UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

# SI-Designer, un tool di ausilio all'integrazione di sorgenti di dati eterogenee distribuite: progetto e realizzazione

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Alberto Zanoli

Correlatore
Ing. Alberto Corni

Controrelatore
Prof. Flavio Bonfatti

Anno Accademico 1997 - 98

Parole chiave:
Intelligent Information Integration
Mediatore
Thesaurus
Parser
Acquisizione di schemi

# RINGRAZIAMENTI

*Ringrazio la Professoressa Sonia Bergamaschi per l'aiuto fornito alla realizzazione della presente tesi.*
*Un ringraziamento particolare va inoltre all'Ing. Alberto Corni e all'Ing. Maurizio Vincini per la preziosa collaborazione e la costante disponibilità.*

# Indice

# Elenco delle figure

# Introduzione

In questo periodo di fine millennio siamo testimoni di un rapido e continuo sviluppo della tecnologia delle comunicazioni, che ha determinato il moltiplicarsi del numero di fonti di informazione presenti in rete. Questa enorme disponibilità di dati è senza dubbio un fatto estremamente positivo, perché consente a chiunque il libero accesso ad informazioni sugli argomenti più disparati; tuttavia, paradossalmente, questa nuova situazione costringe l'utente a dover affrontare una nuova classe di problemi causati proprio dal fatto che l'enorme quantità di informazioni presenti in rete proviene da sorgenti autonome ed eterogenee: da un lato, infatti, l'utente si trova a dover selezionare le sorgenti potenzialmente interessanti, formulando ad ognuna l'interrogazione più appropriata; dall'altro deve essere in grado di filtrare, nella molteplicità delle risposte ottenute, i dati interessanti, scartando le probabili inconsistenze e le ridondanze. È vidente come, in questo contesto, sia fondamentale, da parte dell'utente, avere familiarità con i contenuti, le strutture e i linguaggi di interrogazione propri di queste risorse separate. Partendo dal presupposto che, proprio a causa dell'enorme sviluppo della rete Internet, la stessa è accessibile a sempre più persone, diventa sempre più difficile ipotizzare che l'utente medio possegga la conoscenza necessaria a portare a termine un compito di questo tipo che, peraltro, col passare del tempo, diventa sempre più complesso, giacché il numero di sorgenti è sempre crescente. Si avverte dunque l'esigenza di fornire una applicazione in grado di rendere trasparenti all'utente queste operazioni.

La possibilità di sviluppare applicazioni capaci di combinare ed utilizzare dati provenienti da una molteplicità di sorgenti è un tema di grande attualità, di interesse non solo teorico, ma anche applicativo, come dimostra la sempre maggiore presenza di sistemi commerciali, quali i *Datawarehouse*, i *Dataminer*, i *Sistemi di Workflow*, le *Federazioni di Database*, etc...
I domini di impiego per tali servizi sono innumerevoli. Si pensi, ad esempio, ad una realtà ospedaliera: l'importanza di poter consultare tutte le informazioni sanitarie relative ad una cartella clinica, quali ecografie, radiografie, esami del sangue, etc...
Il lavoro svolto in questa tesi fa parte di un progetto più ampio denominato **MOMIS**(**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources) nato a sua volta all'interno del progetto MURST 40% INTERDATA, come collaborazione tra l'unità operativa dell'Università di Modena e l'unità operativa dell'Università di Milano. Scopo del progetto **MOMIS** è quello di sviluppare un sistema *Mediatore* che integri diverse sorgenti di informazione eterogenee e distribuite, sia

strutturate che semistrutturate.
Avendo a disposizione un insieme di schemi locali, l'intenzione è quella di definire un'unica *vista* integrata che contenga l'insieme delle informazioni che le singole sorgenti intendono condividere. Dato il notevole livello di ambiguità che inevitabilmente caratterizza i problemi legati all'integrazione delle informazioni, il processo di definizione dello schema integrato non può essere completamente automatico, ma necessita della interazione con un progettista che, attraverso un linguaggio dichiarativo opportunamente definito ($ODL_{I^3}$) fornisce al sistema la conoscenza in suo possesso. L'uso di tecniche di Intelligenza Artificiale consente di alleggerire e velocizzare la fase di integrazione, al termine della quale sarà disponibile la vista globale. L'utente potrà interagire esclusivamente con la vista integrata; le interrogazioni, pertanto, saranno formulate ad un'unica fonte *virtuale*, e sarà cura del gestore delle interrogazioni (Query Manager) suddividerle in diverse query, ognuna indirizzata ad una sorgente locale diversa, e riunificare i risultati. Anche in questo caso l'uso di tecniche di Intelligenza Artificiale permette di ottimizzare i tempi di accesso alle risorse, pur mantenendo inalterato l'insieme risposta.

La struttura della tesi è la seguente:

Il Capitolo 1 introduce le problematiche che un sistema di Integrazione deve affrontare. È riportata una classificazione di questi problemi ed una architettura di riferimento che questo tipo di sistemi dovrebbero seguire, proposta di recente in ambito internazionale.

Nel Capitolo 2 viene descritto il sistema MOMIS, in particolare la sua architettura, conforme alla proposta internazionale introdotta nel capitolo1. Viene inoltre data una descrizione di alcuni strumenti software utilizzati, in particolare gli ODB-Tools e WordNet.

Il capitolo 3 descrive dettagliatamente, con l'aiuto di un esempio di riferimento, il processo semiautomatico adottato da MOMIS per pervenire allo schema integrato; viene inoltre dato accenno alla strategia utilizzata da MOMIS per risolvere le problematiche relative alla gestione delle interrogazioni.

Il Capitolo 4 introduce il linguaggio per la descrizione di schemi, denominato $ODL_{I^3}$, e il software, realizzato in questa tesi, che ne realizza il parser; in particolare la struttura dati progettata per contenere le metainformazioni espresse in $ODL_{I^3}$.

Nel Capitolo 5 viene data una descrizione di **SI-Designer**, un modulo software (realizzato in questa tesi) che assiste il progettista durante tutta la fase di progettazione dello schema integrato, e si occupa di coordinare l'esecuzione dei componenti software esterni, e di quelli realizzati in lavori precedenti.

# Capitolo 1

# L'Integrazione delle Informazioni

I primi aspetti teorici da considerare nell'integrazione delle informazioni sono legati alla natura dei dati (testi, immagini, suoni, record, ...) ed ai diversi tipi di sorgenti, che possono essere pagine HTML, DBMS relazionali o ad oggetti, file system, etc... . Gli standard esistenti (TCP/IP, ODBC, OLE, CORBA, SQL, etc...) solo parzialmente risolvono i problemi legati alle diversità Hw e Sw dei protocolli di rete e delle comunicazioni fra i moduli, mentre rimangono irrisolti quelli specifici della modellazione delle informazioni. I modelli dei dati e gli schemi si differenziano infatti gli uni dagli altri in modo da dare una struttura logica ai numerosi generi di dati da immagazzinare e questo crea una eterogeneità *semantica* (o *logica*) non risolvibile da questi standard. Inoltre l'abbondanza di informazioni dovuta ad un numero sempre maggiore di fonti, contribuisce a determinare problemi quali l'*information overload* (sovraccarico delle informazioni); altri problemi per i quali si devono trovare soluzioni sono la riduzione dei tempi di accesso, la salvaguardia della sicurezza e della consistenza, gli elevati costi di mantenimento da affrontare per modificare, eliminare od introdurre una nuova sorgente.

Tutti i problemi elencati sottolineano la complessità degli aspetti che le architetture dedicate all'integrazione devono comprendere. Per facilitare il processo di progettazione e realizzazione di tali moduli dedicati, che siano affidabili, flessibili e tali da far fronte efficacemente ad un panorama in continua evoluzione, si cerca di sviluppare un'architettura di moduli che assicuri il riuso e la capacità di interagire con altri sistemi esistenti.

Gli approcci all'integrazione, descritti in letteratura o effettivamente realizzati, presentano diverse metodologie: la reingegnerizzazione delle sorgenti mediante standardizzazione degli schemi e creazione di un database distribuito; il *repository independence*, un approccio che prevede di isolare al di sotto di una vista integrata le applicazioni ed i dati integrati dalle sorgenti, consentendo la massima autonomia e nascondendo al contempo le differenze esistenti; i *datawarehouse* che realizzano presso l'utente finale delle viste, ovvero delle porzioni di sorgenti,

replicando fisicamente i dati ed affidandosi a complicati algoritmi di 'allineamento' per assicurare la loro consistenza a fronte di modifiche nelle sorgenti originali. Nel seguito si descrive la proposta dell'ARPA (Advanced Research Projects Agency)[1] per un'architettura che favorisca l'autonomia delle sorgenti, assicurando flessibilità e riusabilità e si presenta l'approccio seguito per il progetto MOMIS.

## 1.1 L'Integrazione Intelligente delle Informazioni

L'integrazione delle Informazioni ($I^2$) si distingue da quella dei dati e dei database in quanto non cerca di collegare semplicemente alcune sorgenti, quanto piuttosto risultati opportunamente selezionati da esse [2]. Per ottenere risultati selezionati è richiesta *conoscenza* ed *intelligenza* finalizzate alla scelta delle sorgenti e dei dati, alla loro fusione e alla conseguente sintesi. L'integrazione comporta perciò grandi difficoltà, sia a livello teorico che pratico, oltre a concrete differenze realizzative. La dimensione e la quantità delle problematiche che insorgono qualora si desideri fondere informazioni provenienti da sorgenti autonome fanno sì che si senta l'esigenza di ideare una metodologia sia riusabile, per diminuire i costi di sviluppo di tali applicazioni, sia flessibile, per far fronte intelligentemente alle evoluzioni fisiche e logiche delle sorgenti interessate.

### 1.1.1 Il Programma $I^3$

Un'ambiziosa ricerca, finalizzata ad indicare un'architettura di riferimento che realizzi l'integrazione di sorgenti eterogenee in maniera automatica, è quella del programma $I^3$, sviluppato dall'ARPA, l'agenzia che fa capo al Dipartimento di Difesa americano [1]. L'integrazione aumenta il valore dell'informazione ma richiede una forte adattabilità realizzativa: si devono poter gestire i casi di aggiornamento e sostituzione delle sorgenti, dei loro ambienti e/o piattaforme, della loro ontologia e della semantica. Le tecniche sviluppate dall'*Intelligenza Artificiale*, potendo efficacemente dedurre informazioni utili dagli schemi delle sorgenti, diventano uno strumento prezioso per la costruzione automatica di soluzioni integrate flessibili e riusabili.

Costruire in modo premeditato supersistemi ad hoc che interessino una gran quantità di sorgenti non correlate semanticamente, è faticoso ed il risultato è un sistema scarsamente manutenibile o adattabile, strettamente finalizzato alla risoluzione dei problemi per cui è stato implementato. Secondo il programma $I^3$, una soluzione a questi problemi deriva dall'introduzione di architetture modulari sviluppabili secondo i principi proposti da uno standard il quale ponga le basi dei servizi che devono essere soddisfatti dall'integrazione ed abbassi i costi

di sviluppo e di mantenimento.
Costruire nuovi sistemi risulta realizzabile con minor difficoltà e minor tempo (e quindi costi) se si riesce a supportare lo sviluppo delle applicazioni riutilizzando la tecnologia già sviluppata. Per la riusabilità è fondamentale l'esistenza di interfacce ed architetture standard. Il paradigma suggerito per la suddivisione dei servizi e delle risorse nei diversi moduli, si articola su due dimensioni:

- *orizzontalmente* in tre livelli: livello utente, moduli intermedi che fanno uso di tecniche di IA, risorse di dati;

- *verticalmente*: diversi domini in cui raggruppare le sorgenti, il cui numero deve essere inferiore a dieci.

In generale i diversi domini non sono strettamente connessi all'interno di un certo livello ma si scambiano informazioni e dati; la combinazione delle informazioni avviene a livello dell'utilizzatore riducendo la complessità totale del sistema e consentendo lo sviluppo di numerose applicazioni finalizzate a scopi diversi fra loro. Ad esempio, si supponga di dover integrare informazioni sui trasporti mercantili, ferroviari e stradali: ciò permette all'utente di avere un'idea completa su quale mezzo di trasporto sia maggiormente vantaggioso ai propri fini. Aggiungendo a questo altri domini, quali le situazioni metereologiche ed i costi di immagazzinamento e trasporto, si possono facilitare le scelte di un dirigente che deve decidere come e quando consegnare delle merci.

Il livello dell'architettura su cui si deve focalizzare l'attenzione è quello intermedio: esso costituisce il punto nodale fra le applicazioni sviluppate per gli utenti ed i dati nelle sorgenti. Questo livello deve offrire servizi dinamici quali la selezione delle sorgenti, le gestione degli accessi e delle interrogazioni, il ricevimento e la combinazione dei dati, l'analisi e la sintesi degli stessi.

## 1.2 Architettura di riferimento per sistemi $I^3$

L'architettura di riferimento presentata in questo paragrafo è stata tratta dal sito web [1], e rappresenta una sommaria categorizzazione dei principi e dei servizi che possono e devono essere usati nella realizzazione di un integratore *intelligente* di informazioni derivanti da fonti eterogenee. Alla base del progetto $I^3$stanno infatti due ipotesi:

- la cosiddetta "autostrada delle informazioni" è oggigiorno incredibilmente vasta e, conseguentemente, sta per diventare una risorsa di informazioni utilizzabile poco efficientemente;

- le fonti di informazioni ed i sistemi informativi sono spesso semanticamente correlati tra di loro, ma non in una forma semplice né premeditata. Di conseguenza, il processo di integrazione di informazioni può risultare molto complesso.

In questo ambito, l'obiettivo del programma $I^3$ è di ridurre considerevolmente il tempo necessario per la realizzazione di un integratore di informazioni, raccogliendo e "strutturando" le soluzioni fino ad ora prevalenti nel campo della ricerca. Da sottolineare, prima di passare alla descrizione dell'architettura di riferimento, che questa architettura non implica alcuna soluzione implementativa, bensì vuole rappresentare alcuni dei servizi che deve includere un qualunque integratore di informazioni, e le interconnessioni tra questi servizi. Inoltre, è opportuno rimarcare che non sarà necessario, ed anzi è improbabile, che ciascun sistema che si prefigge di integrare informazioni (o servizi, o applicazioni) comprenda l'intero insieme di funzionalità che descriverò, bensì usufruirà esclusivamente delle funzionalità necessarie ad un determinato compito.

### 1.2.1 A cosa serve la tecnologia $I^3$ e quali problemi deve risolvere

Vi è un immenso spettro di applicazioni che si prestano naturalmente come campi applicativi per queste nuove tecnologie, tra le quali:

- pianificazione e supporto della logistica;
- sistemi informativi nel campo sanitario;
- sistemi informativi nel campo manifatturiero;
- sistemi bancari internazionali;
- ricerche di mercato.

Naturalmente, essendo questa riportata una architettura che pretende di essere il più generale possibile, ed essendo la casistica dei campi applicativi così vasta, sarà possibile identificare, al di là di un insieme di servizi di base, funzionalità più adatte ad una determinata applicazione e funzionalità specifiche di un altro ambiente. Ad esempio, un integratore che vuole interagire con sistemi di basi di dati "classici", come possono essere considerati i sistemi basati sui file, quelli relazionali, i DB ad oggetti, necessiterà di un pacchetto base di servizi molto differenti da un sistema cosiddetto "multimediale", che vuole integrare suoni, immagini ...
Così come possono essere differenti gli obiettivi di un sistema $I^3$, saranno

differenti pure i problemi che si troverà ad affrontare. Tra questi, possono essere identificati:

- *la grande differenza tra le fonti di informazione*:
  - le fonti informative sono semanticamente differenti, e si possono individuare dei livelli di differenze semantiche [3];
  - le informazioni possono essere memorizzate utilizzando differenti formati, come possono essere file, DB relazionali, DB ad oggetti;
  - possono essere diversi gli schemi, i vocabolari usati, le ontologie su cui questi si basano, anche quando le fonti condividono significative relazioni semantiche;
  - può variare inoltre la natura stessa delle informazioni, includendo testi, immagini, audio, media digitali;
  - infine, può variare il modo in cui si accede a queste sorgenti: interfacce utente, linguaggi di interrogazione, protocolli e meccanismi di transazione;
- *la semantica complessa ed a volte nascosta delle fonti*: molto spesso, la chiave per l'uso delle informazioni di vecchi sistemi sono i programmi applicativi su di essi sviluppati, senza i quali può essere molto difficile dedurre la semantica che si voleva esprimere, specialmente se si ha a che fare con sistemi molto vasti e quasi impossibili da interpretare se visti solo dall'esterno;
- *l'esigenza di creare applicazioni in grado di interfacciarsi con porzioni diverse delle fonti di informazione*: molto spesso, non è sempre possibile avere a disposizione l'intera sorgente di informazione, bensì una sua parte selezionata che può variare nel tempo;
- *il grande numero di fonti da integrare*: con il moltiplicarsi delle informazioni, il numero stesso delle fonti da integrare per una applicazione, ad esempio nel campo sanitario, è aumentato considerevolmente, e decine di fonti devono essere accedute in modo coordinato;
- *il bisogno di realizzare moduli $I^3$ riusabili*: benché questo possa essere considerato uno dei compiti più difficili nella realizzazione di un integratore, è importante realizzare non un sistema ad-hoc, bensì un'applicazione i cui moduli possano facilmente essere riutilizzati in altre applicazioni, secondo i moderni principi di riusabilità del software. In questo caso, l'abilità di

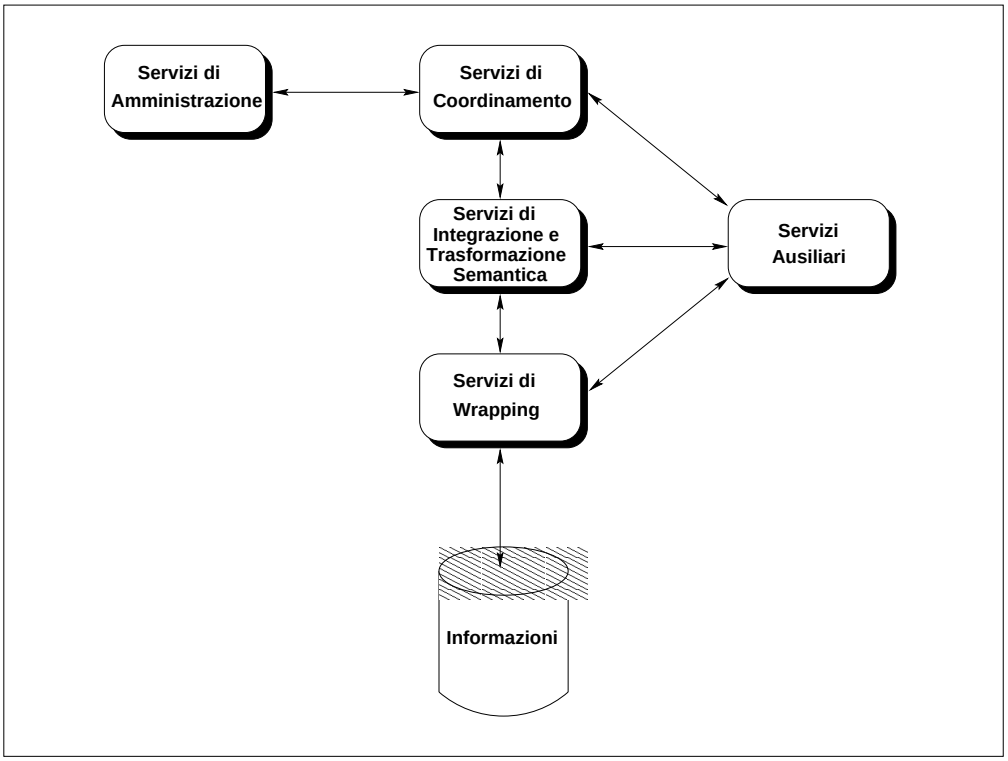



Figura 1.1: Diagramma dei servizi $I^3$

costruire valide funzioni di libreria può considerevolmente diminuire i tempi e le difficoltà di realizzazione di un sistema informativo che si basa su più fonti differenti.

Passiamo ora ad analizzare l'architettura vera e propria di un sistema $I^3$, riportata in Figura 1.1. L'architettura di riferimento dà grande rilevanza ai Servizi di Coordinamento. Questi servizi giocano infatti due ruoli: come prima cosa, possono localizzare altri servizi $I^3$ e fonti di informazioni che possono essere utilizzati per costruire il sistema stesso; secondariamente, sono responsabili di individuare ed invocare a run-time gli altri servizi necessari a dare risposta ad una specifica richiesta di dati.
Sono comunque in totale cinque le famiglie di servizi che possono essere identificati in questa architettura: importanti sono i due assi della figura, orizzontale e verticale, che sottolineano i differenti compiti dei servizi $I^3$.
Se percorriamo l'asse verticale, si può intuire come avviene lo scambio di informazioni nel sistema: in particolare, i servizi di *wrapping* provvedono ad estrarre le informazioni dalle singole sorgenti, che sono poi impacchettate ed integrate dai Servizi di Integrazione e Trasformazione Semantica, per poi essere passati

ai servizi di Coordinamento che ne avevano fatto richiesta. L'asse orizzontale mette invece in risalto il rapporto tra i servizi di Coordinamento e quelli di Amministrazione, ai quali spetta infatti il compito di mantenere informazioni sulle capacità delle varie sorgenti (che tipo di dati possono fornire ed in quale modo devono essere interrogate). Funzionalità di supporto, che verranno descritte successivamente, sono invece fornite dai Servizi Ausiliari, responsabili dei servizi di arricchimento semantico delle sorgenti.
Analizziamone in dettaglio funzionalità e problematiche affrontate.

