨COLON⟩
⟨RuleBodyList⟩ **then** ⟨RuleBodyList⟩
| ⟨LPAR⟩ **case of** ⟨Identifier⟩ ⟨COLON⟩ ⟨CaseList⟩
⟨RPAR⟩

⟨ForAll⟩ : : = **for all**
| **forall**

⟨RuleBodyList⟩ : : = ⟨LRPAR⟩ ⟨RuleBodyList⟩ ⟨RRPAR⟩
| ⟨RuleBody⟩
| ⟨RuleBodyList⟩ **and** ⟨RuleBody⟩

⟨$$2⟩ : : =

⟨RuleBody⟩ : : = ⟨DottedName⟩ ⟨RuleConstOp⟩ ⟨OptRuleCast⟩
⟨LiteralValue⟩ ⟨$$2⟩ ⟨DottedName⟩
⟨RuleConstOp⟩ ⟨OptRuleCast⟩ ⟨DottedName⟩
| ⟨DottedName⟩ **in** ⟨DottedName⟩
| ⟨ForAll⟩ ⟨Identifier⟩ **in** ⟨DottedName⟩ ⟨COLON⟩
⟨RuleBodyList⟩
| **exists** ⟨Identifier⟩ **in** ⟨DottedName⟩ ⟨COLON⟩
⟨RuleBodyList⟩
| ⟨DottedName⟩ ⟨EQUAL⟩ ⟨SimpleTypeSpec⟩
⟨Identifier⟩ ⟨LRPAR⟩ ⟨DottedLiteralList⟩
⟨RRPAR⟩

⟨DottedLiteralList⟩ : : = ⟨DottedLiteral⟩
| ⟨DottedLiteralList⟩ ⟨COMMA⟩ ⟨DottedLiteral⟩

⟨DottedLiteral⟩ : : = ⟨OptSimpleTypeSpec⟩ ⟨DottedName⟩
| ⟨OptSimpleTypeSpec⟩ ⟨LiteralValue⟩

⟨OptSimpleTypeSpec⟩ : : =
| ⟨SimpleTypeSpec⟩

⟨RuleConstOp⟩ ::= ⟨EQUAL⟩
| ⟨GREATEQUAL⟩
| ⟨LESSEQUAL⟩
| ⟨LEFT⟩
| ⟨RIGHT⟩

⟨LiteralValue⟩ ::= ⟨SignedFloatingPtLiteral⟩
| ⟨SignedIntegerLiteral⟩
| ⟨CharacterLiteral⟩
| ⟨StringLiteral⟩

⟨DottedName⟩ ::= ⟨DottedName⟩ ⟨DOT⟩ ⟨Identifier⟩
| ⟨Identifier⟩

⟨OptRuleCast⟩ ::=
| ⟨LRPAR⟩ ⟨SimpleTypeSpec⟩ ⟨RRPAR⟩

⟨CaseList⟩ ::= ⟨CaseSpec⟩
| ⟨CaseList⟩ ⟨CaseSpec⟩

⟨CaseSpec⟩ ::= ⟨LiteralValue⟩ ⟨COLON⟩ ⟨DottedName⟩

⟨ThesaurusRelation⟩ ::= ⟨LocalAttributeName⟩ ⟨ThesRelType⟩
⟨LocalAttributeName⟩ ⟨ThesRelOptProducer⟩
⟨ThesRelOptValidated⟩
| ⟨LocalClassName⟩ ⟨ThesRelType⟩
⟨LocalClassName⟩ ⟨ThesRelOptProducer⟩
⟨ThesRelOptValidated⟩
| ⟨LocalClassName⟩ ⟨ThesRelType⟩
⟨LocalAttributeName⟩ ⟨ThesRelOptProducer⟩
⟨ThesRelOptValidated⟩
| ⟨LocalAttributeName⟩ ⟨ThesRelType⟩
⟨LocalClassName⟩ ⟨ThesRelOptProducer⟩
⟨ThesRelOptValidated⟩

⟨ThesRelType⟩ : : = **syn**
| **bt**
| **nt**
| **rt**

⟨ThesRelOptProducer⟩ : : =
| ⟨IntegerLiteral⟩

⟨ThesRelOptValidated⟩ : : =
| **true**
| **false**

⟨Module⟩ : : = **module** ⟨ModuleIdentifier⟩ ⟨LPAR⟩ ⟨OdlSpecification⟩ ⟨RPAR⟩

⟨ModuleIdentifier⟩ : : = ⟨Identifier⟩

⟨Source⟩ : : = **source** ⟨SourceType⟩ ⟨SourceIdentifier⟩ ⟨SourceOdli3Params⟩ ⟨LPAR⟩ ⟨OdlSpecification⟩ ⟨RPAR⟩

⟨SourceOdli3Params⟩ : : =
| ⟨LRPAR⟩ ⟨SourceOdli3ParamsLines⟩ ⟨RRPAR⟩

⟨SourceOdli3ParamsLines⟩ ⟨Identifier⟩ ⟨EQUAL⟩ ⟨StringLiteral⟩
| ⟨SourceOdli3ParamsLines⟩ ⟨COMMA⟩ ⟨SourceOdli3ParamsLines⟩

⟨SourceIdentifier⟩ : : = ⟨Identifier⟩

⟨SourceType⟩ : : = **relational**
| **nfrelational**
| **object**
| **file**
| **semistructured**

⟨WnAnnotation⟩ : : = ⟨WNANNOTATION⟩ ⟨DottedName⟩
⟨LEMMAVALUE⟩ ⟨EQUAL⟩ ⟨StringLiteral⟩
⟨COMMA⟩ ⟨LEMMASYNTACTICCATEGORY⟩
⟨EQUAL⟩ ⟨IntegerLiteral⟩ ⟨COMMA⟩
⟨LEMMASENSENUMBER⟩ ⟨EQUAL⟩
⟨IntegerLiteral⟩

COMANDI RACER

In questa appendice viene riportato l'elenco delle funzioni messe a disposizione dal linguaggio d'interfaccia di RACER, suddivise per categoria.

Knowledge Base Management Functions

- (in-knowledge-base tbox abox)
- (racer-read-file pathname)
- (racer-read-document URL)
- (include-kb pathname)
- (daml-read-file pathname)
- (daml-read-document URL)
- (owl-read-file pathname)
- (owl-read-document URL)
- (mirror URL mirror$_{URL}$)
- (kb-ontologies KBNAME)
- (save-kb pathname)

TBox Management

- (in-tbox tbox)
- (init-tbox tbox)
- (signature parameter)
- (ensure-tbox-signature tbox)
- (tbox-signature tbox)
- (current-tbox)
- (save-tbox pathname)
- (forget-tbox tbox)
- (delete-tbox tbox)
- (delete-all-tboxes)
- (create-tbox-clone tbox)
- (clone-tbox tbox)
- (find-tbox tbox)
- (tbox-name tbox)
- (clear-default-tbox)
- (associated-aboxes tbox)
- (xml-read-tbox-file pathname)
- (rdfs-read-tbox-file pathname)

ABox Management

- (in-abox abox)
- (init-abox abox)
- (ensure-abox-signature abox)
- (abox-signature abox)
- (kb-signature KBNAME)
- (current-abox)
- (save-abox pathname)
- (forget-abox abox)
- (delete-abox abox)
- (delete-all-aboxes)
- (create-abox-clone abox)
- (clone-abox abox)
- (find-abox abox)
- (abox-name abox)
- (tbox abox)
- (associated-tbox abox)
- (set-associated-tbox abox tbox)

Knowledge Base Declarations

Built-in Concepts

- top
- bottom

Concept Axioms

- (implies C1 C2)
- (equivalent C1 C2)
- (disjoint CN1. . .CNn)
- (define-primitive-concept CN C)
- (define-concept CN C)
- (define-disjoint-primitive-concept CN GNL C)
- (add-concept-axiom tbox C1 C2)
- (add-disjointness-axiom tbox CN GN)

Role Declarations

- (define-primitive-role RN)
- (define-primitive-attribute AN)
- (add-role-axioms tbox RN)
- (functional RN)
- (role-is-functional RN)

- (transitive RN)
- (role-is-transitive RN)
- (inverse RN inverse$_{role}$)
- (inverse-of-role RN inverse$_{role}$)
- (roles-equivalent RN1 RN2)
- (roles-equivalent-1 RN1 RN2)
- (domain RN C)
- (role-has-domain RN C)
- (attribute-has-domain AN C)
- (range RN C)
- (role-has-range RN C)
- (attribute-has-range AN D)
- (implies-role RN1 RN2)
- (role-has-parent RN1 RN2)

Concrete Domain Attribute Declaration

- (define-concrete-domain-attribute AN type domain)

Assertions

- (instance IN C)
- (add-concept-assertion abox IN C)