### 1.2.2 Servizi di Coordinamento

I servizi di Coordinamento sono quei servizi di alto livello che permettono l'individuazione delle sorgenti di dati *interessanti*, ovvero che probabilmente possono dare risposta ad una determinata richiesta dell'utente. A seconda delle possibilità dell'integratore che si vuole realizzare, vanno dalla selezione dinamica delle sorgenti (o brokering, per Integratori Intelligenti) al semplice *Matchmaking*, in cui il mappaggio tra informazioni integrate e locali è realizzato manualmente ed una volta per tutte. Vediamo alcuni esempi.

1. **Facilitation e Brokering Services**: l'utente manda una richiesta al sistema e questo usa un deposito di metadati per ritrovare il modulo che può trattare la richiesta direttamente. I moduli interessati da questa richiesta potranno essere uno solo alla volta (nel qual caso si parla di Brokering) o più di uno (e in questo secondo caso si tratta di facilitatori e mediatori, attraverso i quali a partire da una richiesta ne viene generata più di una da inviare singolarmente a differenti moduli che gestiscono sorgenti distinte, e reintegrando poi le risposte in modo da presentarle all'utente come se fossero state ricavate da un'unica fonte). In questo ultimo caso, in cui una query può essere decomposta in un insieme di sottoquery, si farà uso di servizi di Query Decomposition e di tecniche di Inferenza (mutuate dall'Intelligenza Artificiale) per una determinazione dinamica delle sorgenti da interrogare, a seconda delle condizioni poste nell'interrogazione.
   I vantaggi che questi servizi di Coordinamento portano stanno nel fatto che non è richiesta all'utente del sistema una conoscenza del contenuto delle diverse sorgenti, dandogli l'illusione di interagire con un sistema omogeneo che gestisce direttamente la sua richiesta. E' quindi esonerato dal conoscere i domini con i quali i vari moduli $I^3$ hanno a che fare, ottenendone una considerabile diminuzione di complessità di interazione col sistema.

2. **Matchmaking**: il sistema è configurato manualmente da un operatore all'inizio, e da questo punto in poi tutte le richieste saranno trattate allo stesso

modo. Sono definiti gli anelli di collegamento tra tutti i moduli del sistema, e nessuna ottimizzazione è fatta a tempo di esecuzione.

### 1.2.3 Servizi di Amministrazione

Sono servizi usati dai Servizi di Coordinamento per localizzare le sorgenti *utili*, per determinare le loro capacità, e per creare ed interpretare TEMPLATE. I Template sono strutture dati che descrivono i servizi, le fonti ed i moduli da utilizzare per portare a termine un determinato task. Sono quindi utilizzati dai sistemi meno "intelligenti", e consentono all'operatore di predefinire le azioni da eseguire a seguito di una determinata richiesta, limitando al minimo le possibilità di decisione del sistema.
In alternativa a questi metodi dei Template, sono utilizzate le **Yellow Pages**: servizi di directory che mantengono le informazioni sul contenuto delle varie sorgenti e sul loro stato (attiva, inattiva, occupata). Consultando queste Yellow Pages, il mediatore sarà in grado di spedire alla giusta sorgente la richiesta di informazioni, ed eventualmente di rimpiazzare questa sorgente con una equivalente nel caso non fosse disponibile. Fanno parte di questa categoria di servizi il Browsing: permette all'utente di "navigare" attraverso le descrizioni degli schemi delle sorgenti, recuperando informazioni su queste. Il servizio si basa sulla premessa che queste descrizioni siano fornite esplicitamente tramite un linguaggio dichiarativo leggibile e comprensibile dall'utente. Potrebbe fornirsi a sua volta dei servizi Trasformazione del Vocabolario e dell'Ontologia, come pure di Integrazione Semantica. Da citare sono pure i servizi di Iterative Query Formulation: aiutano l'utente a rilassare o meglio specificare alcuni vincoli della propria interrogazione per ottenere risposte più precise.

### 1.2.4 Servizi di Integrazione e Trasformazione Semantica

Questi servizi supportano le manipolazioni semantiche necessarie per l'integrazione e la trasformazione delle informazioni. Il tipico input per questi servizi saranno una o più sorgenti di dati, e l'output sarà la "vista" integrata o trasformata di queste informazioni. Tra questi servizi si distinguono quelli relativi alla trasformazione degli schemi (ovvero di tutto ciò che va sotto il nome di *metadati*) e quelli relativi alla trasformazione dei dati stessi. Sono spesso indicati come servizi di Mediazione, essendo tipici dei moduli mediatori.

1. Servizi di **integrazione degli schemi**. Supportano la trasformazione e l'integrazione degli schemi e delle conoscenze derivanti da fonti di dati eterogenee. Fanno parte di essi i servizi di trasformazione dei vocaboli e dell'ontologia, usati per arrivare alla definizione di un'ontologia unica che combini

gli aspetti comuni alle singole ontologie usate nelle diverse fonti. Queste operazioni sono molto utili quando devono essere scambiate informazioni derivanti da ambienti differenti, dove molto probabilmente non si condivideva un'unica ontologia. Fondamentale, per creare questo insieme di vocaboli condivisi, è la fase di individuazione dei concetti presenti in diverse fonti, e la riconciliazione delle diversità presenti sia nelle strutture, sia nei significati dei dati.

2. Servizi di **integrazione delle informazioni**. Provvedono alla traduzione dei termini da un contesto all'altro, ovvero dall'ontologia di partenza a quella di destinazione. Possono inoltre occuparsi di uniformare la "granularità" dei dati (come possono essere le discrepanze nelle unità di misura, o le discrepanze temporali).

3. Servizi di **supporto al processo di integrazione**. Sono utilizzati nel momento in cui una query è scomposta in molte subquery, da inviare a fonti differenti, ed i loro risultati devono essere integrati. Comprendono inoltre tecniche di *caching*, per supportare la materializzazione delle viste (problematica molto comune nei sistemi che vanno sotto il nome di datawarehouse).

### 1.2.5 Servizi di Wrapping

Sono utilizzati per fare sì che le fonti di informazioni aderiscano ad uno standard, che può essere interno o proveniente dal mondo esterno con cui il sistema vuole interfacciarsi. Si comportano come traduttori dai sistemi locali ai servizi di alto livello dell'integratore. In particolare, sono due gli obiettivi che si prefiggono:

1. permettere ai servizi di coordinamento e di mediazione di manipolare in modo uniforme il numero maggiore di sorgenti locali, anche se queste non erano state esplicitamente pensate come facenti parte del sistema di integrazione;

2. essere il più riusabili possibile. Per fare ciò, dovrebbero fornire interfacce che seguano gli standard più diffusi ( e tra questi, si potrebbe citare il linguaggio SQL come linguaggio di interrogazione di basi di dati, e CORBA come protocollo di scambio di oggetti). Questo permetterebbe alle sorgenti estratte da questi wrapper "universali" di essere accedute dal numero maggiore possibile di moduli mediatori.

In pratica, compito di un wrapper è modificare l'interfaccia, i dati ed il comportamento di una sorgente, per facilitarne la comunicazione con il mondo esterno.

Il vero obiettivo è quindi standardizzare il processo di wrapping delle sorgenti, permettendo la creazione di una libreria di fonti accessibili; inoltre, il processo stesso di realizzazione di un wrapper dovrebbe essere standardizzato, in modo da poter essere riutilizzato per altre fonti.

### 1.2.6 Servizi Ausiliari

Aumentano le funzionalità degli altri servizi descritti precedentemente: sono prevalentemente utilizzati dai moduli che agiscono direttamente sulle informazioni. Vanno dai semplici servizi di monitoraggio del sistema (un utente vuole avere un segnale nel momento in cui avviene un determinato evento in un database, e conseguenti azioni devono essere attuate), ai servizi di propagazione degli aggiornamenti e di ottimizzazione. .

## 1.3 Il mediatore

L'obiettivo del progetto di ricerca di cui la tesi fa parte è la progettazione e realizzazione di un **mediatore**, ovvero del modulo intermedio dell'architettura precedentemente descritta, che si pone tra l'utente e le sorgenti di informazioni. Secondo la definizione proposta da Wiederhold in [4] "un mediatore è un modulo software che sfrutta la conoscenza su un certo insieme di dati per creare informazioni per una applicazione di livello superiore . . . Dovrebbe essere piccolo e semplice, così da poter essere amministrato da uno, o al più pochi, esperti."
Compiti di un mediatore sono allora:

- assicurare un servizio stabile, anche quando cambiano le risorse;
- amministrare e risolvere le eterogeneità delle diverse fonti;
- integrare le informazioni ricavate da più risorse;
- presentare all'utente le informazioni attraverso un modello scelto dall'utente stesso.

La prima ipotesi che è stata fatta, per restringere il campo applicativo del sistema da progettare (e di conseguenza per restringere il campo dei problemi a cui dare risposta) è di avere a che fare, per il momento, esclusivamente con sorgenti di dati di testo strutturati e semistrutturati, quali possono essere basi di dati relazionali, ad oggetti, file di testo, e pagine html. L'approccio architetturale scelto è quello *classico*, che consta principalmente di 3 livelli:

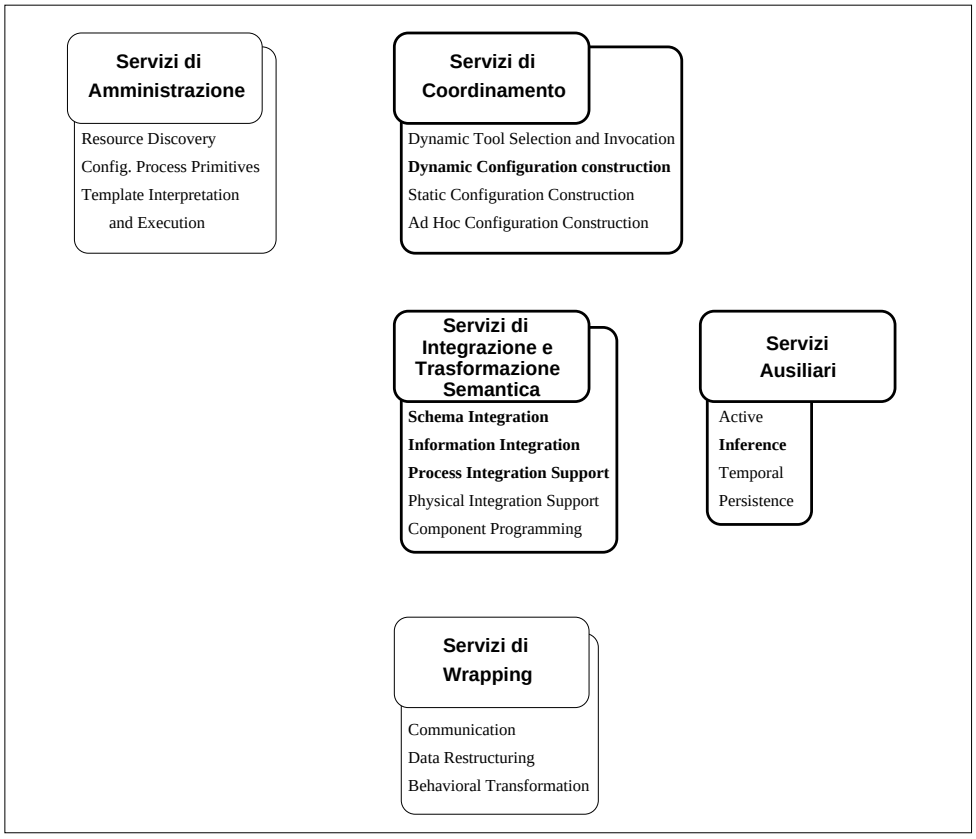



Figura 1.2: Servizi $I^3$ presenti nel mediatore

1. utente: attraverso un'interfaccia grafica l'utente pone delle query su uno schema globale e riceve un'unica risposta, come se stesse interrogando un'unica sorgente di informazioni;

2. mediatore: il mediatore gestisce l'interrogazione dell'utente, combinando, integrando ed eventualmente arricchendo i dati ricevuti dai wrapper, ma usando un modello (e quindi un linguaggio di interrogazione) comune a tutte le fonti;

3. wrapper: ogni wrapper gestisce una singola sorgente, ed ha una duplice funzione: da un lato converte le richieste del mediatore in una forma comprensibile dalla sorgente, dall'altro traduce informazioni estratte dalla sorgente nel modello usato dal mediatore.

Facendo riferimento ai servizi descritti nelle sezioni precedenti, l'architettura del mediatore che si è progettato è riportata in Figura 1.2. In particolare, in questa tesi, sono state esaminate le seguenti funzionalità:

- servizi di Coordinamento: sul modello di facilitatori e mediatori, il sistema sarà in grado, in presenza di una interrogazione, di individuare automaticamente tutte le sorgenti che ne saranno interessate, ed eventualmente di scomporre la richiesta in un insieme di sottointerrogazioni diverse da inviare alle differenti fonti di informazione;

- servizi di Integrazione e Trasformazione Semantica: saranno forniti dal mediatore servizi che facilitino l'integrazione sia degli schemi che delle informazioni, nonchè funzionalità di supporto al processo di integrazione (come può essere la Query Decomposition);

- servizi Ausiliari: sono utilizzate tecniche di Inferenza per realizzare, all'interno del mediatore, una fase di ottimizzazione delle interrogazioni.

Parallelamente a questa impostazione architetturale inoltre, il nostro progetto si vuole distaccare dall'approccio *strutturale*, cioè sintattico, tuttora dominante tra i sistemi presenti sul mercato. L'approccio strutturale, adottato da sistemi quali TSIMMIS [5, 6, 7, 8], è caratterizzato dal fatto di usare un self-describing model per rappresentare gli oggetti da integrare, limitando l'uso delle informazioni semantiche alle regole predefinite dall'operatore. In pratica, il sistema non conosce a priori la semantica di un oggetto che va a recuperare da una sorgente (e dunque di questa non possiede alcuno schema descrittivo) bensì è l'oggetto stesso che, attraverso delle etichette, si autodescrive, specificando tutte le volte, per ogni suo singolo campo, il significato che ad esso è associato. Questo approccio porta quindi ad un insieme di vantaggi, tra i quali possiamo identificare:

- la possibilità di integrare in modo completamente trasparente al mediatore basi di dati fortemente eterogenee e magari mutevoli nel tempo: il mediatore non si basa infatti su una descrizione predefinita degli schemi delle sorgenti, bensì sulla descrizione che ogni singolo oggetto fa di sè. Oggetti simili provenienti dalla stessa sorgente possono quindi avere strutture differenti, cosa invece non ammessa in un ambiente tradizionale object-oriented;

- per trattare in modo omogeneo dati che descrivono lo stesso concetto, o che hanno concetti in comune, ci si basa sulla definizione manuale di rule, che permettono di identificare i termini (e dunque i concetti) che devono essere condivisi da più oggetti.

Altri progetti, e tra questi il nostro, seguono invece un approccio definito *semantico*, che è caratterizzato dai seguenti punti:

- il mediatore deve conoscere, per ogni sorgente, lo schema concettuale (metadati);
- informazioni semantiche sono codificate in questi schemi;
- deve essere disponibile un modello comune per descrivere le informazioni da condividere (e dunque per descrivere anche i metadati);
- deve essere possibile una integrazione (parziale o totale) delle sorgenti di dati.

In questo modo, sfruttando opportunamente le informazioni semantiche che necessariamente ogni schema sottintende, il mediatore può individuare concetti comuni a più sorgenti e relazioni che li legano.

### 1.3.1 Problematiche da affrontare

Pur avendo a disposizione gli schemi concettuali delle varie sorgenti, non è certamente un compito banale individuare i concetti comuni ad esse, le relazioni che possono legarli, né tantomeno è banale realizzare una loro coerente integrazione. Mettendo da parte per un attimo le differenze dei sistemi fisici (alle quali dovrebbero pensare i moduli wrapper) i problemi che si è dovuto risolvere, o con i quali occorre giungere a compromessi, sono (a livello di mediazione, ovvero di integrazione delle informazioni) essenzialmente di due tipi:

1. problemi ontologici;
2. problemi semantici.

### 1.3.2 Problemi ontologici

Come riportato nel glossario A, per ontologia si intende, in questo ambito, ”l’insieme dei termini e delle relazioni usate in un dominio, che denotano concetti ed oggetti”. Con ontologia quindi ci si riferisce a quell’insieme di termini che, in un particolare dominio applicativo, denotano in modo univoco una particolare conoscenza e fra i quali non esiste ambiguità poiché sono condivisi dall’intera comunità di utenti del dominio applicativo stesso. Non è certamente l’obiettivo né di questo paragrafo, né della tesi in generale, dare una descrizione esaustiva di cosa si intenda per ontologia e dei problemi che essa comporta (ancorché ristretti al campo dell’integrazione delle informazioni), ma mi limito a riportare una semplice classificazione delle ontologie (mutuata da

Guarino [9, 10], per inquadrare l'ambiente in cui ci si muove. I livelli di ontologia (e dunque le problematiche ad essi associate) sono essenzialmente quattro:

1. *top-level ontology*: descrivono concetti molto generali come spazio, tempo, evento, azione . . . , che sono quindi indipendenti da un particolare problema o dominio: si considera ragionevole, almeno in teoria, che anche comunità separate di utenti condividano la stessa top-level ontology;

2. *domain e task ontology*: descrivono, rispettivamente, il vocabolario relativo a un generico dominio (come può essere un dominio medico, o automobilistico) o a un generico obiettivo (come la diagnostica, o le vendite), dando una specializzazione dei termini introdotti nelle top-level ontology;

3. *application ontology*: descrivono concetti che dipendono sia da un particolare dominio che da un particolare obiettivo.

Come ipotesi semplificativa di questo progetto, si è considerato di muoversi all'interno delle domain ontology, ipotizzando quindi che tutte le fonti informative condividano almeno i concetti fondamentali (ed i termini con cui identificarli).

### 1.3.3 Problemi semantici

Pur ipotizzando che anche sorgenti diverse condividano una visione similare del problema da modellare, e quindi un insieme di concetti comuni, niente ci dice che i diversi sistemi usino esattamente gli stessi vocaboli per rappresentare questi concetti, né tantomeno le stesse strutture dati. Poiché infatti le diverse sorgenti sono state progettate e modellate da persone differenti, è molto improbabile che queste persone condividano la stessa "concettualizzazione" del mondo esterno, ovvero non esiste nella realtà una semantica univoca a cui chiunque possa riferirsi.
Se la persona P1 disegna una fonte di informazioni (per esempio DB1) e un'altra persona P2 disegna la stessa fonte DB2, le due basi di dati avranno sicuramente differenze semantiche: per esempio, le coppie sposate possono essere rappresentate in DB1 usando degli oggetti della classe COPPIE, con attributi MARITO e MOGLIE, mentre in DB2 potrebbe esserci una classe PERSONA con un attributo SPOSA.
Come riportato in [3] la causa principale delle differenze semantiche si può identificare nelle diverse concettualizzazioni del mondo esterno che persone distinte possono avere, ma non è l'unica. Le differenze nei sistemi di DBMS possono portare all'uso di differenti modelli per la rappresentazione della porzione di mondo in questione: partendo così dalla stessa concettualizzazione, determinate

relazioni tra concetti avranno strutture diverse a seconda che siano realizzate attraverso un modello relazionale, o ad oggetti.
L'obiettivo dell'integratore, che è fornire un accesso integrato ad un insieme di sorgenti, si traduce allora nel non facile compito di identificare i concetti comuni all'interno di queste sorgenti e risolvere le differenze semantiche che possono essere presenti tra di loro. Possiamo classificare queste contraddizioni semantiche in tre gruppi principali:

1. **eterogeneità tra le classi di oggetti**: benché due classi in due differenti sorgenti rappresentino lo stesso concetto nello stesso contesto, possono usare nomi diversi per gli stessi attributi, per i metodi, oppure avere gli stessi attributi con domini di valori diversi o ancora (dove questo è permesso) avere regole differenti su questi valori;

2. **eterogeneità tra le strutture delle classi**: comprendono le differenze nei criteri di specializzazione, nelle strutture per realizzare una aggregazione, ed anche le *discrepanze schematiche*, quando cioè valori di attributi sono invece parte dei metadati in un altro schema (come può essere l'attributo SESSO in uno schema, presente invece nell'altro implicitamente attraverso la divisione della classe PERSONE in MASCHI e FEMMINE);

3. **eterogeneità nelle istanze delle classi**: ad esempio, l'uso di diverse unità di misura per i domini di un attributo, o la presenza/assenza di valori nulli.

Parallelamente a tutto questo, è però il caso di sottolineare la possibilità di sfruttare adeguatamente queste differenze semantiche per arricchire il nostro sistema: analizzando a fondo queste differenze, e le loro motivazioni, si può arrivare al cosiddetto **arricchimento semantico**, ovvero all'aggiungere esplicitamente ai dati tutte quelle informazioni che erano originariamente presenti solo come metadati negli schemi, dunque in un formato non interrogabile.

# Capitolo 2

# Il sistema MOMIS

Tenendo presente tutte le problematiche relative al mediatore esposte nella sezione 1.3, e un insieme di progetti pre-esistenti quali TSIMMIS [5, 6, 7, 8], GARLIC [11, 12] e SIMS [13, 14], si è giunti alla progettazione di un sistema intelligente di Integrazione delle Informazioni.

## 2.1 MOMIS

MOMIS(**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources è il progetto di un sistema $I^3$, per l'integrazione di sorgenti di dati strutturati e semistrutturati. **Momis** nasce all'interno del progetto MURST 40% INTERDATA, come collaborazione fra le unità operative dell'Università di Milano e dell'Università di Modena.

In questa tesi si approfondirà il componente *mediatore* che, come descritto al capitolo 1, nell'architettura a tre livelli si pone nello strato intermedio, tra l'utente e le sorgenti di informazione. In particolare, scopo di questa tesi è il progetto e la realizzazione di due componenti software del mediatore MOMIS: **SI-Designer**, descritto al capitolo 5 e il parser $\text{ODL}_{I^3}$, descritto al capitolo 4.

## 2.2 L'architettura di MOMIS

MOMIS è stato progettato per fornire un accesso integrato ad informazioni eterogenee memorizzate sia in database di tipo tradizionale (e.g. relazionali, object-oriented) o file system, sia in sorgenti di tipo semistrutturato.

Seguendo l'architettura di riferimento $I^3$ [1], in MOMIS si possono distinguere cinque componenti principali (come si può vedere da Fig. 2.1):

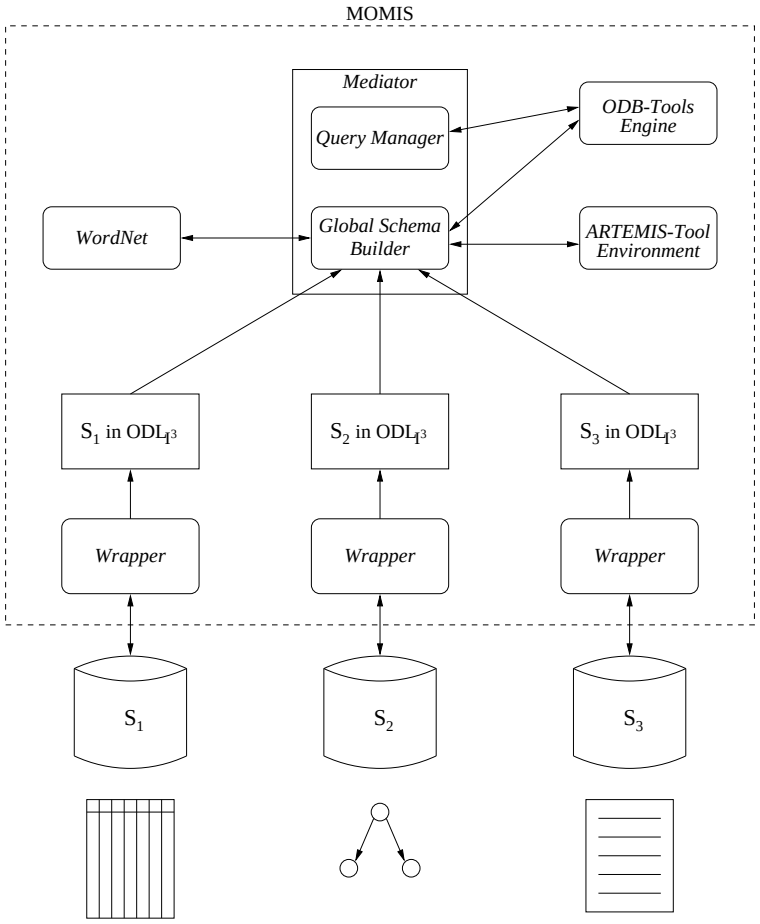



Figura 2.1: Architettura del sistema MOMIS

1. *Wrapper*: posti al di sopra di ciascuna sorgente, sono i moduli che rappresentano l'interfaccia tra il mediatore vero e proprio e le sorgenti locali di dati. La loro funzione è duplice:

   - in fase di integrazione, forniscono la descrizione delle informazioni in essa contenute. Questa descrizione viene fornita attraverso il linguaggio $\text{ODL}_{I^3}$ (descritto in Sezione 4.2);
   - in fase di query processing, traducono la query ricevuta dal mediatore (espressa quindi nel linguaggio comune di interrogazione $\text{OQL}_{I^3}$, definito in analogia al linguaggio OQL) in una interrogazione comprensibile (e realizzabile) dalla sorgente stessa. Devono inoltre esportare i dati ricevuti in risposta all'interrogazione, presentandoli al mediatore attraverso il modello comune di dati utilizzato dal sistema;

2. *mediatore*: è il cuore del sistema, ed è composto da due moduli distinti.

   - Global Schema Builder (GSB): è il modulo di integrazione degli schemi locali che, partendo dalle descrizioni delle sorgenti espresse attraverso il linguaggio $ODL_{I^3}$, genera un unico schema globale da presentare all'utente;
   - Query Manager (QM): è il modulo di gestione delle interrogazioni. In particolare, genera le query in linguaggio $OQL_{I^3}$ da inviare ai wrapper partendo dalla singola query formulata dall'utente sullo schema globale. Servendosi delle tecniche di Logica Descrittiva, il QM genera automaticamente la traduzione della query sottomessa nelle corrispondenti sub-query delle singole sorgenti.

3. *ODB-Tools Engine*, un tool basato sulla **OLCD** Description Logics [15, 16] che compie la validazione di schemi e l'ottimizzazione di query [17, 18, 19].

4. *ARTEMIS-Tool Environment*, un tool che compie analisi e clustering di schemi [20, 21].

5. *WordNet*, un database lessicale della lingua inglese, capace di individuare relazioni lessicali e semantiche fra termini [22].

Il principale scopo che ci si è preposti con MOMIS è la realizzazione di un sistema di mediazione che, a differenza di molti altri progetti analizzati, contribuisca a realizzare, oltre alla fase di query processing, una reale integrazione delle sorgenti. Entrambe queste fasi sono descritte nei prossimi paragrafi e, più approfonditamente, nel capitolo 3.

### 2.2.1 Il processo di Integrazione

L'integrazione delle sorgenti informative strutturate e semistrutturate viene compiuta in modo semi-automatico, utilizzando le descrizioni degli schemi locali in linguaggio $ODL_{I^3}$ e combinando le tecniche di Description Logics e di clustering. Come mostrato in figura 2.2, le attività compiute sono le seguenti:

1. *Generazione del Thesaurus Comune*, grazie al supporto di ODB-Tools e di WordNet. Durante questo passo viene costruito un Thesaurus Comune di *relazioni terminologiche*. Le relazioni terminologiche esprimono la conoscenza inter-schema su sorgenti diverse e corrispondono alle asserzioni intensionali utilizzate in [23]. Le relazioni terminologiche sono derivate in modo semi-automatico a partire dalle descrizioni degli schemi in $ODL_{I^3}$, attraverso l'analisi strutturale (utilizzando ODB-Tools e le tecniche di Description

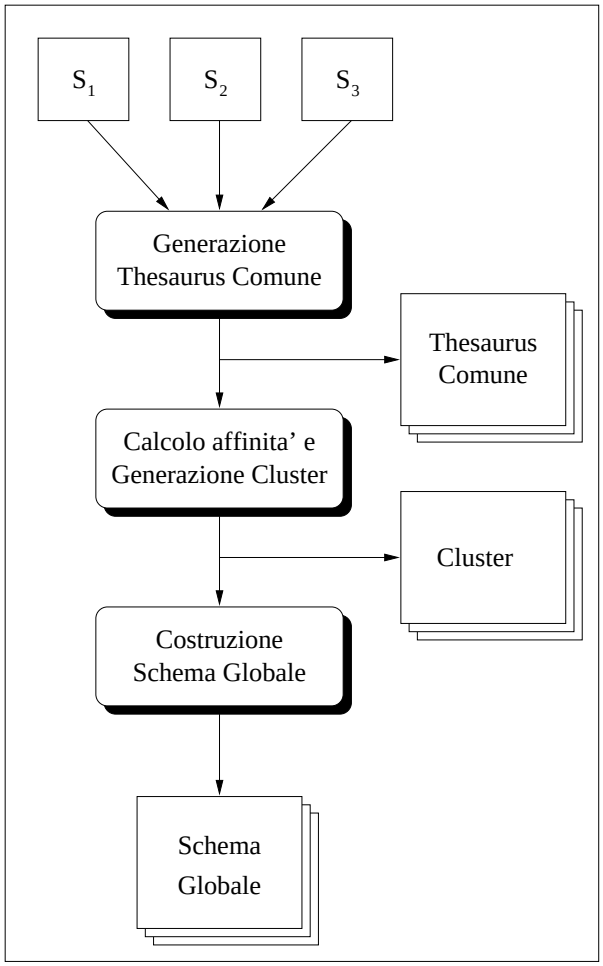



Figura 2.2: Le fasi dell'integrazione

Logics) e di contesto (attraverso l'uso di WordNet) delle classi coinvolte. Questa fase sarà discussa in sezione 3.1.

2. *Generazione dei cluster di classi ODL$_{I^3}$*, con il supporto dell'ambiente ARTEMIS-Tool, le relazioni terminologiche contenute nel Thesaurus vengono utilizzate per valutare il livello di affinità tra le classi ODL$_{I^3}$ in modo da identificare le informazioni che devono essere integrate a livello globale. A tal fine, ARTEMIS calcola i coefficienti che misurano il livello di affinità delle classi ODL$_{I^3}$ basandosi sia sui nomi delle stesse, sia sugli attributi. Le classi ODL$_{I^3}$ con maggiore affinità vengono raggruppate utilizzando le tecniche di clustering [24]. Questa fase sarà discussa nelle sezioni 3.2 e 3.3.

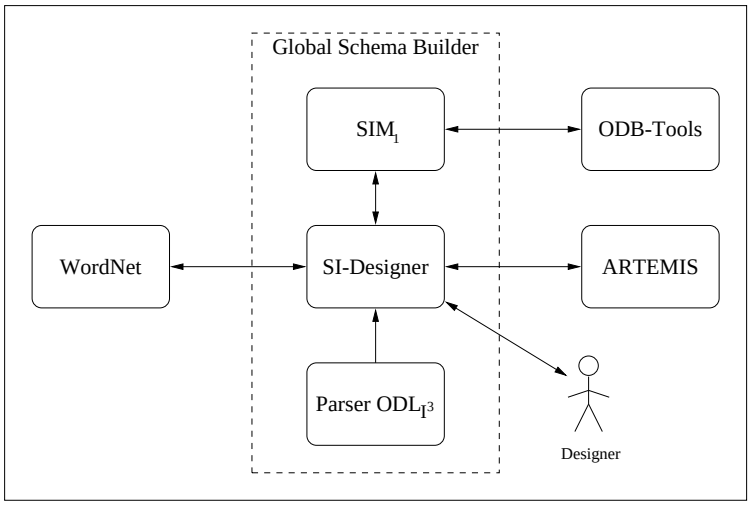



Figura 2.3: Architettura del Global Schema Builder

3. *Costruzione dello schema globale del mediatore*, i cluster di classi $ODL_{I^3}$ affini sono analizzati per costruire lo schema globale del Mediatore. Per ciascun cluster viene definita una classe globale $ODL_{I^3}$ che rappresenta tutte le classi locali che sono riferite al cluster, e che è caratterizzata dall'unione dei loro attributi. L'insieme delle classi globali definite costituisce lo schema globale del Mediatore che deve essere usato per porre le query alle sorgenti locali integrate. Questa fase sarà discussa nelle sezioni 3.4 e 3.5.

### Il Global Schema Builder

Il Global Schema Builder è la parte del Mediatore di Momis che si preoccupa della costruzione dello schema integrato, secondo le fasi appena elencate. Come si nota dalla figura 2.3, è composto da tre componenti che interagiscono con strumenti software esterni, descritti a loro volta nella prossima sezione.

I componenti il GSB sono:

1. *$SIM_1$*: descritto in [25], si occupa della generazione del Thesaurus comune, in particolare della estrazione delle relazioni dalla struttura degli schemi sorgenti e, con l'aiuto di ODB-Tools, della validazione delle relazioni integrate dal progettista nonché dell'inferenza di nuove relazioni.

2. *SI-Designer*: ha il duplice scopo di interfacciarsi fra il sistema e il progettista (fornendo a quest'ultimo un'interfaccia amichevole per l'interazione)

e di coordinare l'esecuzione dei diversi software che partecipano all'integrazione; in particolare:

- interagisce con WordNet per estrarre automaticamente relazioni lessicali tra i termini usati per dare nome a classi ed attributi;
- utilizza Artemis per svolgere il calcolo delle affinità;
- assiste il progettista nelle ultime fasi della costruzione dello schema globale, generando in linguaggio $ODL_{I^3}$ il codice che lo descrive.

Il componente SI-Designer, realizzato in questa tesi, è descritto in dettaglio nel capitolo 5.

3. *Parser* $ODL_{I^3}$: effettua l'analisi degli schemi sorgenti, verificandone la consistenza, come meglio spiegato nel capitolo 4. Analizza e archivia, inoltre, tutte le informazioni sullo schema integrato. Anche la realizzazione del parser fa parte del lavoro di questa tesi.

### 2.2.2 Query processing e ottimizzazione

Quando l'utente pone una query sullo schema globale, MOMIS la analizza e produce un insieme di sub-query che saranno inviate a ciascuna sorgente informativa coinvolta. In accordo con altri approcci proposti in quest'ambito [13, 14], il processo consiste di due attività principali:

1. *Ottimizzazione Semantica*. Il Mediatore agisce sulla query sfruttando la tecnica di ottimizzazione semantica supportata da ODB-Tools in modo da ridurre il costo del piano d'accesso. L'ottimizzazione è basata sull'inferenza logica a partire dalla conoscenza contenuta nei vincoli di integrità dello schema globale. La stessa procedura di ottimizzazione semantica si realizza in termini locali su ogni query tradotta dal Mediatore nella formulazione del piano d'accesso: in questo caso ci si basa sui vincoli di integrità presenti sui singoli schemi locali.

2. *Formulazione del piano di accesso.* Dopo aver ottenuto la query ottimizzata, viene generato l'insieme di sub-query relative alle sorgenti coinvolte. A questo scopo, il Mediatore utilizza l'associazione tra classi globali e locali per esprimere la query globale in termini degli schemi locali, nonché l'eventuale conoscenza di regole interschema definite sulle estensioni delle classi locali.

L'Ottimizzazione Semantica e il query plan saranno descritti nel paragrafo 3.6.1.

## 2.3 Gli strumenti utilizzati

### 2.3.1 ODB-Tools

ODB-Tools è un sistema per l'acquisizione e la verifica di consistenza di schemi di basi di dati e per l'ottimizzazione semantica di interrogazioni nelle basi di dati orientate agli oggetti (OODB). È stato sviluppato presso il Dipartimento di Scienze dell'Ingegneria dell'Università di Modena [17, 26]. L'ambiente teorico su cui è basato ODB-Tools include due elementi fondamentali:

1. **OLCD**(**O**bject **L**anguage with **C**omplements allowing **D**escriptive cycles), derivante dalla famiglia KL-ONE [27], proposto come formalismo comune per esprimere descrizioni di classi, vincoli di integrità, ed interrogazioni e dotato di tecniche di inferenza basate sul calcolo della sussunzione introdotte per le *Logiche Descrittive* nell'ambito dell'Intelligenza Artificiale;

2. **espansione semantica** di un tipo, realizzata attraverso l'algoritmo di sussunzione.

#### Aspetti generali

Il formalismo **OLCD** deriva dal precedente **ODL**, proposto in [28, 29], che già estendeva l'espressività di linguaggi di logica descrittiva al fine di rappresentare la semantica dei modelli di dati ad oggetti complessi (*CODMs*), recentemente proposti in ambito di basi di dati deduttive e basi di dati orientate agli oggetti. La caratteristica principale di ODL consiste nell'assumere una ricca struttura per il sistema dei tipi di base: oltre ai classici tipi atomici *integer*, *boolean*, *string*, *real*, e tipi *mono-valore*, viene ora aggiunta la possibilità di utilizzare anche dei sottoinsiemi di questi (come potrebbero essere, ad esempio, intervalli di interi). Sulla base di questi tipi di base si possono definire i *tipi valore*, attraverso gli usuali costruttori di tipo definiti nei *CODMs*, quali *tuple*, *insiemi* e *tipi classe*, che denotano insiemi di oggetti con una identità ed un valore associato. Ai tipi può poi essere assegnato un nome, mantenendo la distinzione tra nomi di *tipi valore* e nomi di *tipi classe*, che d'ora in poi denomineremo semplicemente *classi*: ciò equivale a dire che i nomi dei tipi vengono partizionati in nomi che indicano insiemi di oggetti (*tipi classe*) e nomi che rappresentano insiemi di valori (*tipi valore*). ODL introduce inoltre la distinzione tra nomi di tipi *virtuali*, che descrivono condizioni necessarie e sufficienti per l'appartenenza di un oggetto del dominio ad un tipo (concetto che si può quindi collegare al concetto di *vista*), e nomi di tipi *primitivi*, che descrivono condizioni necessarie di appartenenza (e che quindi si ricollegano alle classi di oggetti). In [30], ODL è stato esteso per permettere la

formulazione dichiarativa di un insieme rilevante di vincoli di integrità definiti sulla base di dati. L'estensione di ODL con vincoli è stata denominata **OLCD**(**O**bject **L**anguage with **C**omplements allowing **D**escriptive cycles). Attraverso questa logica descrittiva, è possibile descrivere, oltre alle classi, anche le *regole di integrità*, permettendo la formulazione dichiarativa di un insieme rilevante di vincoli di integrità sottoforma di regole *if-then* i cui antecedenti e conseguenti sono espressioni di tipo ODL. In tale modo, è possibile descrivere correlazioni tra proprietà strutturali della stessa classe, o condizioni sufficienti per il popolamento di sottoclassi di una classe data.
In [16] è stato presentato il sistema OLCD-Designer, per l'acquisizione e la validazione di schemi OODB descritti attraverso **OLCD**, che preserva la consistenza della tassonomia di concetti ed effettua inferenze tassonomiche. In particolare, il sistema prevede un algoritmo di *sussunzione* che determina tutte le relazioni di specializzazione tra tipi, e un algoritmo che rileva gli eventuali tipi *inconsistenti*, cioè tipi necessariamente vuoti.
In [30] l'ambiente teorico sviluppato in [16] è stato esteso per effettuare l'ottimizzaziome semantica delle interrogazioni, dando vita al sistema ODB-QOptimizer. Sono state inoltre sviluppate delle interfacce software per tradurre descrizioni ed interrogazioni espresse rispettivamente nei linguaggi ODL e OQL (proposti dal gruppo di standardizzazione ODMG-93 [28, 29]) in **OLCD**.

La nozione di ottimizzazione semantica di una query è stata introdotta, per le basi di dati relazionali, da King [31, 32] e da Hammer e Zdonik [33]. L'idea di base di queste proposte è che i vincoli di integrità, espressi per forzare la consistenza di una base di dati, possano essere utilizzati anche per ottimizzare le interrogazioni fatte dall'utente, trasformando la query in una *equivalente*, ovvero con lo stesso insieme di oggetti di risposta, ma che può essere elaborata in maniera più efficiente.
Sia il processo di consistenza e classificazione delle classi dello schema, che quello di ottimizzazione semantica di una interrogazione, sono basati in ODB-Tools sulla nozione di *espansione* semantica di un tipo: l'espansione semantica permette di incorporare ogni possibile restrizione che non è presente nel tipo originale, ma che è logicamente implicata dallo schema (inteso come l'insieme delle classi, dei tipi, e delle regole di integrità). L'espansione dei tipi si basa sull'iterazione di questa trasformazione: se un tipo *implica* l'antecedente di una regola di integrità, allora il conseguente di quella regola può essere aggiunto alla descrizione del tipo stesso. Le *implicazioni* logiche fra i tipi (in questo caso il tipo da espandere e l'antecedente di una regola) sono determinate a loro volta utilizzando l'algoritmo di *sussunzione*, che calcola relazioni di sussunzione, simili alle relazioni di raffinamento dei tipi definite in [34].
Il calcolo dell'espansione semantica di una classe permette di rilevare nuove relazioni *isa*, cioè relazioni di specializzazione che non sono esplicitamente definite

dal progettista, ma che comunque sono logicamente implicate dalla descrizione della classe e dello schema a cui questa appartiene. In questo modo, una classe può essere automaticamente classificata all'interno di una gerarchia di ereditarietà. Oltre che a determinare nuove relazioni tra classi virtuali, il meccanismo, sfruttando la conoscenza fornita dalle regole di integrità, è in grado di riclassificare pure le classi base (generalmente gli schemi sono forniti in termini di classi base).
Analogamente, rappresentando a run-time l'interrogazione dell'utente come una classe virtuale (la query non è altro che una classe di oggetti di cui si definiscono le condizioni necessarie e sufficienti per l'appartenenza), questa viene classificata all'interno dello schema, in modo da ottenere l'interrogazione più specializzata tra tutte quelle semanticamente equivalenti a quella iniziale. In questo modo la query viene spostata verso il basso nella gerarchia e le classi a cui si riferisce vengono eventualmente sostituite con classi più specializzate: diminuendo l'insieme degli oggetti da controllare per dare risposta all'interrogazione, ne viene effettuata una vera ottimizzazione indipendente da qualsiasi modello di costo.

### 2.3.2 WordNet

**WordNet** [22] è un database lessicale elettronico considerato la più importante risorsa disponibile per i ricercatori nei campi della linguistica computazionale, dell'analisi testuale, e in altre aree associate. È stato sviluppato dal Cognitive science Laboratory alla Princeton University, sotto la direzione del Professor George A. Miller.