- (forget-concept-assertion abox IN C)
- (related IN1 IN2 R)
- (add-role-assertion abox IN1 IN2 R)
- (forget-role-assertion abox IN1 IN2 R)
- (forget-disjointness-axiom tbox CN GN)
- (forget-disjointness-axiom-statement tbox concepts)
- (define-distinct-individual IN)
- (state forms)
- (forget forms) forget-statement tbox abox statements)
- (add-constraint-assertion abox constraint)
- (constraints forms)
- (add-attribute-assertion abox IN ON AN)
- (constrained IN ON AN)

Reasoning Modes

- *auto-classify*
- *auto-realize*

Evaluation Functions and Queries

Queries for Concept Terms

- (concept-satisfiable? C)

- (concept-satisfiable-p C tbox)
- (concept-subsumes? C1 C2)
- (concept-subsumes-p C1 C2 tbox)
- (concept-equivalent? C1 C2)
- (concept-equivalent-p C1 C2 tbox)
- (concept-disjoint? C1 C2)
- (concept-disjoint-p C1 C2 tbox)
- (concept? CN)
- (concept-p CN tbox)
- (concept-is-primitive? CN)
- (concept-is-primitive-p CN tbox)
- (alc-concept-coherent C)

Role Queries

- (role-subsumes? R1 R2)
- (role-subsumes-p R1 R2 tbox)
- (role? R)
- (role-p R tbox)
- (transitive? R)
- (transitive-p R tbox)
- (feature? R)

- (feature-p R tbox)
- (cd-attribute? AN)
- (cd-attribute-p AN tbox)
- (symmetric? R)
- (symmetric-p R tbox)
- (reflexive? R)
- (reflexive-p R tbox)
- (atomic-role-inverse R tbox)
- (role-inverse R)
- (role-domain RN)
- (atomic-role-domain RN)
- (role-range RN)
- (atomic-role-range RN)
- (attribute-domain AN)
- (attribute-domain-1 AN)

TBox Evaluation Functions

- (classify-tbox)
- (check-tbox-coherence)
- (tbox-classified-p tbox)
- (tbox-classified?)

- (tbox-prepared-p tbox)
- (tbox-prepared?)
- (tbox-cyclic-p tbox)
- (tbox-cyclic?)
- (tbox-coherent-p tbox)
- (tbox-coherent?)
- (get-tbox-language)
- (get-meta-constraint)
- (get-concept-definition CN)
- (get-concept-negated-definition CN)

ABox Evaluation Functions

- (realize-abox)
- (abox-realized-p abox)
- (abox-realized?)
- (abox-prepared-p abox)
- (abox-prepared?)
- (compute-all-implicit-role-fillers)
- (compute-implicit-role-fillers)
- (get-abox-language)

ABox Queries

- (abox-consistent-p)
- (abox-consistent?)
- (check-abox-coherence)
- (individual-instance? IN C)
- (constraint-entailed? constraint)
- (individuals-related? IN1 IN2 R)
- (individual-equal? IN1 IN2)
- (individual-not-equal? IN1 IN2)
- (individual? IN)
- (cd-object? ON
- (cd-object-p ON tbox)

Retrieval

TBox Retrieval

- (taxonomy tbox)
- (concept-synonyms CN)
- (atomic-concept-synonyms CN tbox)
- (concept-descendants C)
- (atomic-concept-descendants C tbox)

- (concept-ancestors C)
- (atomic-concept-ancestors C tbox)
- (concept-children C)
- (atomic-concept-children C tbox)
- (concept-parents C)
- (atomic-concept-parents C tbox)
- (role-descendants R)
- (atomic-role-descendants R tbox)
- (role-ancestors R)
- (atomic-role-ancestors R tbox)
- (role-children R)
- (atomic-role-children R tbox)
- (role-parents R)
- (atomic-role-parents R tbox)
- (role-synonyms R)
- (atomic-role-synonyms R tbox)
- (all-tboxes)
- (all-atomic-concepts)
- (all-equivalent-concepts)
- (all-roles)

- (all-features)
- (all-attributes)
- (attribute-type)
- (all-transitive-roles)
- (describe-tbox)
- (describe-concept CN)
- (describe-role R)

ABox Retrieval

- (individual-direct-types IN)
- (most-specific-instantiators IN abox)
- (individual-types IN)
- (instantiators IN abox)
- (concept-instances C)
- (retrieve-concept-instances C abox candidates)
- (individual-fillers IN R)
- (retrieve-individual-fillers IN R abox)
- (individual-attribute-fillers IN AN)
- (retrieve-individual-attribute-fillers IN AN)
- (told-value ON)
- (retrieve-related-individuals R abox)