WordNet è un sistema di riferimento, disponibile on-line, la cui architettura è ispirata alle attuali teorie psicolinguistiche legate alla memoria lessicale umana.
Sostantivi, verbi, aggettivi e avverbi della lingua inglese vengono organizzati in insiemi di sinonimi (*synset*), ognuno dei quali rappresenta un determinato concetto lessicale. Vari tipi di relazioni collegano fra loro i synset.

Il punto di partenza della semantica lessicale è il riconoscimento che esiste una associazione convenzionale fra la forma delle parole (il modo in cui, cioè, vengono pronunciate e scritte) e i concetti che esse esprimono. La corrispondenza tra la forma delle parole e il significato che esprimono può essere sintetizzata in una tabella, la cosiddetta **Matrice lessicale** mostrata in figura 2.4

Dalla tabella di figura si può notare che la corrispondenza è di tipo molti-a-molti, da cui emergono due proprietà:

- *Polisemia*: proprietà di una stessa parola di avere due o più significati. Nel-

| Word Meanings | Word Forms | | | | |
|---|---|---|---|---|---|
| | $F_1$ | $F_2$ | $F_3$ | . . . . . | $F_n$ |
| $M_1$ | $E_{1,1}$ | $E_{1,2}$ | | | |
| $M_2$ | | $E_{2,2}$ | | | |
| $M_3$ | | | $E_{3,3}$ | | |
| . . . | | | | . . . . . | |
| $M_m$ | | | | | $E_{m,n}$ |

Figura 2.4: La Matrice Lessicale

la matrice compaiono due o più elementi in colonna, come, ad esempio, accade per la forma $F_2$;

- *Sinonimia*: proprietà di un significato di avere due o più parole in grado di esprimerlo. Nella matrice compaiono due o più elementi in riga, ad esempio in corrispondenza di $M_1$.

Polisemia e Sinonimia sono problemi lessicografici tra loro complementari. Mentre è in atto una comunicazione in forma scritta/orale, da un lato il lettore/ascoltatore riceve la parola, cercando di capirne il significato tra tutti quelli che la stessa può esprimere, dall'altro lo scrittore/oratore conosce il concetto che vuole esprimere e si trova davanti la scelta di una fra le varie forme cho lo riassumono.

WordNet rappresenta i concetti seguendo la teoria cosiddetta *differenziale*, secondo cui i diversi significati non sono necessariamente denotati da definizioni scritte, ma vengono rappresentati e distinti tra loro, attraverso l'uso di differenti simboli conosciuti dall'utente, ed ai quali l'utente stesso associa il concetto. In altri termini, questa scelta parte dal presupposto che l'utente conosca già sufficientemente bene la lingua (nella fattispecie l'inglese), e sia pertanto in grado di riconoscere il significato di una determinata parola in base ai vari sinonimi che la affiancano all'interno di un synset, senza che ne venga data esplicitamente una definizione.

### Tipi di relazioni

Occorre premettere che la distinzione fondamentale operata da WordNet sui termini consiste nel suddividerli in sostantivi, verbi, aggettivi e avverbi. Non si vedranno pertanto mai accostate, ad esempio in un synset, parole appartenenti a categorie differenti.

Il database implementato collega i termini in base a relazioni semantiche. Poiché una relazione semantica è una relazione fra significati, e visto che i significati, a causa della sinonimia, sono associati a set di termini sinonimi, è naturale pensare alle relazioni semantiche come puntatori che collegano non i singoli termini, ma bensì gli insiemi di sinonimi.
Da ciò emerge una distinzione tra la relazione di sinonimia e le altre relazioni semantiche, denotata anche dal fatto che ogni insieme di termini sinonimi è racchiuso fra parentesi graffe $\{\}$, mentre gli insiemi prodotti da tutte le altre relazioni sono racchiusi fra parentesi quadre [ ].
Le relazioni semantiche godono di due proprietà:

- Se esiste una relazione $R$ fra gli insiemi $\{x,x'$, ...$\}$ e $\{y,y'$, ...$\}$, allora deve esistere una relazione inversa $R'$ fra $\{y,y'$, ...$\}$ e $\{x,x'$, ...$\}$. $R$ e $R'$ possono, non necessariamente, coincidere.

- Se esiste una relazione $R$ fra gli insiemi $\{x,x'$, ...$\}$ e $\{y,y'$, ...$\}$, allora vale [$x\ R\ y$], [$x\ R\ y'$], ... , [$x'\ R\ y$], etc...

**Synonymy** Come emerso, la sinonimia tra termini è la relazione più importante, perché ogni insieme *synset* che ne scaturisce rappresenta la semantica di un concetto.

Due termini si definiscono sinonimi se possono essere indifferentemente interscambiati senza che il significato cambi. Notare che, se esistono, sono molto rari i sinonimi reali. Più comunemente si parla di sinonimi riferiti ad un particolare contesto, perciò la definizione diventa:
*Due termini sono sinonimi in un contesto linguistico C se l'uso di uno o dell'altro in C non altera il reale valore*.

**Antonymy** Due termini sono in relazione di antonimia se uno è il contrario dell'altro.
Notare che l'antonimo del termine $x$ non sempre coincide con '$non\ x$': ad esempio 'povero' e 'ricco' sono in relazione di antonimia, ma 'non ricco' non coincide con 'povero', dal momento che un uomo può essere contemporaneamente 'non ricco' e 'non povero'. Al contrario delle relazioni che seguono, l'antonimia è

di tipo lessicale, si applica cioè fra singoli termini e non fra concetti espressi da synset.
Ad esempio non si può affermare che {rise,ascend} e {fall,discend} siano antonimi, pur essendolo singolarmente [rise/fall] (e anche [ascend/discend]).

**Hyponymy** La relazione di iponimia è semantica, lega cioè i concetti, e quindi i synset.
*Un concetto è iponimo di un altro quando lo specializza: tra i due esiste un rapporto di tipo ISA.* Ad esempio 'maple' (acero) è iponimo di 'tree' (albero) e 'tree' è iponimo di 'plant' (pianta). L'iponimia gode della proprietà transitiva: nell'esempio si deduce che 'maple' è iponimo di 'plant'.
Questa proprietà consente la costruzione si sistemi ereditari, vere e proprie gerarchie nelle quali ogni concetto iponimo eredita tutte le caratteristiche del suo superconcetto e ne aggiunge almeno una che lo distingue dallo stesso superconcetto e da qualsiasi suo altro iponimo.

L'iponimia è inoltre una relazione asimmetrica: il suo termine duale è **Hypernymy** (Ipernimia).

**Meronymy** La Meronimia è una relazione semantica che si esprime fra due concetti $x$ e $y$ quando $x$ *is a part of* $y$.
Come la relazione precedente, anche la Meronimia gode della proprietà transitiva, ed è asimmetrica: la relazione duale è **Holonymy**.
Come nel caso precedente si possono realizzare gerarchie di concetti meronimi/olonimi, con la differenza che, in questo caso, uno stesso meronimo può avere più olonimi: in altri termini uno stesso componente può contemporaneamente far parte di differenti concetti composti.

I diversi tipi di relazioni appena descritte rappresentano associazioni che, nel database implementato da WordNet, formano una rete complessa. Data una parola, secondo la teoria *differenziale* adottata, il suo significato si determina in base alla collocazione che la stessa ha all'interno della rete appena citata.

**Morfologia delle parole** Un aspetto che finora non è stato considerato, riguarda la morfologia delle parole che, nella lingua inglese, si manifesta come segue:

- declinazioni per i sostantivi, ad esempio distinzione fra singolare e plurale;
- coniugazioni per i verbi.

Nelle sue prime versioni WordNet non teneva conto di questo aspetto, perciò le sue funzionalità risultavano fortemente limitate: basti pensare all'inserimento

di un termine plurale che non viene riconosciuto.
Attualmente è stato inserito un software che non modifica la sostanza del database, nel senso che al suo interno le parole non vengono replicate in tutte le possibili forme, ma sono archiviate nella sola forma canonica. Il compito di questo componente è quello di interfacciarsi fra l'utente e il database lessicale: ogni termine in input viene tradotto in forma canonica e successivamente inviato al database.
Nonostante la morfologia della lingua inglese sia, rispetto ad altre lingue, abbastanza semplice, la realizzazione di questo componente non lo è stata, a causa della massiccia presenza di coniugazioni di verbi irregolari.

Come si vedrà nel capitolo 3, il sistema Momis sfrutta relazioni terminologiche di tipo synonymy (SYN), hyponymy (NT), hypernymy (BT) e meronimy/holonymy (considerata come relazione associativa RT)[1]. Momis passa a WordNet i termini usati per rappresentare i nomi di classi e attributi, e riceve in risposta tutte le relazioni esistenti fra essi.

## 2.4 Le sorgenti semistrutturate

Come detto, Il sistema MOMIS è in grado di trattare dati provenienti da sorgenti strutturate e semistrutturate. Per quanto riguarda i database strutturati convenzionali, la descrizione dello schema è sempre disponibile e può quindi essere tradotto automaticamente in linguaggio $ODL_{I^3}$ da uno specifico wrapper. Diverso è il discorso delle sorgenti semistrutturate, dove, a priori, non è definito alcuno schema. Per questa ragione risulta necessaria una procedura di estrazione dello schema per le sorgenti semistrutturate (e.g., nella forma di dataguides [35]).
Questo argomento, riguardante il *wrapping*, esula dalla trattazione di questa tesi, tuttavia è opportuno accennare brevemente a come MOMIS si comporta riguardo ai dati semistrutturati. In MOMIS introduciamo il concetto di *object pattern* per rappresentare a livello intensionale la struttura degli oggetti di una sorgente semistrutturata.

### 2.4.1 Dati semistrutturati e object patterns

L'incremento nel numero e nella dimensione delle sorgenti informative che in questi anni si sono rese disponibili all'utente generico è stato accompagnato da

[1] Attualmente la relazione antonymy non viene considerata; si potrebbe tuttavia pensare di sfruttarla in futuro come ulteriore conoscenza, interpretando l'antonimia tra due classi come relazione di disgiunzione estensionale.

un'esplosione nella varietà dei formati in cui i dati vengono rappresentati. Nella maggior parte dei casi, purtroppo, sebbene si possa riconoscere nei dati una sorta di struttura, questa è talmente irregolare da non poter essere facilmente riconducibile né ai consolidati modelli relazionali [36] e neppure a quelli più ricchi, quali quelli ad oggetti [28]. In genere si parla di questi dati in termini di dati semistrutturati (semistructured data), come confermato da diversi interventi presenti in letteratura [37, 38]. A causa della natura frammentata dei dati, l'enfasi degli interventi viene posta sull'importanza di derivare una rappresentazione concisa della struttura, in modo tale da fornire all'utente, che si trova ad interagire con le informazioni, un'idea della struttura e del contesto riguardante la sorgente dei dati. Questa possibilità di giungere alla definizione di una struttura per dati semistrutturati facilita la formulazione di query e potrebbe essere usata per la fase di ottimizzazione delle stesse.

La caratteristica di base dei dati semistrutturati risiede nella loro natura che può essere definita "auto-descrittiva" ("self-describing"). Questo significa che le informazioni, che generalmente vengono associate allo schema, sono direttamente specificate negli stessi dati. Come detto, in letteratura vari sono i modelli proposti per la rappresentazione dei dati semistrutturati [38, 39, 40]. In accordo a questi modelli, la nostra proposta prevede di rappresentare sorgenti semistrutturate come grafi ad albero etichettati, dove i dati semistrutturati possono rappresentare sia i nodi che le etichette sugli archi. Infatti, supponiamo di utilizzare sorgenti semistrutturate conformi al modello OEM [41, 40], la cui definizione è la seguente:

**Definizione 1 (Oggetto Semistrutturato)** *Un* oggetto semistrutturato*, indicato con $so$, è una tripla nella forma* $\langle$`id`, `label`, `value`$\rangle$ *dove:*

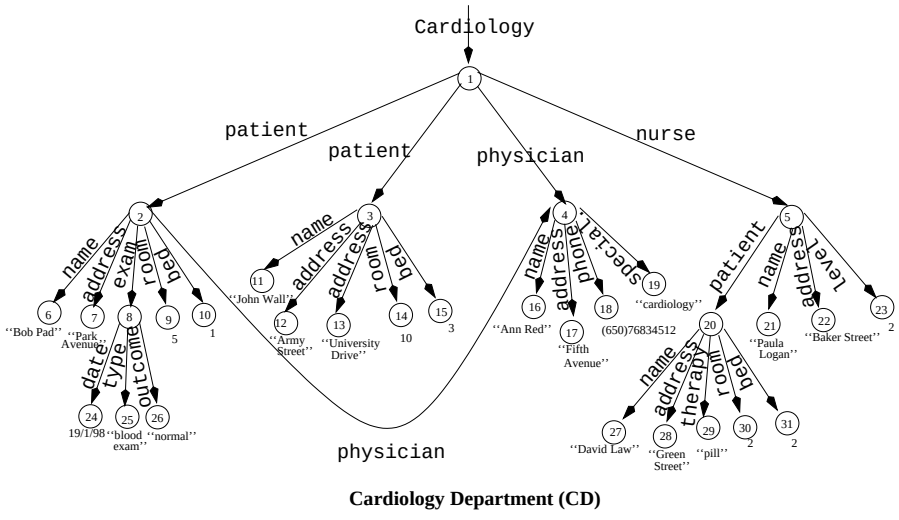
- `id` *è l'idenficatore dell'oggetto;*
- `label` *è una stringa che descrive ciò che l'oggetto rappresenta;*
- `value` *è il valore dell'oggetto, che può essere atomico o complesso.*

Per quanto concerne i valori atomici, essi possono essere *integer*, *real*, *string*, *image*, mentre un valore complesso è un insieme di oggetti semistrutturati, cioè un insieme di coppie (*id*,*label*). In analogia a questa caratterizzazione, definiamo *oggetto semistrutturato atomico* un oggetto a valore atomico, e *oggetto semistrutturato complesso* un oggetto il cui valore è complesso.

Per esempio,$\langle$`16`, `name`, "`Ann Red`"$\rangle$ rappresenta un oggetto semistrutturato atomico con valore di tipo stringa, mentre l'oggetto $\langle$`8`, `exam`, {`(24, date), (25, type), (26, outcome)`}$\rangle$ rappresenta un oggetto semistrutturato

complesso il cui valore è dato da tre oggetti atomici. L'oggetto complesso può quindi essere pensato come il padre di tutti gli oggetti che definiscono il suo valore (detti oggetti figli). In generale, un dato oggetto può avere uno o più genitori: nella nostra notazione indichiamo che un oggetto $so'$ è figlio di un altro oggetto $so$ tramite $so \rightarrow so'$ e indichiamo con *label*$(so)$ l'etichetta di $so$. Perciò, intuitivamente, una sorgente semistrutturata $S$ può essere vista come un grafo orientato, in cui i nodi rappresentano gli oggetti semistrutturati e gli archi orientati rappresentano la relazione tra un oggetto complesso e tutti gli oggetti atomici che ne individuano il valore.

Un esempio di sorgente semistrutturata è rappresentato in figura 2.5, dove sono presenti informazioni relative al Dipartimento di Cardiologia (`Cardiology Department`) di un dato ospedale.




**Cardiology Department (CD)**

Figura 2.5: Cardiology Department (CD)

Nei modelli di dati semistrutturati, le etichette sono il più possibile esplicative e possono essere utilizzate per raggruppare gli oggetti assegnando la stessa etichetta ad oggetti correlati. Per questa ragione, data una sorgente semistrutturata $S$, vogliamo scoprire i tipi differenti di oggetti memorizzati attraverso un approccio basato sui pattern, che tenga conto della semantica di mondo aperto caratteristica delle Description Logics. L'approccio si può riassumere brevemente nel seguente modo: tutti gli oggetti complessi *so* di $S$ vengono partizionati in base alle loro etichette e valori in insiemi disgiunti, indicati come $set_l$, in modo tale che tutti gli oggetti che fanno riferimento allo stesso insieme abbiano la stessa etichetta $l$. Successivamente ad ogni insieme associamo un *object pattern*. Formalmente, un *object pattern* è definito nel seguente modo:

**Definizione 2 (Object pattern)** *Dato un insieme di oggetti set$_l$ definito sulla sorgente semistrutturata S. L'*object pattern *dell'insieme set$_l$ è una coppia nella forma* $\langle l,A \rangle$*, dove* $l$ *è l'etichetta degli oggetti correlati all'insieme set$_l$, ed* $A = \bigcup$ *label*$(so')$ *tale che esiste almeno un oggetto* $so \in$ *set$_l$ con* $so \rightarrow so'$.

Partendo da questa definizione, è facile capire come un *object pattern* sia rappresentativo di tutti i diversi oggetti che descrivono lo stesso concetto in una data sorgente semistrutturata. Più specificatamente, $l$ indica il concetto e $A$ le proprietà (dette anche attributi) che caratterizzano il concetto all'interno della sorgente. Poiché gli oggetti semistrutturati possono essere eterogenei, a volte le etichette nell'insieme $A$ di un *object pattern* possono essere definite solo per alcuni oggetti dell'insieme set$_l$ e non per tutti: chiameremo queste particolari etichette come "optional", indicandole con il simbolo "*".

In figura 2.6 sono indicati tutti gli *object pattern* della sorgente semistrutturata considerata come esempio. Sono stati definiti quattro *object pattern*: `Patient` che contiene informazioni riguardo ai pazienti; `Physician` e `Nurse` contenenti informazioni riguardo al personale medico; `Exam` contenente informazioni riguardo la `date`, il `type` e `outcome` di un esame clinico.

| | |
|---|---|
| Patient-pattern | = (`Patient`,{`name,address,exam*,room,bed,therapy*,physician*`}) |
| Physician-pattern | = (`Physician`,{`name,address,phone,specialization`}) |
| Nurse-pattern | = (`Nurse`,{`name,address,level,patient`}) |
| Exam-pattern | = (`Exam`,{`date,type,outcome`}) |

Figura 2.6: Gli object pattern della sorgente semistrutturata di esempio

Come detto la descrizione degli *object pattern* segue la semantica di mondo aperto *open world semantics* tipica dell'approccio seguito in Description Logics [42, 43, 44, 27]. Gli oggetti di un pattern condividono una struttura minima rappresentata dalle proprietà non opzionali, ma possono avere altre proprietà addizionali che caratterizzano il singolo oggetto (o un numero ridotto di oggetti) che chiamiamo optional. In questo modo, gli oggetti di una sorgente semistrutturata possono evolversi ed aggiungere nuove proprietà, pur rimanendo istanze valide dell'*object pattern* corrispondente e quindi mantenendo efficaci le query sull'*object pattern*.

Il nostro approccio si pone quindi l'obiettivo di distinguere, all'interno di una sorgente semistrutturata, la parte relativa alla struttura da quella dei dati: il livello intensionale contiene ciascuna descrizione degli *object pattern* secondo la descrizione ODL$_{I^3}$. Il livello estensionale contiene invece gli oggetti semistrutturati (vedi figura 2.7).

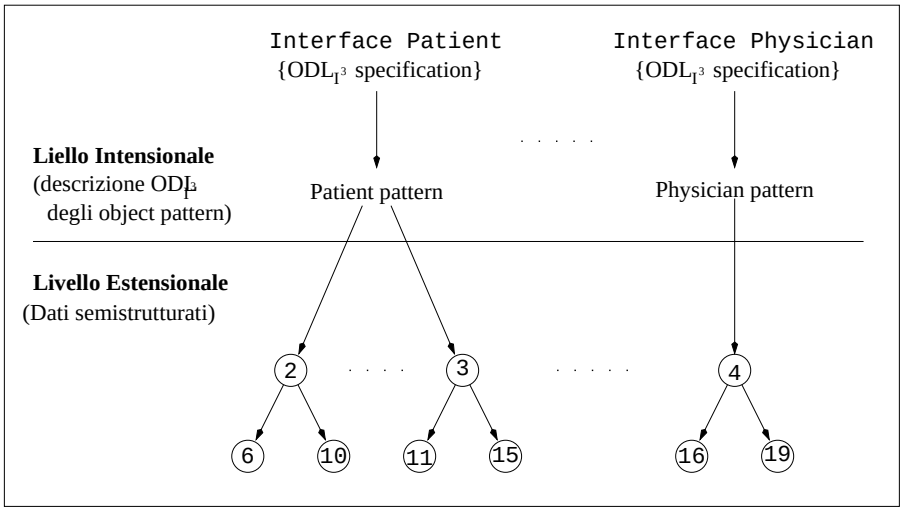



Figura 2.7: Livelli di astrazione degli oggetti semistrutturati

## 2.5 Esempio di riferimento

Nei prossimi capitoli verranno ripresi in dettaglio tutti i passaggi che vengono effettuati in fase di integrazione e di query processing. Al fine di rendere chiari tutti i passaggi, consideriamo il seguente esempio di riferimento, riguardante un dominio applicativo di una realtà ospedaliera.

Si consideri un ambiente ospedaliero ed in particolare i Dipartimenti di cardiologia (`Cardiology`) e quello di terapia intensiva (`Intensive Care`), dove, ovviamente, c'è l'esigenza di condividere le informazioni riguardo i propri pazienti. Il dipartimento `Cardiology` (brevemente indicato come `CD`) contiene oggetti di tipo semistrutturato relativi a pazienti con malattie di tipo ischemico, cardiocircolatorio ed ipertensione, ed informazioni riguardo il personale medico afferente il dipartimento. In figura 2.5 è riportata una porzione di esempio dei dati: abbiamo un oggetto complesso radice con quattro figli anch'essi di tipo complesso, due pazienti, un medico ed una nurse. (Ovviamente nell'applicazione reale ci sono molti pazienti, medici e nurse). Ciascun `Patient` è dotato di oggetti atomici `name`, `address`, `room`, e `bed`. `Physician` contiene a sua volta quattro oggetti atomici (i.e., `name`, `address`, `phone`, e `specialization`), mentre `Nurse` contiene come valori tre oggetti atomici (i.e., `name`, `address`, e `level`) ed uno complesso (i.e., il `patient` di cui è responsabile ).

Il dipartimento `Intensive Care` (`ID`) memorizza le proprie informazioni in un database relazionale relative a pazienti la cui diagnosi riguarda trauma o

**Intensive Care Department (ID)**

```
Patient(code, first_name, last_name, address, test, doctor_id)
Doctor(id, first_name, last_name, phone, address, availability, position)
Test(number, type, date, laboratory, result)
Dis_Patient(code, date, note)
```

Figura 2.8: Intensive Care Department (ID)

infarto e sullo staff medico. Il database contiene quattro relazioni: `Patient`, `Doctor`, `Test` e `Dis_Patient` (vedere figura 2.8). Le informazioni personali relative ai pazienti sono mantenute nella relazione `Patient`. `Dis_Patient` è un sottotipo di `Patient` e contiene informazioni riguardo ai pazienti dimessi. Ciascun "clinician", cioè il personale medico quali dottori, infermieri, farmacisti (che costituiscono la relazione `Doctor`) è caratterizzato da `first_name`, `last_name`, `phone`, `address`, `availability` e `position`. La relazione `Test` è usata per mantenere informazioni riguardanti tutti gli esami dei pazienti.

# Capitolo 3

# La costruzione dello schema integrato

In questo capitolo verranno analizzate in dettaglio le fasi che caratterizzano il processo di integrazione, facendo riferimento all'esempio intrododdo nella sezione 2.5.

## 3.1 Generazione del Thesaurus

Nell'ambito di un approccio semantico alla integrazione di dati (quale è quello adottato nel progetto **Momis**) è di fondamentale importanza la conoscenza delle informazioni semantiche relative al contesto e alla struttura dei vari schemi sorgenti.
Tale conoscenza è contenuta nel cosiddetto *Common Thesaurus*, un dizionario all'interno del quale sono presenti un insieme di relazioni terminologiche che legano tra loro classi ed attributi.

Dato l'insieme $S = \{S_1, S_2, \ldots, S_n\}$ dei sorgenti da integrare, sia $c_{ij} \in S_i$ la generica classe del source di posto $i$. Ogni classe è caratterizzata da un nome e da un insieme: $c_{ij} = \langle n_{c_{ij}}, A(c_{ij})\rangle$. $A(c_{ij})$ rappresenta l'insieme degli attributi della classe $c_{ij}$, ognuno dei quali, a sua volta, possiede un nome ed un dominio di valori che può assumere: $A(c_{ij}) = \{a_1, a_2, ..., a_h, ...\}$ con $a_h = \langle n_h, d_h\rangle$.
Sia $T$ l'insieme dei nomi di tutte le classi e di tutti gli attributi appartenenti ai vari schemi sorgenti.
Dati due diversi nomi di classi e/o attributi $t_i, t_j \in T$ tali che $t_i \neq t_j$, le relazioni che possono legare questi due termini sono le seguenti:

- SYN (synonym_of): i due termini sono sinonimi, cioè possono essere interscambiati nelle sorgenti perché rappresentano lo stesso concetto della re-

altà. La relazione SYN gode della proprietà di simmetria, ovvero $\langle t_i$ SYN $t_j\rangle \Longleftrightarrow \langle t_j$ SYN $t_i\rangle$. Nel nostro esempio si ha $\langle$`Test` SYN `Exam`$\rangle$.

- BT (boader_term): $\langle t_i$ BT $t_j\rangle$ equivale ad affermare che $t_i$ ha un significato più ampio di $t_j$. Ad esempio $\langle$`Medical_staff` BT `Nurse`$\rangle$.

- NT (narrower_term): questa relazione esprime il concetto opposto della relazione BT, ovvero $\langle t_i$ NT $t_j\rangle \Longleftrightarrow \langle t_j$ BT $t_i\rangle$. Nel nostro esempio, dunque, si avrà $\langle$`Nurse` NT `Medical_staff`$\rangle$.

- RT (related_term) definita fra due termini che in qualche modo sono legati, perché appartenenti al medesimo contesto. La relazione RT gode della proprietà di simmetria, cioè $\langle t_i$ RT $t_j\rangle \Longleftrightarrow \langle t_j$ RT $t_i\rangle$. Un possibile esempio di relazione RT è $\langle$`Patient` RT `Exam`$\rangle$.

.

In figura 3.1 sono evidenziate le fasi costituenti il processo che porta alla generazione del Thesaurus comune.
Lo sforzo compiuto a questo livello è quello di rendere il processo il più possibile automatico, minimizzando l'intervento del progettista, la cui partecipazione si rende comunque indispensabile.

### 3.1.1 Estrazione di relazioni dalla struttura degli schemi

La struttura di ogni schema locale contiene delle informazioni semantiche ricavabili dalle gerarchie di ereditarietà e aggregazione. Terminata la fase di acquisizione degli schemi locali, il sistema è quindi in grado di estrarre in modo automatico alcune relazioni.
In particolare:

1. negli schemi ad oggetti (e semistrutturati tradotti ad oggetti):

   - relazioni BT e NT derivanti dalle gerarchie di ereditarietà
   - relazioni RT derivanti dalle gerarchie di aggregazione

2. negli schemi relazionali:

   - relazioni RT derivanti dalle foreign key

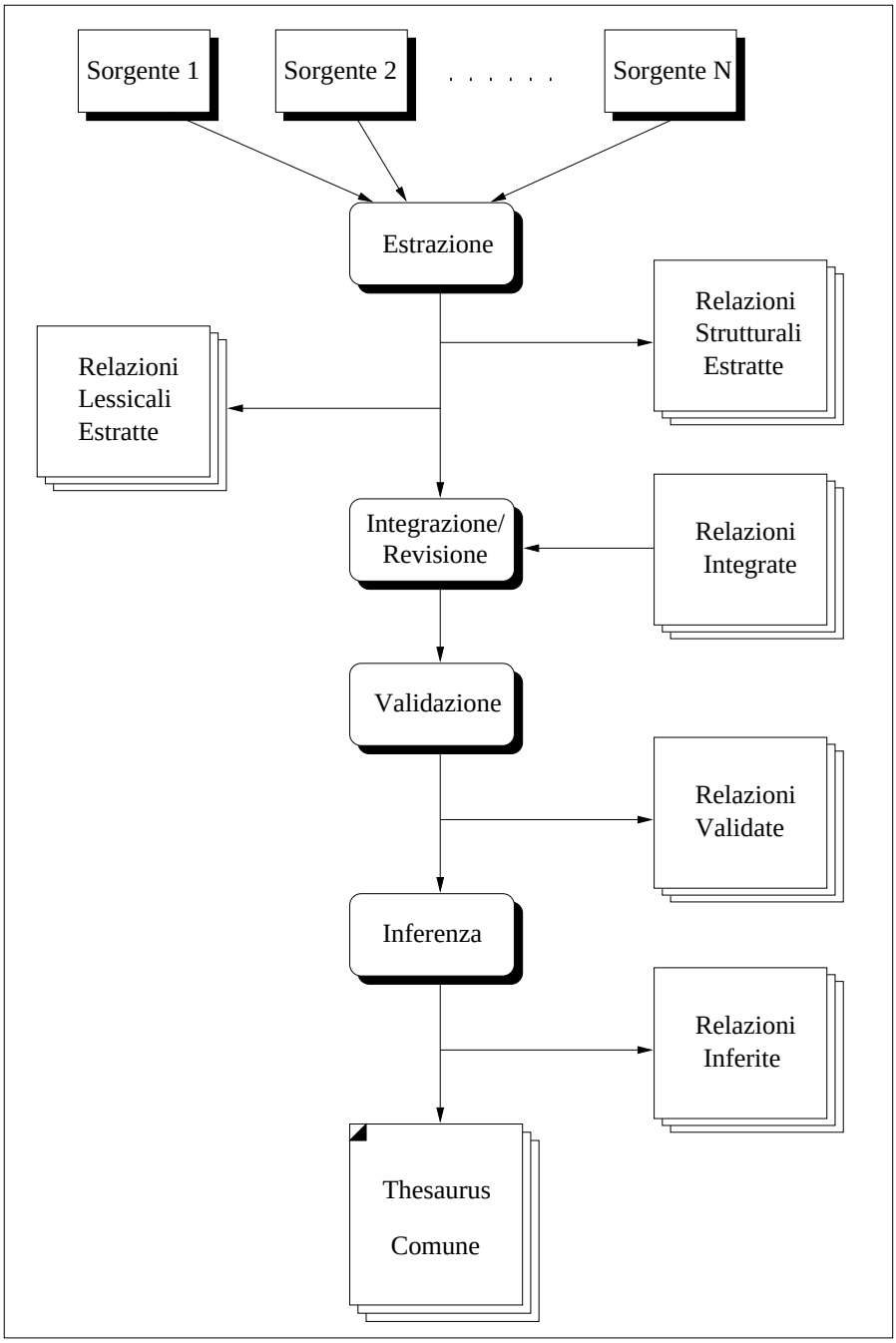



Figura 3.1: Il processo di generazione del Thesaurus comune

**Nell'esempio:** Prendendo spunto dall'esempio relativo alla realtà ospedaliera, le relazioni estratte sono le seguenti:

⟨ID.Patient RT ID.Doctor⟩
⟨ID.Patient BT ID.Dis_Patient⟩
⟨ID.Patient RT ID.Test⟩
⟨CD.Nurse RT CD.Patient⟩
⟨CD.Patient RT CD.Physician⟩

Si può facilmente notare che le relazioni estratte in questa prima fase di costruzione del thesaurus hanno le seguenti peculiarità:

- Sono tutte intraschema, cioè legano tra loro coppie di classi appartenenti allo stesso schema sorgente: ciò è dovuto al fatto che sono ricavate da schemi che, a livello di struttura, sono tra loro completamente disgiunti

- Non esistono relazioni di tipo SYN: infatti all'interno di uno stesso schema, due classi strutturalmente identiche sono necessariamente la stessa classe

### 3.1.2 Estrazione lessicale di relazioni

Un ulteriore insieme di relazioni può essere estratto a partire dall'insieme T dei nomi di classi ed attributi. L'analisi che viene svolta è di tipo lessicale, basata sul significato dei termini, e si avvale del database lessicale WordNet [22], descritto in sezione 2.3.2.
Le relazioni che scaturiscono da questa fase, a differenza di quanto accade nella precedente, possono includere la sinonimia tra termini e sono principalmente interschema.

**Nell'esempio:** Vengono estratte le seguenti relazioni:

⟨doctor BT physician⟩
⟨test SYN exam⟩
⟨name BT first_name⟩
⟨name BT last_name⟩

Rapportate alle entità presenti negli schemi dell'esempio si trasformano come segue:

⟨`ID.Doctor` BT `CD.Physician`⟩
⟨`ID.Test` SYN `CD.Exam`⟩
⟨`CD.Patient.name` BT `ID.Patient.first_name`⟩
⟨`CD.Patient.name` BT `ID.Patient.last_name`⟩
⟨`CD.Patient.name` BT `ID.Doctor.first_name`⟩
⟨`CD.Patient.name` BT `ID.Doctor.last_name`⟩
⟨`CD.Physician.name` BT `ID.Patient.first_name`⟩
⟨`CD.Physician.name` BT `ID.Patient.last_name`⟩
⟨`CD.Physician.name` BT `ID.Doctor.first_name`⟩
⟨`CD.Physician.name` BT `ID.Doctor.last_name`⟩
⟨`CD.Nurse.name` BT `ID.Patient.first_name`⟩
⟨`CD.Nurse.name` BT `ID.Patient.last_name`⟩
⟨`CD.Nurse.name` BT `ID.Doctor.first_name`⟩
⟨`CD.Nurse.name` BT `ID.Doctor.last_name`⟩

Notare come da un unica relazione terminologica se ne possano ricavare anche più di una tra classi e/o attributi locali.

### 3.1.3 Revisione/Integrazione delle relazioni

A questo punto del processo si può affermare che le relazioni estratte dalla struttura degli schemi sorgenti siano esatte, perché tali sono le informazioni da cui derivano. Lo stesso non si può dire delle relazioni derivate attraverso il dizionario lessicale, dal momento che potrebbero essere messi in relazione termini che, nel particolare contesto in esame, sono completamente estranei. Per questo motivo in questa fase interviene il progettista, che ha la possibilità di modificare o cancellare relazioni e, soprattutto, aggiungerne di nuove all'insieme fin qui prodotto.

**Nell'esempio:** Il progettista aggiungerà le seguenti relazioni terminologiche:

⟨`ID.Doctor` BT `CD.Nurse`⟩
⟨`CD.Patient.physician` BT `ID.Patient.doctor_id`⟩
⟨`CD.Nurse.level` SYN `ID.Doctor.position`⟩
⟨`CD.Exam.outcome` SYN `ID.Test.result`⟩

### 3.1.4 Validazione delle relazioni

In questa fase il sistema analizza tutte le relazioni fra attributi aggiunte nelle due fasi precedenti e decide quali tra queste sono ritenute *valide* e quali no. Infatti, se da un punto di vista ontologico due termini rappresentano concetti della realtà tra loro correlati, non è detto che i domini dei relativi attributi siano allo stesso modo compatibili.

Siano $a_t = \langle n_t, d_t \rangle$ e $a_q = \langle n_q, d_q \rangle$ una coppia di attributi posti in relazione tra loro.
Il compito di confrontarne i rispettivi domini è proprio di ODB-Tools che, a seconda del tipo di relazione adotta i sequenti criteri:

- $\langle n_t$ SYN $n_q \rangle$: la relazione è *valida* se i domini $d_t$ e $d_q$ sono equivalenti o se uno dei due è più specializzato dell'altro;
- $\langle n_t$ BT $n_q \rangle$: la relazione è *valida* se $d_t$ contiene o è equivalente a $d_q$;
- $\langle n_t$ BT $n_q \rangle$: analogamente al caso precedente la relazione è *valida* se $d_t$ è un sottoinsieme non proprio di $d_q$;

**Nota bene**: nel caso di dati semistrutturati, la cui traduzione nel linguaggio ad oggetti comune $\text{ODL}_{I^3}$ prevede che una classe possa avere più implementazioni in *union*, spesso accade che uno stesso attributo abbia domini diversi: in tal caso ogni relazione che coinvolge quell'attributo è *valida* se almeno uno dei suoi domini rispetta i criteri sopra elencati.

**Nell'esempio:** Ad ogni relazione esaminata viene associato un *flag* booleano che rappresenta il risultato di questa fase.
Riferendoci all'esempio, ODB-Tools produce il seguente risultato:

| | |
|---|---|
| ⟨`CD.Patient.name` BT `ID.Patient.first_name`⟩ | [1] |
| ⟨`CD.Patient.name` BT `ID.Patient.last_name`⟩ | [1] |
| ⟨`CD.Patient.name` BT `ID.Doctor.first_name`⟩ | [1] |
| ⟨`CD.Patient.name` BT `ID.Doctor.last_name`⟩ | [1] |
| ⟨`CD.Physician.name` BT `ID.Patient.first_name`⟩ | [1] |
| ⟨`CD.Physician.name` BT `ID.Patient.last_name`⟩ | [1] |
| ⟨`CD.Physician.name` BT `ID.Doctor.first_name`⟩ | [1] |
| ⟨`CD.Physician.name` BT `ID.Doctor.last_name`⟩ | [1] |
| ⟨`CD.Nurse.name` BT `ID.Patient.first_name`⟩ | [1] |
| ⟨`CD.Nurse.name` BT `ID.Patient.last_name`⟩ | [1] |
| ⟨`CD.Nurse.name` BT `ID.Doctor.first_name`⟩ | [1] |
| ⟨`CD.Nurse.name` BT `ID.Doctor.last_name`⟩ | [1] |
| ⟨`CD.Patient.physician` BT `ID.Patient.doctor_id`⟩ | [0] |
| ⟨`CD.Nurse.level` SYN `ID.Doctor.position`⟩ | [0] |
| ⟨`CD.Exam.outcome` SYN `ID.Test.result`⟩ | [1] |

### 3.1.5 Inferenza di nuove relazioni

Sulla base delle relazioni fin qui ottenute, attraverso un procedimento automatico che fa uso di tecniche di inferenza, viene generato un nuovo insieme di relazioni

legali tra classi, che contribuiscono ad arricchire ulteriormente (e completare) il thesaurus.
Per poter agire in questo senso occorre effettuare alcune modifiche agli schemi locali; è importante precisare che tali modifiche hanno carattere provvisorio, servono solamente per il calcolo di deduzione di nuove relazioni inferite.

Le **classi** sono inserite nello schema e collegate tra loro in modo da conformarsi alle relazioni esistenti:

- ogni relazione BT e NT dà luogo a una gerarchia di ereditarietà;
- la relazione SYN produce una doppia ereditarietà, da cui emerge come le due classi siano vicendevolmente in rapporto *is_a*;
- ogni relazione RT produce un'aggregazione

Notare come, oltre a ricomporsi i singoli schemi locali, le relazioni lessicali aggiungano nuovi collegamenti interschema, in modo tale che alla fine risulti un unico schema.

Gli **attributi** coinvolti nelle relazioni validate vengono organizzati in gerarchie nelle quali la sinonimia pone due attributi allo stesso livello, mentre le relazioni BT e NT pongono a livello superiore il termine più generale. Alla fine risultano diversi alberi gerarchici isolati; ogni attributo viene rinominato, in particolare assume il nome del termine di più alto livello dell'albero cui appartiene.

Quest'ultima operazione ha lo scopo di rendere il più possibile confrontabili le intensioni dello schema prodotto dall'analisi delle relazioni tra le classi. In questo modo l'intervento di ODB-Tools, attraverso tecniche di intelligenza artificiale basate sul calcolo di Sussunzione, è in grado di inferire tra le classi locali nuove gerarchie di aggregazione ed ereditarietà, che si traducono facilmente in nuove relazioni per il thesaurus.

In figura 3.2 viene mostrato lo schema relativo all'esempio ospedaliero, che rappresenta anche la forma grafica del Thesaurus per quanto riguarda le relazioni fra classi locali: le linee grosse rappresentano gerarchie di ereditarietà, mentre quelle sottili gerarchie di aggregazione. Un'ulteriore distinzione viene fatta fra le relazioni esplicite preesistenti (linee continue) e quelle inferite (linee tratteggiate), tradotte come segue:

⟨ID.Patient RT CD.Physician⟩
⟨CD.Patient RT ID.Doctor⟩
⟨ID.Patient RT CD.Exam⟩
⟨CD.Patient RT ID.Test⟩
⟨ID.Dis_Patient RT CD.Physician⟩
⟨ID.Dis_Patient RT ID.Doctor⟩
⟨ID.Dis_Patient RT CD.Exam⟩
⟨ID.Dis_Patient RT ID.Test⟩

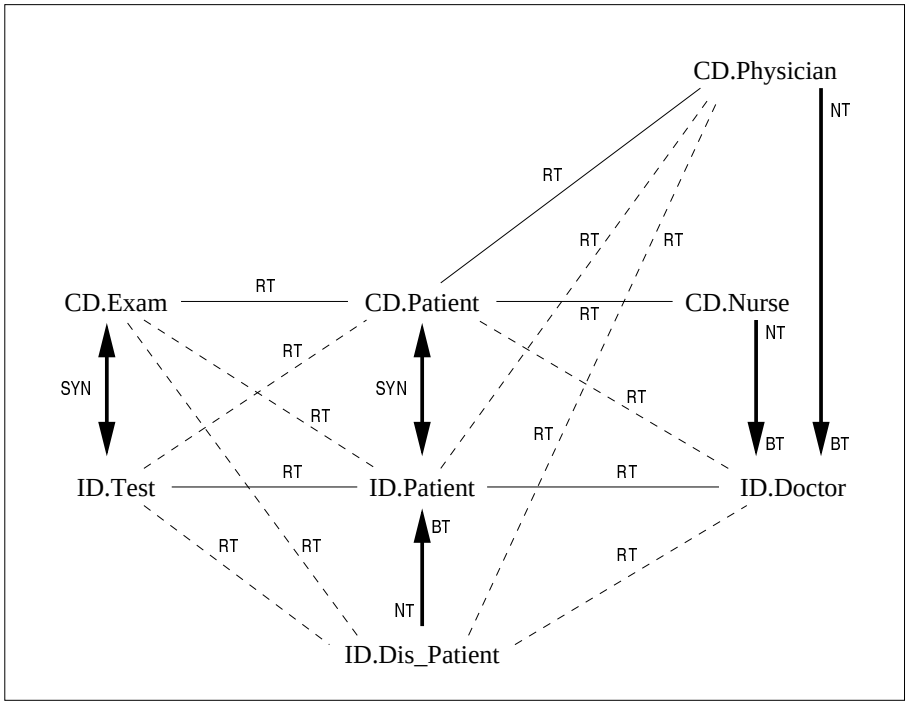



Figura 3.2: Rappresentazione grafica del Thesaurus comune

## 3.2 Calcolo delle affinità

La conoscenza semantica contenuta nel Thesaurus comune appena generato rappresenta il punto di partenza per il successivo processo che porta alla definizione dello schema globale integrato. Scopo di questa parte, descritta approfontitamente in [45], è quantificare il grado di *affinità* mutuo fra le classi locali, al fine di raggrupparle opportunamente in diversi *cluster*, da cui sarà possibile generare la vista globale.