- (related-individuals R)
- (retrieve-individual-filled-roles IN1 IN2 abox)
- (retrieve-direct-predecessors R IN abox)
- (all-aboxes)
- (all-individuals)
- (all-concept-assertions-for-individual IN)
- (all-concept-assertions)
- (all-role-assertions)
- (all-constraints)
- (all-attribute-assertions)
- (describe-abox)
- (describe-individual IN)

Reporting Errors and Inefficiencies

- (logging-on filename)
- (logging-off)

Bibliografia

[ABdR99] Carlos Areces, Wiet Bouma, and Maarten de Rijke. Description logics and feature interaction. In *Description Logics*, 1999.

[AFM03] A. Artale, E. Franconi, and F. Mandreoli. Description logics for modelling dynamic information, 2003.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *SIGMOD*, pages 159–173. ACM Press, 1989.

[Apt90] K. R. Apt. Logic programming. In J. Van Leeuwen, editor, *Formal Models and Semantics*, volume 2, chapter 10, pages 493–574. Elsevier., Amsterdam,New York, Oxford, Tokyo, 1990.

[Atz93] P. Atzeni, editor. *LOGIDATA$^+$: Deductive Databases with Complex Objects*, volume 701 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg - Germany, 1993.

[Baa90] F. Baader. Terminological cycles in KL-ONE-based knowledge representation languages. In *8th National Conference of the American Association for Artificial Intelligence*, volume 2, pages 621–626, Boston, Mass., USA, 1990.

[Baa91] F. Baader. Augmenting concept languages by transitive closure of roles: An alternative to terminological cycles. In *12th International Joint Conference on Artificial Intelligence.*, Sydney, Australia, 1991.

[BB03] Alexander Borgida and Ronald J. Brachman. Conceptual modeling with description logics. In *Description Logic Handbook*, pages 349–372, 2003.

[BBB+95] J. P. Ballerini, D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. A semantics driven query optimizer for OODBs. In A. Borgida, M. Lenzerini, D. Nardi, and B. Nebel, editors, *DL95 - Intern. Workshop on Description Logics*, volume 07.95 of *Dip. di Informatica e Sistemistica - Univ. di Roma La Sapienza - Rapp. Tecnico*, pages 59–62, Roma, June 1995.

[BBC+79] Ronald J. Brachman, R. Bobrow, P. Cohen, Klovstad Klovstad, B. L. Webber, and W. A. Woods. Research in natural language understanding. Technical Report 4274, Bolt Beranek and Newman Inc., aug 1979.

[BBLS95] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency checking in complex object database schemata with integrity constraints. In P. Atzeni and V. Tannen, editors, *5th Int. Workshop on Database Programming Languages*, Lecture Notes in Computer Science, pages 48–57, Gubbio, Italy, September 1995. Springer-Verlag.

[BBMR89] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *SIGMOD*, pages 58–67, Portland, Oregon, 1989.

[BBSV97a] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. ODB-QOptimizer: a tool for semantic query optimization in OODB. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, April 1997.

[BBSV97b] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. ODB-Tools: a description logics based tool for schema validation and semantic query optimization in Object Oriented Databases. In *Proc. of Int. Conference of the Italian Association for Artificial Intelligence (AI*IA97)*, Rome, 1997.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[Ber02] D. Berardi. Using dls to reason on uml class diagrams, 2002.

[BFG+] Kerstin Bücher, Yves Forkl, Günther Görz, Martin Klarner, and Bernd Ludwig. Discourse and application modeling for dialogue systems.

[BH91a] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *12th International Joint Conference on Artificial Intelligence, IJCAI-91*, pages 452–457, Sydney, Australia, 1991.

[BH91b] Franz Baader and Philipp Hanschke. A scheme for integrating concrete domains into concept languages. Technical Report RR-91-10, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH
Erwin-Schrödinger Strasse
Postfach 2080
67608 Kaiserslautern
Germany, 1991.

[BH93] Franz Baader and Philipp Hanschke. Extensions of concept languages for a mechanical engineering application. In *Proceedings of the 16th German Conference on Artificial Intelligence*, pages 132–143. Springer-Verlag, 1993.