Il livello di similarità tra le classi, si calcola su due differenti dimensioni:

- Da un punto di vista strutturale, attraverso lo *Structural Affinity Coefficient*, calcolato tra due classi sulla base delle relazioni tra i loro attributi;
- Sul piano dei nomi, attraverso il *Name Affinity Coefficient*, calcolato a partire dalle relazioni che legano coppie di classi.

La combinazione di questi due coefficienti produce il *Global Affinity Coefficient*, che rappresenta il vero indicatore di similarità fra due classi.

Per effettuare il calcolo di affinità, il Thesaurus viene organizzato in una struttura simile alle Associative Networks [46], dove i nodi (ciascuno dei quali rappresenta genericamente un termine, sia esso il nome di una classe o il nome di un attributo) sono uniti attraverso relazioni terminologiche. A loro volta, tutte le relazioni presenti in questa rete sono percorribili in entrambi i sensi (dunque anche le BT e NT): due termini sono quindi affini se esiste un percorso che li unisce, formato da qualsivoglia relazioni. In figura 3.3 viene mostrato lo schema relativo all'esempio ospedaliero. Per dare una valutazione numerica della affinità tra due termini, a ogni tipo di relazione viene associato un peso (denominato *strength* e denotato da $\sigma_{\Re}$), che sarà tanto maggiore quanto più questo tipo di relazione contribuisce a legare due termini (sarà quindi: $\sigma_{syn} \geq \sigma_{bt/nt} \geq \sigma_{rt}$).

In questa sezione si userà $\sigma_{ij_{\Re}}$ per denotare il peso della relazione terminologica $\Re$ definita tra i termini $t_i$ e $t_j$. Nel nostro esempio, e nelle sperimentazioni precedentemente realizzate presso l'Università di Milano, si è adottato:
$\sigma_{syn} = 1$;
$\sigma_{bt} = \sigma_{nt} = 0.8$;
$\sigma_{rt} = 0.5$.

**Definizione 3 (Funzione di Affinità)** Presi due termini, $t_i$ e $t_j$, possono essere presenti nel grafo zero o più cammini che li uniscono, formati da relazioni. Ad ognuno di questi cammini corrisponde naturalmente un valore, dato dal prodotto dei pesi delle relazioni in esso coinvolte. La Funzione di Affinità $A_{thes}(t_i, t_j)$ tra due termini, $t_i$ e $t_j$, restituisce il valore maggiore tra questi, corrispondente al cammino più *stringente*, che unisce questi termini (che non sempre coincide col cammino più breve), definito come segue:

$$A_{thes}(t_i, t_j) = \begin{cases} 1 & \text{se } t_i = t_j \\ \sigma_{i1_{\Re}} \cdot \sigma_{12_{\Re}} \cdot \ldots \cdot \sigma_{(k-1)j_{\Re}} & \text{se } t_i \rightarrow^k t_j \\ 0 & \text{in tutti gli altri casi} \end{cases}$$

dove la notazione $t_i \rightarrow^k t_j$ denota appunto il più *stringente* tra questi cammini di lunghezza $k$, con $k \geq 1$, tra $t_i$ e $t_j$ nel grafo.

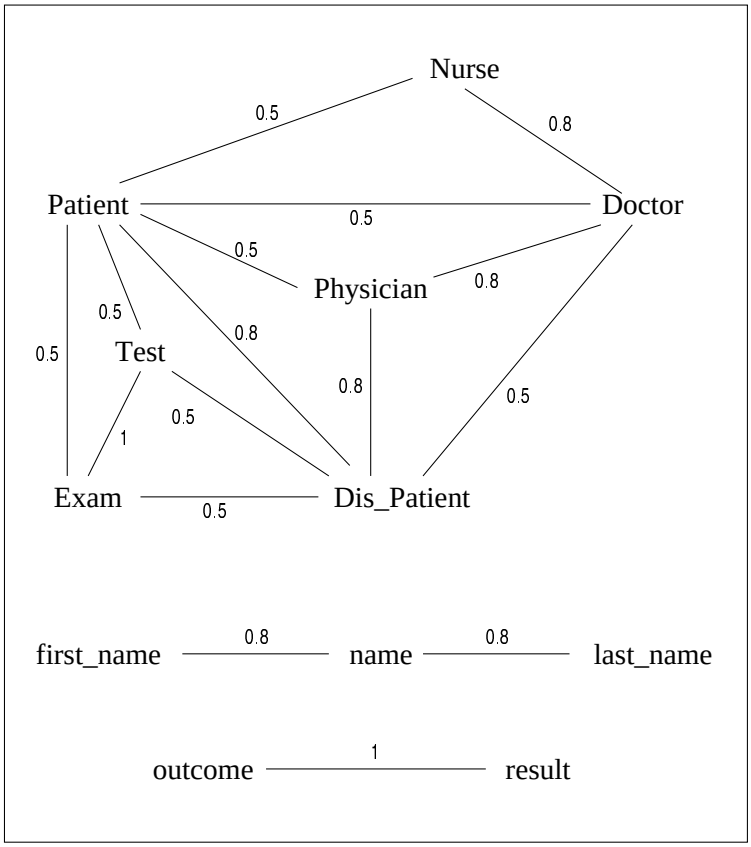



Figura 3.3: Grafo non orientato delle affinità terminologiche

Il livello di Affinità tra termini dipende quindi dalla lunghezza del cammino che li unisce, ma pure dal tipo delle relazioni coinvolte in questo cammino (e quindi dal loro peso). Per ogni coppia di termini, sarà necessariamente $A_{thes} \in [0, 1]$. La Affinità tra due temini sarà 0 se non esiste alcun cammino che li unisca, 1 se i due termini coincidono.

**Nell'esempio** Si consideri il Thesaurus illustrato in Figura 3.3. Esistono diversi cammini tra le classi `Physician` e `Nurse`; certamente il più stringente è

$\texttt{Physician} \rightarrow^{nt} \texttt{Doctor} \rightarrow^{bt} \texttt{Nurse}$

dal quale si deduce che

$A_{thes}(\texttt{Physician}, \texttt{Nurse}) = 0.8 \cdot 0.8 = 0.64.$

**Definizione 4 (Termini Affini)** Due termini $t_i, t_j$ si dicono *affini*, e si denotano

con $t_i \sim t_j$, se la loro Funzione dei Affinità restituisce un valore maggiore o uguale ad un predefinito valore di soglia $\alpha > 0$, cioè:

$$t_i \sim t_j \Longleftrightarrow A_{thes}(t_i, t_j) \geq \alpha$$

Per esempio, si supponga $\alpha = 0.3$.
In questo caso, dal momento che $A_{thes}(\texttt{Physician}, \texttt{Nurse}) = 0.64$, possiamo concludere che $\texttt{Physician} \sim \texttt{Nurse}$.

### 3.2.1 Coefficienti di Affinità

In questo paragrafo, vengono date le definizioni dei parametri attraverso i quali si determina l'affinità tra due classi, cioè i coefficienti *Name Affinity Coefficient*, *Structural Affinity Coefficient* e *Global Affinity Coefficient* facendo riferimento a due classi $\text{ODL}_{I^3}$ $c$ e $c'$ appartenenti rispettivamente alle sorgenti $S$ e $S'$.

**Definizione 5 (Name Affinity Coefficient)** Misura la affinità di due classi calcolata rispetto ai loro nomi. Il *Name Affinity Coefficient* di due classi $c$ e $c'$ denotato da $NA(c,c')$, è la misura della affinità tra i loro nomi, $n_c$ e $n_{c'}$, calcolata come segue:

$$NA(c, c') = \begin{cases} A_{thes}(n_c, n_{c'}) & \text{se } n_c \sim n_{c'} \\ 0 & \text{in tutti gli altri casi} \end{cases}$$

In sostanza il *Name Affinity Coefficient* tra due classi coincide esattamente con la Funzione di affinità solo se il suo valore supera la soglia $\alpha$, altrimenti è nullo.

**Nell'esempio** Si considerino le classi `Physician` e `Nurse`. In virtù di quanto calcolato in precedenza si avrà che $NA(\texttt{Physician}, \texttt{Nurse}) = 0.64$
Se si prendono in considerazione le classi `Exam` e `Doctor`, si nota che
$A_{thes}(\texttt{Exam}, \texttt{Doctor}) = 0.25 \leq \alpha \Rightarrow NA(\texttt{Exam}, \texttt{Doctor}) = 0$

**Definizione 6 (Structural Affinity coefficient)** Lo Structural Affinity Coefficient di due classi $c$ e $c'$, scritto $SA(c, c')$, è la misura dell'affinità dei loro schemi di attributi, calcolata come segue:

$$SA(c,c') = \frac{2\cdot \mid \{(a_t,a_q) \mid a_t \in A(c), a_q \in A(c'), n_t \sim n_q\} \mid}{\mid A(c) \mid + \mid A(c') \mid} \cdot F_c$$

Lo Structural Affinity Coefficient tra due classi è valutato utilizzando la funzione di Dice, e raffinato da un fattore di controllo $F_c$, e restituisce un valore compreso nell'intervallo [0,1] proporzionale al numero di attributi *affini* tra le classi considerate. Il numeratore della frazione rappresenta il numero di coppie di attributi i cui nomi sono *termini affini*, mentre a denominatore compare la somma delle cardinalità degli schemi di attributi delle classi.
Rispetto alla definizione originaria di $SA$ precedentemente sviluppata (si veda [20]), è stato aggiunto un fattore di controllo $F_c$, definito come segue:

$$F_c = \frac{|\{x \in C |\ flag(x)=1\ \}|}{|C|}$$

$$\text{C} = \{(a_t,a_q) \mid a_t \in A(c), a_q \in A(c'), \langle n_t \text{ SYN } n_q\rangle \vee \langle n_t \text{ BT } n_q\rangle \vee \langle n_t \text{ NT } n_q\rangle\}$$

dove C è l'insieme delle coppie di attributi validabili (ovvero delle coppie coinvolte in relazioni dirette, che possono essere validate attraverso un controllo sui domini) e $flag(x) = 1$ sta per un risultato positivo della suddetta validazione.

Il termine $F_c$ realizza un controllo sui domini degli attributi coinvolti nella relazione da esaminare (il controllo è quello esposto nel paragrafo 3.1.4), permettendo quindi di non limitare la computazione del coefficiente strutturale alla sola analisi dei nomi che identificano gli attributi (analisi terminologica), ma di estenderla alla considerazione dei domini che caratterizzano questi attributi. In pratica, una relazione che coinvolge attributi viene pesata in modo maggiore o minore nel calcolo del coefficiente a seconda che questa relazione trovi o meno riscontro anche nei tipi dei domini, e non solo nei nomi degli attributi. Il termine $F_c$ va quindi a rifinire il coefficiente $SA$, moltiplicando la prima parte di questo per un termine compreso tra 0 e 1: in particolare, $F_c$ è il rapporto tra numero di relazioni validate memorizzate nel Thesaurus tra attributi delle due classi, e numero totale di queste relazioni.
In questo modo, maggiore sarà il numero di attributi affini tra le due classi, e maggiore il numero di controlli positivi, più prossimo all'unità risulterà il valore dello *Structural Affinity Coefficient*.

**Nell'esempio** Si considerino gli schemi delle sorgenti riportati nelle figure 2.6 e 2.8. Per quanto riguarda le classi `Physician` e `Nurse` abbiamo:

- 2 coppie di attributi affini:
  - `CD.Physician.name` $\sim$ `CD.Nurse.name`
  - `CD.Physician.address` $\sim$ `CD.Nurse.address`
- 8 attributi negli schemi (4 in uno e 4 nell'altro)
- 2 relazioni dirette fra i nomi degli attributi:
  - `CD.Physician.name` SYN `CD.Nurse.name`
  - `CD.Physician.address` SYN `CD.Nurse.address`
- entrambe le relazioni del punto precedente sono *valide*.

Di conseguenza il valore dello *Structural Affinity Coefficient* sarà calcolato nel seguente modo:

$SA(\texttt{Physician}, \texttt{Nurse}) = \frac{2 \cdot 2}{4+4} \cdot \frac{2}{2} = 0.5.$

**Definizione 7 (Global Affinity Coefficient)** Il Global Affinity Coefficient di due classi $c$ e $c'$, denotato da $GA(c, c')$, è la misura della loro affinità calcolata come la somma pesata di *Name Affinity Coefficient* e *Structural Affinity Coefficient*:

$$GA(c, c') = \begin{cases} w_{NA} \cdot NA(c, c') + w_{SA} \cdot SA(c, c') & \text{se } NA(c, c') \neq 0 \\ 0 & \text{in tutti gli altri casi} \end{cases}$$

dove i pesi $w_{NA}$ e $w_{SA}$, con $w_{NA}, w_{SA} \in [0, 1]$ e $w_{NA} + w_{SA} = 1$, sono stati introdotti per dare al progettista la possibilità di variare caso per caso l'importanza dovuta ad ognuno dei due coefficienti rispetto all'altro. Nel nostro esempio di riferimento, abbiamo considerato ugualmente rilevanti ai fini dell'integrazione i due coefficienti, ponendo quindi $w_{NA} = w_{SA} = 0.5$.
Durante il calcolo di questo coefficiente globale, è comunque data implicitamente una maggiore rilevanza al *Name Affinity coefficient*, e quindi ai nomi delle classi stesse: infatti condizione necessaria affinché esista un $GA$ non nullo tra due classi, è che le stesse siano *affini*, pertanto per classi i cui nomi non hanno nulla in comune ($NA = 0$) non è neppure valutata la affinità rispetto ai loro attributi, e conseguentemente il loro $GA$ risulterà a sua volta nullo.

**Nell'esempio** Partendo dai coefficienti riportati negli Esempi precedenti, il Global Affinity Coefficient delle classi `Physician` e `Nurse` è calcolato nel seguente modo:

$$GA(\texttt{Physician}, \texttt{Nurse}) = 0.5 \cdot 0.64 + 0.5 \cdot 0.5 = 0.57$$

## 3.3 Generazione dei Cluster

Per l'identificazione degli insiemi di classi affini negli schemi considerati sono utilizzate, all'interno del mediatore, tecniche di clustering, attraverso le quali le classi sono automaticamente classificate in gruppi caratterizzati da differenti livelli di affinità, formando un albero.

La procedura di clustering utilizza una matrice $M$ di rango $r$, con $r$ uguale al numero totale di classi $\text{ODL}_{I^3}$ che devono essere analizzate. In ogni casella $M[h,k]$ della matrice è rappresentato il coefficiente globale $GA()$ riferito alle classi $c_h$ e $c_k$.
La matrice $M$ ha le seguenti proprietà:

- $\forall h,k < r$, con $h \neq k$, $M[h,k] = M[k,h] \iff$ la matrice è simmetrica. Infatti, poiché le relazioni sono simmetriche (e quindi il grafo non è orientato) il *Name Affinity Coefficient* è a sua volta simmetrico; lo stesso si può dire per lo *Structural Affinity Coefficient* osservando la formula che lo genera. Di conseguenza anche $GA()$ gode della proprietà di simmetria.
- $\forall h < r$, $M[h,h]$ non ha valore: la diagonale principale, cioè, non deve essere presa in considerazione poiché non ha senso mettere a confronto una classe locale con se stessa

La procedura di clustering è iterativa e comincia allocando per ogni classe un singolo cluster: successivamente, ad ogni iterazione, i due cluster tra i quali sussiste il $GA()$ di valore massimo nella matrice $M$ sono uniti. $M$ è così aggiornata dopo ogni operazione di fusione tra cluster, cancellando le righe e le colonne corrispondenti ai cluster unificati, e inserendo una nuova riga ed una nuova colonna che rappresenti il nuovo cluster determinato. Vengono quindi calcolati i coefficienti $GA()$ tra questo cluster aggiunto e tutti quelli già presenti nella matrice: in particolare, viene mantenuto il valore $GA()$ massimo tra i due che erano stati già calcolati tra i cluster rimossi ed il corrispondente cluster col quale si vuole determinare il nuovo valore del coefficiente globale. La procedura termina quando tutte le classi appartengono ad un unico cluster.

L'output di questa fase non è comunque il cluster finale, contenente tutte le classi: ben più importante è l'albero che si è definito attraverso questa procedura

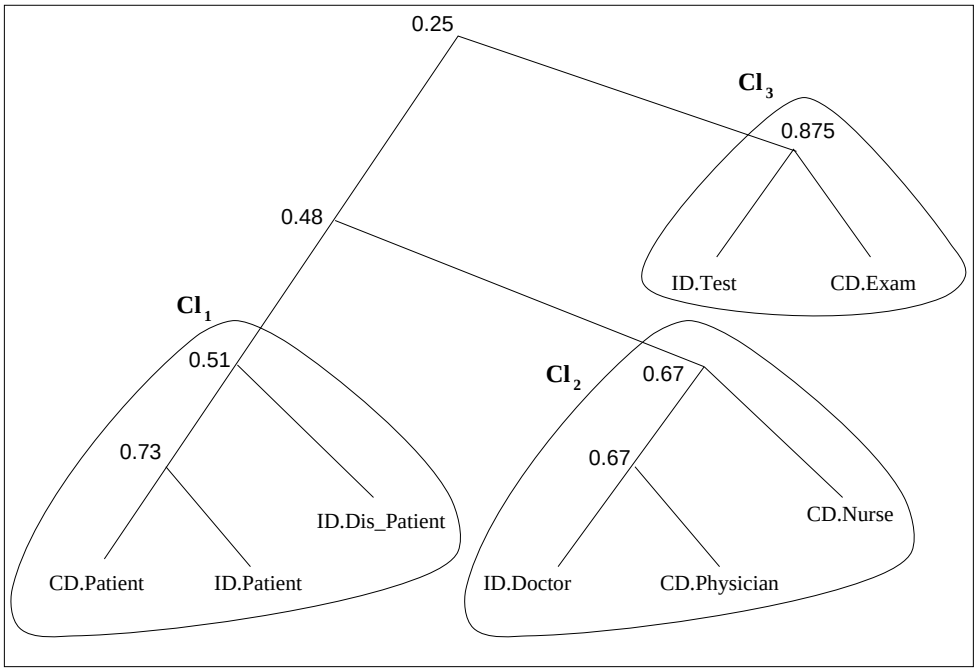



Figura 3.4: Albero di affinità

di clustering, riportato, sempre relativamente all'esempio ospedaliero, in Figura 3.4. In questo albero, le foglie rappresentano tutte le classi locali rappresentate: foglie contigue sono caratterizzate da alta affinità, foglie tra loro molto lontane rappresenteranno invece concetti differenti.
Ogni nodo rappresenta un livello di clusterizzazione, ed ha associato il coefficiente di affinità tra i due sottoalberi (cluster) che unisce. In questo modo, scegliendo un valore di soglia di riferimento, si possono formare non un unico, bensì un insieme di cluster, all'interno dei quali sono raggruppate tutte le classi tra le quali esiste una affinità (rappresentata dal valore di $GA$) maggiore del valore soglia predefinito. Come mostrato in figura, abbiamo scelto come soglia il valore 0.5: tutte le classi di partenza delle sorgenti locali $ID$ e $CD$ sono state raggruppate iterativamente (attraverso la procedura esposta sopra) in tre cluster finali $Cl_1$, $Cl_2$ e $Cl_3$.

## 3.4 Costruzione delle classi globali

L'ultimo passo che porta alla generazione dello schema globale, è quello che associa una classe globale ad ogni cluster determinato nella fase precedente; lo

schema globale, visibile dall'utente finale e sul quale l'utente stesso porrà le interrogazioni, è formato dall'insieme di tutte queste classi.

Il processo di trasformazione di ogni cluster (definito come gruppo di classi locali ritenute affini) in classe globale, è articolato nelle seguenti fasi:

- Unione degli attributi di tutte le classi appartenenti al cluster;
- Fusione degli attributi "simili".

Questa cosiddetta "Unione ragionata" degli attributi locali, è un'operazione importante e delicata: importante perché attraverso lo schema di attributi visibile all'utente, si deve dare a quest'ultimo la possibilità di porre query semplici ma espressive; delicata perché non è immediato stabilire quali attributi debbano essere collassati in uno solo.

Le uniche informazioni che il sistema ha in merito ai rapporti tra attributi appartenenti a diverse classi (o relazioni), sono quelle nemorizzate nel Thesaurus comune (vedi sezione 3.1) generato prima della fase di creazione dei cluster.

### 3.4.1 Unione degli attributi

Ogni cluster generato nella fase precedente è semplicemente una lista delle classi locali che ne fanno parte, cioè che, attraverso il calcolo di affinità illustrato nella sezione precedente, sono state ritenute *simili*. Questa fase consiste semplicemente nella raccolta e concatenamento di tutti gli attributi appartenenti alle suddette classi.

$$A(Cl_i) = \bigcup^{j} A(c_j), \forall c_j \in Cl_i$$

**Nell'esempio** $A(Cl_1) =$ `code, first_name, last_name, address, test, doctor_id, code, date, note, name, address, exam,room, bed, therapy, physician`.

### 3.4.2 Fusione degli attributi

Il sistema è in grado di conoscere le affinità degli attributi in base alle relazioni che li legano tra loro, fornite da WordNet e dal designer, e successivamente filtrate dal processo di validazione basato sul confronto dei domini, spiegato nel paragrafo 3.1.4.

### Attributi relativi a relazioni validate

Partendo proprio dalle relazioni validate, si identificano tutte quelle che mettono a confronto due attributi appartenenti al medesimo Cluster; il processo di fusione opera nel modo seguente:

- Tutti gli attributi sinonimi (o omonimi) vengono collassati in uno solo il cui nome potrebbe essere quello di uno solo dei partecipanti o, meglio, potrebbe essere dato dal concatenamento dei singoli nomi, lasciando successivamente al progettista il compito di trovare un nome conciso ed appropriato.
  Ad esempio nel Cluster $Cl_3$ gli attributi `result` e `outcome` (appartenenti rispettivamente alle classi `ID.Test` e `CD.Exam`) legati da sinonimia, sono fusi in `result_outcome`

- Ogni gerarchia di termini legati da relazioni di specializzazione (BT e NT) viene sostituita da un unico attributo, il cui nome coincide col termine più generale della gerarchia stessa, quello cioè che ne sta a capo e che quindi la rappresenta.
  Nell'esempio abbiamo, riguardanti i cluster $Cl_1$ e $Cl_2$: `name` BT `first_name` e `name` BT `last_name`, da cui si ricava l'unico attributo `name`.

### Attributi relativi a relazioni non validate

La fusione di attributi coinvolti in relazioni validate è un'operazione automatica. Lo stesso non si può dire nel caso di relazioni non validate, per le quali il problema è più complesso.
Da un lato abbiamo un'informazione fornita dal progettista o da un dizionario lessicale (la relazione terminologica) secondo cui i concetti espressi dai nomi degli attributi sono tra loro correlati, dall'altro l'analisi sui domini mostra che gli attributi in questione sono incompatibili.
Per capire meglio il problema, prendiamo in esame l'esempio di riferimento: una delle relazioni non validate è la seguente:

`CD.Patient.physician` BT `ID.Patient.doctor_id`

Questa relazione non è valida secondo l'analisi di ODB-Tools perché l'attributo `doctor_id` appartiene all'insieme degli interi, mentre `physician` è una collezione di OID che realizza aggregazione tra le classi `CD.Patient` e `CD.Physician`. Tuttavia un'analisi attenta sui sorgenti, mostra che `doctor_id`, all'interno della relazione `ID.Patient` (ID è infatti un sorgente

relazionale) è *foreign key* e realizza anch'esso una aggregazione fra le relazioni `ID.Patient` e `ID.Doctor`.

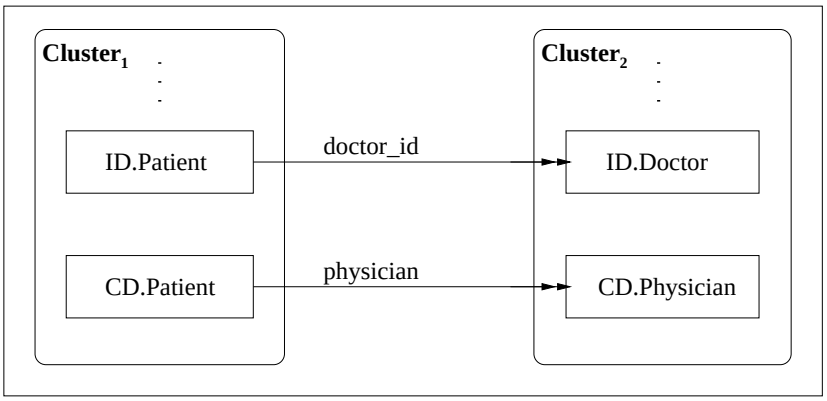



Figura 3.5: Fusione di attributi contenuti in relazioni non valide

Dalla figura 3.5, che mostra graficamente il significato complesso dei due attributi, si nota che le classi puntate appartengono entrambe allo stesso cluster $Cl_2$. In tal caso è opportuno che i due attributi vengano fusi in uno solo, il cui nome coinciderà con il termine più generale dei due:

$$\texttt{physician, doctor\_id} \longrightarrow \texttt{physician}$$

**Esempio** Supponiamo di voler formulare la seguente query: *Seleziona i nomi di tutti i pazienti assistiti dal dottor 'Mario Rossi'*; alla luce della fusione appena fatta, l'utente effettuerà la seguente interrogazione:

```
select  Y.name
from    ⟨nome della classe globale associata a Cl1⟩ as Y
where   exists X in Y.physician :  X.name = 'Mario Rossi'
```

Questa query sarà tradotta dal query Manager del mediatore nelle due seguenti interrogazioni rivolte rispettivamente ai sorgenti CD e ID:

```
select Y.name
from   Patient as Y
where  exists X in Y.physician : X.name = 'Mario Rossi'

select name
from   Patient as P, Doctor as D
```

```
where  P.doctor_id = D.id
and    D.first_name = 'Mario'
and    D.last_name = 'Rossi'
```

Nel caso del sorgente relazionale, ovviamente, è stato necessario tradurre il join implicito espresso dalla *dot notation* in join esplicito.

Se `physician` e `doctor_id` non fossero stati integrati nell'unico attributo `physician`, non sarebbe stato possibile ottenere le informazioni richieste, se non con una query ben più elaborata.

```
select Y.name
from   Cl1 as Y, Cl2 as Z
where  exists X in Y.physician : X.name = 'Mario Rossi'
or     (    Y.doctor_id = Z.id
        and Z.first_name = 'Mario'
        and Z.last_name = 'Rossi' )
```

Non in tutti i casi ha senso fondere insieme due o più attributi complessi che mappano la stessa classe globale; le condizioni affinché sia possibile raggruppare attributi appartenenti a relazioni non valide sono le seguenti:

- Devono rappresentare entrambi una gerarchia di aggregazione (collezioni di attributi complessi o foreign key)
- Le rispettive classi locali mappate devono appartenere allo stesso cluster; stessa condizione deve valere anche per le classi di appartenenza, ma questo è sempre vero perché le relazioni che accostano attributi appartenenti a cluster differenti non vengono prese in considerazione.
- Devono avere lo stesso significato semantico: questa condizione è garantita se tra i due attributi in questione esiste una relazione di tipo SYN, BT o NT

## 3.5 Revisione dello schema globale

Dopo aver analizzato e manipolato i cluster fino a pervenire alla definizione dello schema di attributi di ognuno, subentra una fase in cui è necessaria l'interazione con il progettista, il quale deve operare alcune scelte al fine di pervenire ad uno schema chiaro ed espressivo. In particolare occorre:

- dare un nome alle classi globali
- definire il mapping fra attributi globali e locali

- Assegnare valori di default
- Aggiungere, eventualmente, nuovi attributi globali

### 3.5.1 Nome delle classi globali

Il sistema è in grado, sfruttando le relazioni terminologiche in suo possesso, di proporre dei nomi candidati, tuttavia il nome più appropriato associato ad ogni cluster, che ne sintetizzi il concetto rappresentato può fornirlo solo il progettista.
Nel nostro esempio, un nome appropriato per la classe nata da $Cl_1$ è `Hospital_Patient`, la classe relativa a $Cl_2$ si chiamerà `Medical_Staff`, mentre quella associata a $Cl_3$ potrebbe chiamarsi `Exam`.

### 3.5.2 Mapping fra attributi globali e attributi locali

Come spiegato al paragrafo 3.4.2, la definizione dello schema di una classe globale si ottiene, per certi attributi, attraverso la loro manipolazione.
Per consentire al gestore delle interrogazioni di tradurre correttamente le query che l'utente rivolge allo schema integrato, ma che devono essere rigirate ai singoli schemi locali, è necessario conoscere l'origine di ogni attributo globale, in particolare da quali attributi locali è generato.
Preso un attributo globale, i casi possibili di mapping sono i seguenti:

- Si ottiene da un unico attributo locale: in tal caso il sistema, in base alle informazioni in suo possesso è in grado di realizzare automaticamente il mapping; il nome coincide con quello locale.
  Ad esempio `Medical_Staff.phone` mappa in `ID.Doctor.phone`

- Si ottiene dalla fusione di più attributi, tutti appartenenti a classi locali diverse: come nel caso precedente il mapping si ottiene in modo automatico; il progettista può intervenire per assegnare alla grandezza globale un nome appropriato.
  È il caso di `Hospital_Patient.physician` che mappa su `ID.Patient.doctor_id` e su `CD.Patient.physician`

- Si ottiene attraverso la fusione di più attributi, alcuni dei quali appartengono alla stessa classe locale. In tal caso ci sono due tipi di corrispondenza, e il designer deve scegliere quale è più corretta:

  – corrispondenza in **and**
  L'attributo globale è ottenuto dal concatenamento degli attributi

locali.
Nel nostro esempio è il caso di `Hospital_Patient.name` che mappa in `ID.Patient.first_name` e `ID.Patient.last_name`, così come `Medical_Staff.name` mappa in `ID.Doctor.first_name` e `ID.Doctor.last_name`
Nelle query che coinvolgono attributi con corrispondenza in *and*, a seconda che l'attributo compaia in *select* o nella clausola di *where* la traduzione avviene come mostrato nei prossimi due esempi.

```
select name
from   Hospital_Patient
```

Il query manager la tradurrà per il sorgente ID nel modo seguente:

```
select first_name, last_name
from   Patient
```

Vediamo il caso in cui `name` compare nella clausola di *where*

```
select phone
from   Medical_Staff
where  name = 'Mario Rossi'
```

Per il sorgente ID sarà formulata come:

```
select phone
from   Doctor
where  first_name = 'Mario'
and    last_name = 'Rossi'
```

– corrispondenza in **union**
l'attributo globale corrisponde ad uno degli attributi locali. In questo caso viene specificato un terzo parametro chiamato *tag attribute*, che funge da *switch*; in base cioè al valore assunto da tale parametro il query manager individua quale attributo locale chiamare in causa.
**Esempio:** Supponiamo di avere la classe globale `Automobile` comprendente la classe locale `S.Car`. Quest'ultima ha due attributi, `S.Car.it_price` e `S.Car.us_price` specificanti il prezzo in lire e dollari rispettivamente, e un terzo attributo `S.Car.country` indicante la nazione dove il prodotto viene commercializzato. Durante il processo di integrazione si è deciso di unificare i due attributi locali nell'unico globale `Automobile.price`. In questa situazione il mapping è di tipo *union* e il *tag attribute* è `S.Car.country`.
Per sapere in base a quale valore di `S.Car.country` l'attributo globale assume il valore di uno o l'altro attributo locale, deve essere

definita una apposita regola:

```
rule Rule1 { case of S.Car.country:
             'Italy' : S.Car.it_price;
             'USA'   : S.Car.us_price;  }
```

In questo modo, data la query posta allo schema globale che chiede il prezzo di ogni automobile:

```
select price
from   Automobile
```

il Query manager formulerà le due seguenti interrogazioni al sorgente S:

```
select it_price
from   Car
where  country = 'Italy'

select us_price
from   Car
where  country = 'USA'
```

la risposta sarà ottenuta unendo le istanze restituite dalle due query

### 3.5.3 Valori di default

Nelle classi locali dove un attributo globale non ha mapping, a volte può essere necessario aggiungere un valore di *default* per quell'attributo. In alcuni casi il progettista è a conoscenza di tale informazione, in altri casi l'informazione stessa è già presente negli schemi locali come metadato e, se non si opera in questo senso, si rischia di perderla nello schema globale, riducendo la capacità di ottenere risposte corrette alle interrogazioni.

### 3.5.4 Nuovi attributi globali

Può capitare, soprattutto quando un cluster contiene molte classi locali, che alcune delle informazioni ricavabili dalla struttura dei singoli schemi locali possano non essere reperibili nello schema globale; in altri termini, potrebbe

risultare impossibile isolare un concetto tra diverse entità contenute all'interno di una stessa classe globale.
Infatti, per come è costruito lo schema, ogni classe globale ottenuta è l'unione intensionale ed estensionale di alcune classi locali, invisibili individualmente a chi usufruisce dello schema integrato. L'aggiunta di uno o più attributi globali da parte del designer è un'opportunità per ovviare a questo problema e per arricchire ulteriormente il contenuto informativo dello schema integrato.
Nel caso dell'esempio ospedaliero, la conoscenza del reparto dove è ricoverato un paziente o dove lavora un medico, è nota a livello dei singoli sorgenti, perché contenuta, sottoforma di metadato, nel nome degli stessi sorgenti. Per poter distinguere i reparti anche dal punto di vista integrato è necessario aggiungere l'attributo globale `dept` nelle classi `Hospital_Patient` e `Medical_Staff`. Un attributo aggiunto non può avere mapping su attributi locali, però può assumere valori di default in corrispondenza di determinate classi locali. Nel nostro caso avremo:
`Hospital_Patient.dept = 'Cardiology'` associato a `CD.Patient`
`Hospital_Patient.dept = 'Intensive Care'` per `ID.Patient`
`Hospital_Patient.dept = 'Intensive Care'` per `ID.Dis_Patient`
Alla luce di questa aggiunta sarà possibile, da parte dell'utente, recuperare *'la lista dei pazienti ricoverati in cardiologia'*

```
select name
from   Hospital_Patient
where  dept = 'Cardiology'
```

tradotta, di fatto, in:

```
select name
from   CD.Patient
```

## 3.6 Query Processing e Reformulation

La strategia descritta in questo capitolo, utilizzata da **Momis** per la generazione dello schema globale prevede che quest'ultimo rappresenti semplicemente una vista, un contenitore virtuale, mentre in realtà i dati sono fisicamente allocati in diversi database sorgenti.
Come già visto nella sezione precedente, in fase di interrogazione è indispensabile, da parte del query manager, la conoscenza delle connessioni tra tutte le entità globali e tutte quelle locali. In particolare serve una struttura, attraverso

la cui consultazione sia possibile tradurre qualsiasi query posta dall'utente, e quindi reperire ogni informazione richiesta. Tale struttura è data dalla cosiddetta **Mapping Table**, un insieme di tabelle (una per ogni classe globale) contenenti le informazioni di mapping descritte nella sezione 3.5.

| **Hospital_Patient** | **code** | **name** | **exam** | **physician** | **dept** | ... |
|---|---|---|---|---|---|---|
| CD.Patient | *null* | name | exam | physician | 'Cardiology' | ... |
| ID.Patient | code | first_name *and* last_name | test | doctor_id | 'Intensive Care' | ... |
| ID.Dis_Patient | code | *null* | *null* | *null* | 'Intensive Care' | ... |

Figura 3.6: Porzione di Mapping table della classe globale `Hospital_Patient`

In figura 3.6, è mostrata parte della Mapping Table relativa alla classe globale `Hospital_Patient` generata a partire dal cluster $Cl_1$. La cella superiore sinistra contiene il nome della classe: le righe contengono informazioni relative alle classi locali appartenenti al cluster $Cl_1$ (la cella più a sinistra di ogni riga è il nome); le colonne contengono i dati relativi agli attributi globali della classe in questione (i cui nomi sono in alto). Pertanto ogni cella, individuata dall'incrocio fra una riga e una colonna, contiene (se esiste) il tipo di mapping di un attributo globale nella classe locale.

Riferendoci all'esempio di figura, l'attributo globale `name` mappa nella classe locale `CD.Patient` come `name`, in `ID.Patient` come `first_name` *and* `last_name`, mentre non è presente corrispondenza con `ID.Dis_Patient`, perciò la relativa cella è denotata dall'indicazione *null*;

### 3.6.1 Query Reformulation

In questa sezione viene descritto il processo che, sfruttando le informazioni memorizzate nelle mapping table e le regole di integrità, realizza la Query Reformulation. Scopo di questo processo, che verrà attivato dal Query Manager di **MOMIS** [47] ogniqualvolta l'utente pone una interrogazione sullo schema globale (esprimendola dunque in termini di classi e attributi globali), è arrivare alla definizione automatica delle interrogazioni che devono essere inviate alle diverse fonti di informazione per dare risposta alla query originale.

Le fasi che caratterizzano la Query Reformulation sono le seguenti:

1. *ottimizzazione semantica*: sfruttando le informazioni semantiche presenti a livello di schema globale, ed eventuali regole di integrità definite dal

progettista (sempre sullo schema globale), è realizzata una ottimizzazione semantica dell'interrogazione posta dall'utente;

2. *formulazione del query plan*: basandosi sulla mapping table, e su un algoritmo di eliminazione delle sorgenti *"inutili"*, il sistema genera automaticamente un insieme di query da spedire alle sorgenti locali;

3. *ottimizzazione basata sulla conoscenza estensionale*: nel caso in cui siano disponibili delle regole definite tra le estensioni delle classi locali, queste informazioni sono sfruttate dal sistema per limitare ulteriormente il numero di accessi alle sorgenti da interrogare, mantenendo inalterato l'insieme delle risposte alla query posta dall'utente.

**Ottimizzazione Semantica**

Il Query Manager opera sull'interrogazione posta dall'utente sfruttando tecniche di ottimizzazione semantica supportate da ODB-Tools [48, 17, 18, 19]. In particolare, sulla base di regole di integrità definite sullo schema globale del mediatore, la query viene arricchita con nuovi predicati nella clausola di *where*, implicati da quelli inizialmente presenti. Al termine di questo processo la nuova query incorpora ogni possibile restrizione logica non presente in quella originaria, col risultato che, in generale, il costo in termini di tempo che questa modifica comporta è più che compensato dal guadagno dovuto al minor costo di reperimento delle risposte all'interrogazione.

Supponiamo, ad esempio, che nello schema globale `Hospital` sia definita una regola secondo cui tutti i pazienti col cuore a rischio si trovino nel reparto 'Cardiology'. Formalmente la regola si esprime nel seguente modo:

```
rule R1
   forall X in Hospital_Patient:
      (X.Exam.result = 'Hearth risk')
      then X.dept = 'Cardiology' ;
```

Supponiamo che al mediatore sia formulata la sequente interrogazione:

```
Q1:
select name
from   Hospital_Patient
where  exam.result = 'Heart risk'
```

Il Query Manager, utilizzando il query optimizer di ODB-Tools, esegue l'espansione semantica della query sfruttando la conoscenza aggiunta dalla rule `R1`. La query risultante, equivalente a quella posta dall'utente, è la seguente:

```
Q1':
select name
from   Hospital_Patient
where  exam.result = 'Heart risk'
and    dept = 'Cardiology'
```

Il vantaggio dell'ottimizzazione semantica si realizza se sono presenti indici sugli attributi coinvolti nei predicati aggiunti nella clausola di *where* .

**Formulazione del Query Plan**

Una volta che il sistema MOMIS è in possesso della query globale, eventualmente ottimizzata, deve essere costruito un piano di accesso per andare a recuperare le informazioni da fornire in risposta all'utente. In questa sezione viene presentato il meccanismo attraverso il quale saranno definite le query da spedire alle sorgenti locali.

Per ciascuna sorgente informativa coinvolta, il Query Manager traduce la query ottimizzata sostituendo, con l'aiuto della mapping table, i termini globali con i corrispondenti locali. Non tutte le classi locali devono essere interrogate, in particolare si definiscono due proprietà che distinguono fra loro le classi rispetto ad una data query posta sullo schema globale:

- *utilità*: una classe locale è *non utile* rispetto ad una data query se almeno uno dei predicati booleani posti in *and* nella clausola *where* risulta sempre falso. In tal caso nessuna delle istanze della classe soddisfa le condizioni richieste, e pertanto la classe non sarà interrogata. Per individuare le classi non utili il Query Manager si basa sui valori di default presenti nella Mapping Table: se nella query è richiesto che un attributo globale abbia un determinato valore e lo stesso attributo, in corrispondenza di una classe locale assume un valore di default differente, tale classe risulta *non utile* e non sarà interrogata.
  Nel nostro esempio, la query `Q1'` contiene il predicato `dept = 'Cardiology'` che esclude le classi locali `ID.Patient` e `ID.Dis_Patient` per le quali l'attributo globale `dept` assume, di default, valore `'Intensive Care'`.
  Notare che con la query `Q1` non ottimizzata questa esclusione non sarebbe possibile, e comporterebbe accessi inutili al sorgente `ID`.

- *verificabilità*: una classe locale è *non verificabile* se almeno uno degli attributi globali presenti nella clausola *where* non è mappato nella classe stessa (valore *null* nella Mapping Table). In tal caso non è possibile verificare

quali istanze di quella classe appartengono all'insieme risposta della query, e pertanto è necessario fare una scelta:

- si può decidere di scartare le classi non verificabili: così facendo, però, non si garantisce la completezza della risposta alla query, poiché alcune fra le istanze scartate potrebbero farne parte;
- si può decidere di interrogare comunque la classe non verificabile: in tal caso non è garantita la correttezza della risposta alla query; infatti le istanze della classe in questione che fanno parte della risposta alla query non sono del tutto verificate.

Nel sistema Momis si è deciso di privilegiare la correttezza della risposta a una query, pertanto è stata fatta la scelta drastica di interrogare solo le classi locali verificabili.
È opportuno precisare che ci sono diversi gradi di verificabilità, legati al rapporto fra il numero di condizioni verificabili e il numero totale di condizioni nella query. Si potrebbe allora ipotizzare l'esistenza di un livello di *credibilità* delle risposte, fissato dall'utente, che esprima la soglia minima di verificabilità affinché una classe sia interrogata.

Tornando all'esempio, la query locale generata dal Query manager e indirizzata al sorgente `CD`, è la seguente:

```
Q1":
select name
from   Patient
where  exam.outcome = 'Heart risk'
```

Da questo esempio si nota che nell'interrogazione locale manca la condizione sul dipartimento; innanzitutto non sarebbe possibile includerla perché non esiste localmente un attributo per cui è stato imposto un valore di sefault; inoltre, come è vero che un valore di default diverso da quello richiesto dalla query consente di scartare la classe per la quale tale default è definito, allo stesso modo se su un attributo globale la classe locale assume, di default, lo stesso valore richiesto dalla query, allora quella condizione è soddisfatta per tutte le istanze della classe considerata, e pertanto la condizione nella quale è presente l'attributo globale non compare nella query locale.