[BL85] R. J. Brachman and H. J. Levesque. A fundamental tradeoff in knowledge representation and reasoning. In *Reading in Knowledge Representation*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1985.

[BMPSR99] Ronald J. Brachman, D. L. McGuinness, P. F. Pathel-Schneider, and A. L. Resnik. Reducing classic to pratice: Knowledge representation theory meets reality. *Artificial Intelligence*, pages 203–237, 1999.

[BN94a] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[BN94b] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[Bra77] R. J. Brachman. Pattern associativity and the retrieval of semantic networks. pages 3–50, New York, 1977. Accademic Press.

[BS85] R. J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[BSKB] Olivier Bodenreider, Barry Smith, Anand Kumar, and Anita Burgun. Investigating subsumption in dl-based terminologies: Case study in SNOMED CT.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984.

[Cat96] R. G. G. Cattel. *The Object Database Standard - ODGM93*. Morgan Kaufmann, 1996.

[CGL99] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Representing and reasoning on XML documents: A description logic approach. *Journal of Logic and Computation*, 9(3):295–318, 1999.

[cli01] The clio project: Managing heterogeneity. 30(1):78, March 2001.

[CLN94] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. A unified framework for class based representation formalisms. In J. Doyle, E. Sandewall, and P. Torasso, editors, *Proc. of the 4th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR'94)*, pages 109–120. Morgan Kaufmann, 1994.

[D2.ry] SEWASIE Deliverable D2.1. Specification of the general framework for the multilingual semantic enrichment processes and of the semantically enriched data stores, 2003, febrary.

[D.B] C. Sartori D.Beneventano, S. Bergamaschi. Description logics for semantic query optimization in object-oriented database systems. *ACM Trans. on Database Systems*, 28.

[D.B94] D.Beneventano. *Uno strumento di inferenza nelle basi di dati ad oggetti: la sussunzione*. PhD thesis, Dip. di Elettronica, Informatica e Sistemistica, Università di Bologna, Bologna, 1994.

[DB01] S. Castano A. Corni R. Guidetti G. Malvezzi M. Melchiori e M. Vincini D. Beneventano, S. Bergamaschi. Modal logics and the two-variable fragment. In *Annual Conference of the European Association for Computer Science Logic CSL'01*, LNCS, Paris, France, 2001. Springer Verlag.

[DBSB90] Premkumar T. Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. LaSSIE: a knowledge-based software information system. In *International Conference on Software Engineering*, pages 249–261, 1990.

[GKS02] M. Gabsdil, A. Koller, and K. Striegnitz. Natural language and inference in a computer game, 2002.

[Hay79] Patrick J. Hayes. The logic of frame. In D. Metzing, editor, *Frame Conception and Text Understanding*, pages 95–128. Walter de Gruyter and Co., Berlin, 1979.

[HM99] Volker Haarslev and Ralf Möller. RACE system description. In *Description Logics*, 1999.

[HM00] Volker Haarslev and Ralf Möller. Expressive abox reasoning with number restrictions, role hierarchies, and transitively closed roles. In A.G. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the Seventh International Conference on Knowledge Representation and Reasoning (KR2000)*, pages 273–284. Morgan Kaufmann, 2000.

[HM01] Volker Haarslev and Ralf Möller. Racer system description. In *Proceedings of the First International Joint Conference on Automated Reasoning*, pages 701–706. Springer-Verlag, 2001.

[HM04] Volker Haarslev and Ralf Möller. *RACER User's Guide and Reference Manual Version 1.7.19*. Concordia University, Montreal, Canada, april 2004.

[HMH+01] Mauricio A. Hernandez, Renee J. Miller, Laura M. Haas, Lingling Yan, C. T. Howard Ho, and Xuqing Tian. Clio: A semi-automatic tool for schema mapping. In *SIGMOD Record*, 2001.

[HMT01] V. Haarslev, R. Möller, and A.-Y. Turhan. Exploiting pseudo models for tbox and abox reasoning in expressive description logics. In *Proceedings of the International Joint Conference on Automated Reasoning IJCAR'01*, LNAI. Springer Verlag, 2001.

[HST99] Ian Horrocks, Ulrike Sattler, and Stephan Tobies. Practical reasoning for expressive description logics. In Harald Ganzinger, David McAllester, and Andrei Voronkov, editors, *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR'99)*, number 1705, pages 161–180. Springer-Verlag, 1999.

[HST00] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with individuals for the description logic shiq. In David MacAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction (CADE-17)*, number 1831 in Lecture Notes in Computer Science, Germany, 2000. Springer Verlag.

[JGC01] Claudio Bartolini Javier Gonzalez-Castillo, David Trastour. Description logics for matchmaking of services, 2001.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL*, pages 111–119, 1987.

[KLSS95] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The information manifold. In C. Knoblock and A. Levy, editors, *In Working Notes of the AAAI Spring Symposium on Information Gathering from Heterogeneous*, Stanford University, Stanford, California, 1995.

[Llo87] J. W. Lloyd. *Logic Programming*. Springer-Verlag, Berlin, 2nd edition, 1987.

[LR89a] C. Lecluse and P. Richard. Modelling complex structures in object-oriented databases. In *Symp. on Principles of Database Systems*, pages 362–369, Philadelphia, PA, 1989.

[LR89b] C. Lecluse and P. Richard. The O_2 database programming language. In *15th Int. Conf. on Very Large Databases*, pages 411–422, Amsterdam, February 1989.

[LSW00] C. Lutz, U. Sattler, and F. Wolter. Information integration: the momis project demonstration. In *ACM International Conference on Very Large Data Bases (VLDB 2000)*, Cairo, Egypt, 2000.

[Lut99] Carsten Lutz. The complexity of reasoning with concrete domains. LTCS-Report 99-01, LuFg Theoretical Computer Science, RWTH Aachen, Germany, 1999.

[Mac88] Robert M. MacGregor. A deductive pattern matcher. In *AAAI88, Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 403–408, Saint Paul, Minnesota, 1988.

[MB87] R. MacGregor and R. Bates. The LOOM knowledge representation language. In *Proceedings of KnowledgeBased Systems Workshop*, 1987.

[Min81] M. Minsky. A framework for representing knowledge. In *Mind Design*, pages 95–128. MIT press, 1981.

[MN83] J. Minker and J. M. Nicholas. On recursive axiom in deductive databases. *Information Systems*, 8(1):1–13, 1983.

[Neb90a] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, Heidelberg, New York, 1990.

[Neb90b] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2), 1990.

[Neb91] B. Nebel. Terminological cycles: Semantics and computational properties. In J. F. Sowa, editor, *Principles of Semantic Networks*, chapter 11, pages 331–362. Morgan Kaufmann Publishers, Inc., San Mateo, Cal. USA, 1991.

[NSDM03] Tommaso Di Noia, Eugenio Di Sciascio, Francesco M. Donini, and Marina Mongiello. A system for principled matchmaking in an electronic marketplace. In *Proceedings of the twelfth international conference on World Wide Web*, pages 321–330. ACM Press, 2003.

[PSS93] Peter F. Patel-Schneider and B. Swartout. Description-logic knowledge representation system specification. http://www-db.research.bell-labs.com/user/pfps/papers/krss-spec.ps, NOV 1993.

[Qui67] M.R. Quillian. Word concepts: A theory and simulation of some basic semantic capabilities. In *Behavioral Science 12*, pages 410–430, 1967.

[SBM01] D. Beneventano S. Bergamaschi, S. Castano and M.Vincini. Retrieving and integrating data from multiple sources: the momis approach. In *Special Issue on Intelligent Information Integration, Data & Knowledge Engineering*, volume 36, Num. 1, pages 215–249. Elsevier Science Publishers B.V. (North-Holland), Science B.V., March 2001.

[Sch91] Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In Ray Myopoulos, John; Reiter, editor, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 466–471, Sydney, Australia, August 1991. Morgan Kaufmann.

[SCM03] Ulrike Sattler, Diego Calvanese, and Ralf Molitor. Relationships with other formalisms. In *Description Logic Handbook*, pages 137–177, 2003.

[SSS91] Manfred Schmidt-Schauß and Gert Smolka. Attributive concept descriptions with unions and complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[Ull89] J. D Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Maryland - USA, 1989.

[VV.93] AA. VV. The common object request broker: Architecture and specification. Technical report, Object Request Broker Task Force, 1993. Revision 1.2, Draft 29, December.

[Wei92] Volker Weispfenning. Comprehensive gröbner bases. *J. Symb. Comput.*, 14(1):1–30, 1992.

[Wes03] Michael Wessel. Some practical issues in building a hybrid deductive geographic information system with a dl-component, 2003.

[YNM91] John Yen, Robert Neches, and Robert M. MacGregor. CLASP: Integrating term subsumption systems and production systems. *Knowledge and Data Engineering*, 3(1):25–32, 1991.