Se sono definite delle regole di integrità sui singoli schemi sorgenti, allora ogni query locale che scaturisce dalla traduzione dell'interrogazione posta dall'utente, prima di essere inviata agli schemi, può, allo stesso modo di quella globale, essere ottimizzata.

Supponiamo ad esempio che sulla sorgente `CD` sia definita una rule secondo cui i pazienti col cuore a rischio siano degenti oltre la camera n. 20. Secondo il formalismo $\text{ODL}_{I^3}$ si ha:

```
rule R2
   forall X in Patient:
      (exist X.Exam.result = 'Hearth risk')
      then X.room > 20 ;
```

La query `Q1"`, prima di essere inviata al sorgente `CD` verrà ottimizzata e trasformata come segue:

```
Q1''':
select name
from   Patient
where  exam.outcome = 'Heart risk'
and    room > 20
```

Questa trasformazione è particolarmente vantaggiosa se esiste un indice sull'attributo `room`.

**Ottimizzazione basata sulla conoscenza estensionale**

Nella formulazione del query plan potrebbero essere sfruttate un insieme di informazioni estensionali, al fine di migliorare la traduzione delle query e il reperimento delle risposte alle stesse.
In fase di progettazione vengono sfruttate informazioni intensionali (semantiche o terminologiche).

Vi sono casi in cui, magari grazie a conoscenze a priori (può essere il caso di più database derivati da un'unica sorgente di partenza) o a un confronto tra i progettisti stessi delle sorgenti locali, si può supporre che sia possibile definire relazioni che intercorrono tra le estensioni delle classi che sono state integrate dal mediatore in un unico cluster.
Per questi casi, sono state definite quattro tipologie di proprietà *inter-schema* atte a esprimere mutue relazioni esistenti tra le estensioni.
In particolare, sia $EXT(c)$ l'insieme delle istanze (*estensione*) della classe $c \in Cl_i$. Esempi di proprietà interschema che possono essere definite tra due classi $c$ e $c'$ a livello estensionale sono i seguenti:

1. EQUIVALENZA: due classi $c$ e $c'$ sono equivalenti a livello estensionale, denotato da $c \equiv_{ext} c'$, se e solo se $EXT(c) \equiv EXT(c')$;

2. INCLUSIONE: una classi $c$ è contenuta a livello estensionale in $c'$, denotato da $c \subset_{ext} c'$, se e solo se $EXT(c) \subset EXT(c')$;

3. INTERSEZIONE: due classi $c$ e $c'$ sono intersecate a livello estensionale, denotato da $c \cap_{ext} c'$, se e solo se $EXT(c) \cap EXT(c') \neq \emptyset$ e $c \subset_{ext} c^{'}$ non è verificato;

4. DISGIUNZIONE: due classi $c$ e $c'$ sono disgiunte a livello estensionale, denotato da $c \emptyset_{ext} c'$, se e solo se $EXT(c) \cap EXT(c') = \emptyset$.

Le relazioni (o assiomi) estensionali vengono definite secondo la sintassi propria delle regole di integrità; a differenza delle regole utilizzate per l'ottimizzazione semantica delle interrogazioni (sia a livello globale che locale), le rule che definiscono gli assiomi estensionali sono di tipo *inter-schema* e vengono espresse in termini locali, cioè fra le estensioni di classi locali che in fase di progettazione sono stare raggruppate nello stesso cluster.
Come ampiamente spiegato in [49, 47], sfruttando questa conoscenza aggiunta fornita dal progettista, il Query Manager è in grado di ricavare delle gerarchie di ereditarietà (dette *Gerarchie Estensionali*) fra le classi appartenenti allo stesso cluster. Quando viene posta una interrogazione ad una classe globale, il mediatore, in fase di query plan, individua nella gerarchia estensionale associata alla classe globale stessa, l'insieme minimo di classi locali che possono fornire la risposta desiderata.

**Esempio 1** Supponiamo di avere due classi appartenenti a sorgenti diverse, $\texttt{S}_1\texttt{.Cl}_A$ e $\texttt{S}_2\texttt{.Cl}_B$, raggruppate nello stesso cluster. Per esse il progettista afferma equivalenza estensionale, attraverso le seguenti due regole:

```
rule RE1
          forall X in S1.Cl_A
          then X in S2.Cl_B ;

rule RE2
          forall X in S2.Cl_B
          then X in S1.Cl_A ;
```

Qualora venga posta una interrogazione alla classe globale di appartenenza, il Query Manager eviterà di tradurre la query per entrambe le classi locali, sapendo di ottenere due volte la stesse risposta: pertanto sceglierà, tra le due, la classe che ad esempio garantisce la *verificabilità*, oppure, se entrambe le classi sono in grado di verificare la clausola *where*, quella che consente un più veloce reperimento delle risposte, ad esempio perché possiede indici sugli attributi da verificare.
Questo esempio suggerisce un'ulteriore considerazione: qualora gli attributi presenti nella clausola *where* siano mappati in parte in una classe e in parte nell'altra, in modo tale che, prese singolarmente, nessuna delle due classi locali sia completamente *verificabile*, il Query Manager può interrogarle entrambe, ottenendo in

risposta, due insiemi di istanze completi ma non corretti; operando un equijoin fra questi insiemi il risultato finale è la risposta corretta. In questo caso, però, occorre avere almeno una chiave comune per le due classi che consenta di fare l'equijoin.

L'esempio dimostra come, in presenza di regole sulle estensioni, sia possibile in certi casi migliorare la correttezza delle risposte, in altri casi ottimizzare ulteriormente il piano di accesso alle sorgenti locali, con conseguente riduzione dei costi.

### 3.6.2 Unificazione dei dati

Una volta ricevuti i dati richiesti dalle sorgenti, in risposta alle varie query locali a loro spedite, si pone il problema di come riorganizzare queste informazioni per presentarle all'utente.

Posto in altri termini, il problema è il riconoscimento automatico di tutte le informazioni che appartengono ad uno stesso *oggetto globale*. Dal punto di vista teorico, la soluzione ideale sarebbe la presenza di una *chiave universale*, o di un *oid* comune, condivisa da tutte le sorgenti (o almeno da tutte le sorgenti che fanno parte del sistema) che individui univocamente al loro interno tutti gli oggetti. È però impensabile che questo sia realmente realizzabile, dal momento che, anche nel migliore dei casi, questo identificatore sarebbe comunque valido e unico solo all'interno di un contesto limitato (si può per esempio prendere per le persone il codice fiscale, che perde però significato all'esterno di una determinata nazione ... ). Un'altra soluzione potrebbe essere il riunificare le risposte ricevute non sulla base di un oid universale, ma sulla base delle chiavi locali (quando dichiarate) delle singole sorgenti, ma il problema rimane, ora in altri termini:

- supponendo che esista per esempio una chiave `nome` in tutte le tabelle interrogate, niente ci assicura che persone con nomi uguali siano effettivamente la stessa persona in contesti diversi;

- supponendo invece che come chiave interna delle tabelle si utilizzi un codice (pensiamo ad un esempio magazzino), il problema è più complicato: è molto probabile che si usino codici diversi per identificare lo stesso concetto, pur se il dominio degli attributi `codice` è uguale in tutte le sorgenti.

È quindi un problema aperto, la cui unica soluzione, fino a questo momento, sembra essere lo spostare questo tipo di decisione al progettista rimettendo a lui questa responsabilità. Congiuntamente alla definizione di proprietà interschema, in cui si definiscono regole anche di tipo estensionale per cui due sorgenti hanno gli stessi dati, o dati parzialmente duplicati tra loro, è ragionevole che sia compito del progettista indicare esplicitamente anche il campo (o l'insieme di campi) che dovrà essere utilizzato dal sistema per riunire, attraverso l'uso di *join*, le informazioni che fanno riferimento ad una unica entità.

# Capitolo 4

# Acquisizione di schemi sorgenti

In questo capitolo verrà affrontato il tema dell'acquisizione degli schemi sorgenti, che rappresenta il primo passo nel processo di definizione dello schema integrato. Le informazioni contenute in tali schemi, secondo l'architettura a tre livelli $I^3$ adottata dal progetto **Momis**, sono passate al Global Schema Builder dai cosiddetti *wrapper*, che hanno il compito di interporsi tra i diversi database sorgenti e il mediatore, di cui il GSB fa parte.
In particolare, nella fase di progettazione, è compito dei wrapper eliminare le eterogeneità legate al tipo di sorgenti e alla sintassi con cui sono definiti, traducendo la descrizione degli schemi, siano essi relazionali, ad oggetti, semistrutturati, etc ... in un unico linguaggio descrittivo comune. Nel sistema *Momis* si è scelto di descrivere i sorgenti basandosi sul paradigma a oggetti; pertando, indipendentemente dal modello originale adottato da ogni singolo database sorgente, ogni entità viene descritta dal relativo wrapper utilizzando sempre il concetto di *classe*.

## 4.1 Il linguaggio ODL

ODL (**O**bject **D**efinition **L**anguage) è il linguaggio per la specifica di schemi ad oggetti proposto dal gruppo di standardizzazione ODMG-93 [28, 29] e universalmente riconosciuto come standard. Le caratteristiche peculiari di ODL, al pari di altri linguaggi basati sul paradigma ad oggetti, possono essere così riassunte:

- definizione di tipi-classe e tipi-valore;
- distinzione fra intensione ed estensione di una classe di oggetti
- definizione di attributi semplici e complessi
- definizione di attributi atomici e collezioni (set, list, bag)

- definizione di relazioni binarie con relazioni inverse
- dichiarazione della signature dei metodi

La sintassi del linguaggio ODL è descritta in appendice B.

## 4.2 Il linguaggio ODL$_{I^3}$

Il linguaggio ODL rappresenta il punto di partenza nel progetto di integrazione; pur essendo, infatti, ben progettato per rappresentare la conoscenza relativa ad un singolo schema ad oggetti, è certamente incompleto se calato in un contesto di integrazione di basi di dati eterogenee quale è quello descritto in questa tesi.
Si è reso pertanto necessario definirne un'estensione, denominata ODL$_{I^3}$, in accordo con le raccomandazioni della proposta di standardizzazione per i linguaggi di mediazione, risultato del lavoro di workshop $I_3$ svoltosi presso l'università del Maryland.

Tuttavia, partendo da questa proposta, secondo cui i diversi sistemi di mediazione dovrebbero poter supportare sorgenti con modelli complessi (come quelli ad oggetti) e modelli più semplici (come file di strutture), si è comunque cercato di discostarsi il meno possibile dal linguaggio ODL.
Gli scopi raggiunti dall'estensione a ODL che ha portato al linguaggio ODL$_{I^3}$, sono i seguenti:

- per ogni classe, è data al wrapper la possibilità di indicare nome e tipo del sorgente di appartenenza;
- per le classi appartenenti ai sorgenti relazionali è possibile definire le chiavi candidate ed eventuali foreign key;
- attraverso l'uso del costrutto *union* ogni classe può avere più strutture dati alternative, mentre il costrutto *optional* consente di indicare la natura opzionale di un attributo. Queste caratteristiche sono in accordo con la strategia utilizzata per la descrizione di dati semistrutturati;
- il linguaggio supporta la definizione di grandezze locali e di grandezze globali;
- il linguaggio supporta la dichiarazione di regole di mapping (o *mapping rule* fra grandezze globali e grandezze locali;
- è data la possibilità di definire regole di integrità (o *if then rule*), sia sugli schemi locali, sia sullo schema globale;

- il linguaggio supporta la definizione di relazioni terminologiche di sinonimia (SYN), ipernimia (BT), iponimia (NT) e associazione (RT);
- il linguaggio può essere automaticamente tradotto nella logica descrittiva OLCD usata da ODB-Tools, e quindi utilizzarne le capacità nei controlli di consistenza e nell'ottimizzazione semantica delle interrogazioni.

Nelle appendici C e D è illustrata, rispettivamente in BNF e in forma grafica (diagramma sintattico), la sintassi del linguaggio ODL$_{I^3}$.

## 4.3 Il Parser per ODL$_{I^3}$

La fase di *parsing* consiste nell'acquisizione delle informazioni contenute in uno o più file di testo: ad esempio il parsing rappresenta la prima fase del processo di compilazione di un file sorgente; tale file deve essere scritto nel rispetto di un insieme di regole sintattiche e grammaticali note sia a chi lo produce (tipicamente una persona) sia al parser stesso, che si preoccupa di controllare che tali regole siano rispettate, generando, ove necessario, opportuni messaggi d'errore. Il software che realizza il parser deve, per quanto possibile, svolgere anche alcuni controlli semantici relativi alla consistenza delle informazioni contenute nel testo da analizzare.
L'obiettivo principale del *parser* resta ovviamente quello di definire una struttura dati in memoria centrale e di riempirla in base ai dati contenuti nei file che analizza. Tale struttura, che viene utilizzata dagli componenti del mediatore, deve avere le seguenti caratteristiche:

- deve essere rappresentativa di tutta la conoscenza che il linguaggio ODL$_{I^3}$ può esprimere;
- deve essere facile e intuitiva la sua consultazione da parte dei componenti che attingono alle informazioni memorizzate.

Nei paragrafi successivi verrà illustrata la sintassi di ODL$_{I^3}$, e la struttura dati che è progettata per contenere la descrizione degli schemi sorgenti da integrare.

## 4.4 La struttura dati

### 4.4.1 I tipi

ODL (e quindi ODL$_{I^3}$) prevede una distinzione fondamentale nei tipi; un tipo può essere:

- **Tipo valore**
- **Tipo classe**, chiamato comunemente **Classe**

I tipi valore sono propri delle variabili e degli attributi semplici, infatti, a differenza delle classi, ogni istanza di questo tipo è priva di identificatore, cioè ha come unica proprietà il suo valore.
Al contrario gli attributi complessi e gli oggetti in generale sono istanze di classi, con OID, interfaccia e comportamento La figura 4.1 mostra come questa differenza si manifesta nell'**Object Model**.

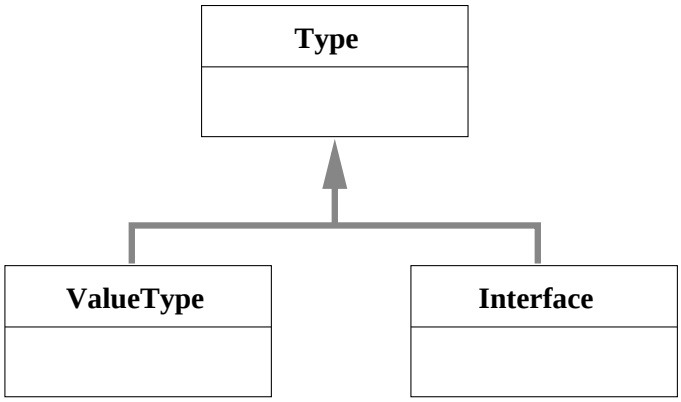



Figura 4.1: I tipi in ODL$_{I^3}$

### 4.4.2 I tipi valore

Come la figura 4.2 mostra, i *ValueType* si suddividono in:

- **SimpleType**
- **EnumType**
- **ConstrType**

#### SimpleType

I tipi semplici sono tutti i tipi atomici base, cioè i tipi predefiniti (integer, float, char, boolean, etc...), i vari tipi collezione **TemplateType** (set, list, bag, array), nonché i tipi nuovi **DefinedType**: secondo la sintassi ODL la definizione di nuovi tipi segue le stesse regole del linguaggio **ANSI C**.

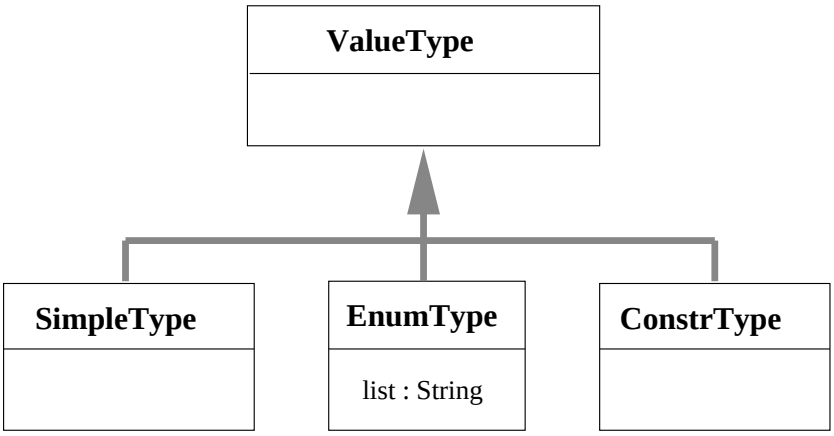



Figura 4.2: I tipi valore

Lo schema di figura 4.3, mostra che un tipo definito ha un nome, un tipo mappato (tipo-valore, non puo essere tipo-classe) e, nel caso sia un array, un insieme di valori interi che ne indicano le dimensioni; il fatto che il tipo riferito sia un generico **ValueType** e non necessariamente un **SimpleType**, consente la dichiarazione di variabili, ad esempio strutturate, in modo semplice: prima si definisce un tipo nuovo, poi le variabili secondo la sintassi dei tipi atomici. Per quanto riguarda i **TemplateType**, essi rappresentano collezioni di istanze tutte omogenee tra loro, in particolare di tipo **SimpleType**; di conseguenza non è possibile dichiarare direttamente una variabile come $set$ di $struct$, lo si può fare in due passi, attraverso la definizione di un tipo nuovo, come mostra l'esempio che segue:

```
typedef struct {
          int i,j;
          float f;
        } tipostruct;

set<tipostruct> var1,var2;
```

**ConstrType**

Fanno parte di questa classe i tipi **StructType**, **UnionType** e **EnumType**, come si nota in figura 4.4. I primi due di essi hanno un *tagName* opzionale, che consente di dichiarare più variabili dello stesso tipo complesso senza dover tutte le volte elencarne la struttura. Tuttavia il *tagName* non deve essere considerato come un nome di tipo. Infatti, al contrario del nome di un tipo definito, il *tagName* non è a

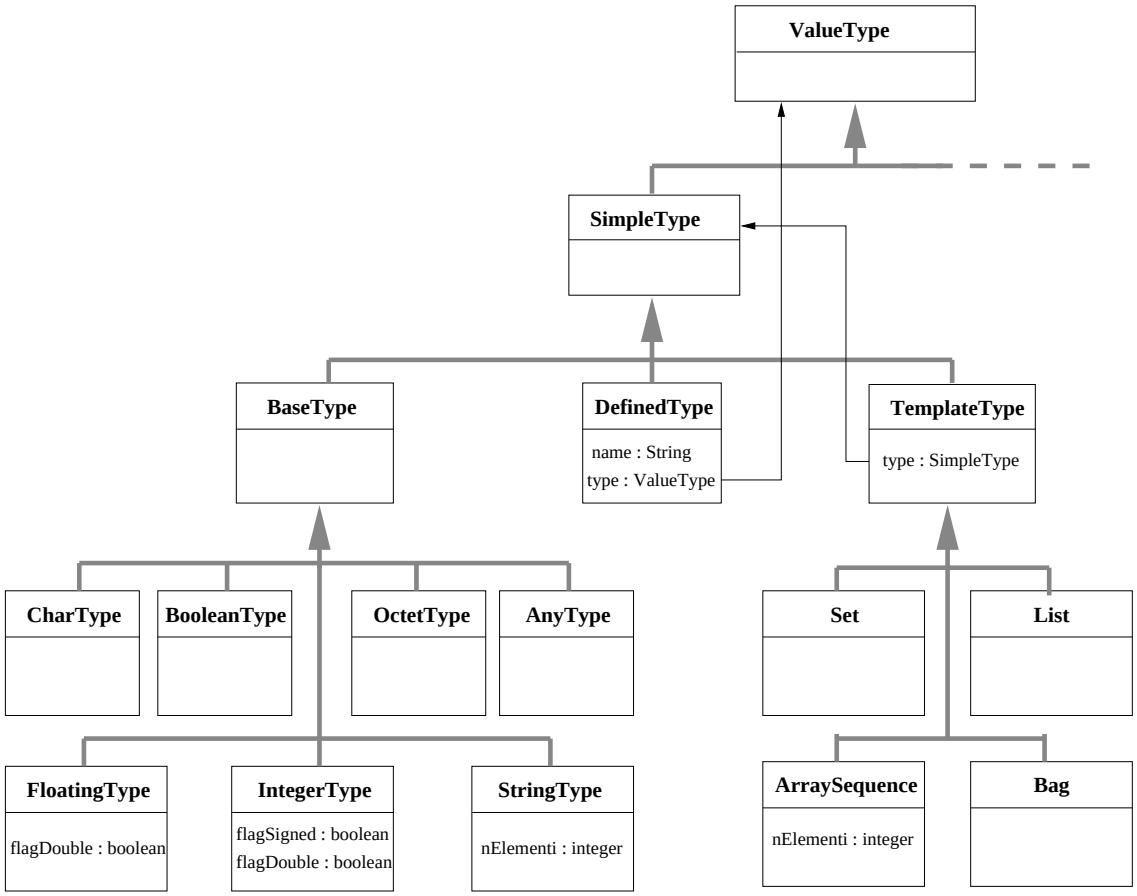



Figura 4.3: I tipi semplici

sé stante, ma deve sempre essere accompagnato dal *tipo-struttura*: ad esempio la seguente definizione

```
typedef struct nometag {
          int a;
          boolean b;
          char c;
        } tipostruct;
```

consente successivamente di dichiarare in modo del tutto equivalente le variabili nei due seguenti modi:

```
tipostruct var1;
struct nometag var2;
```

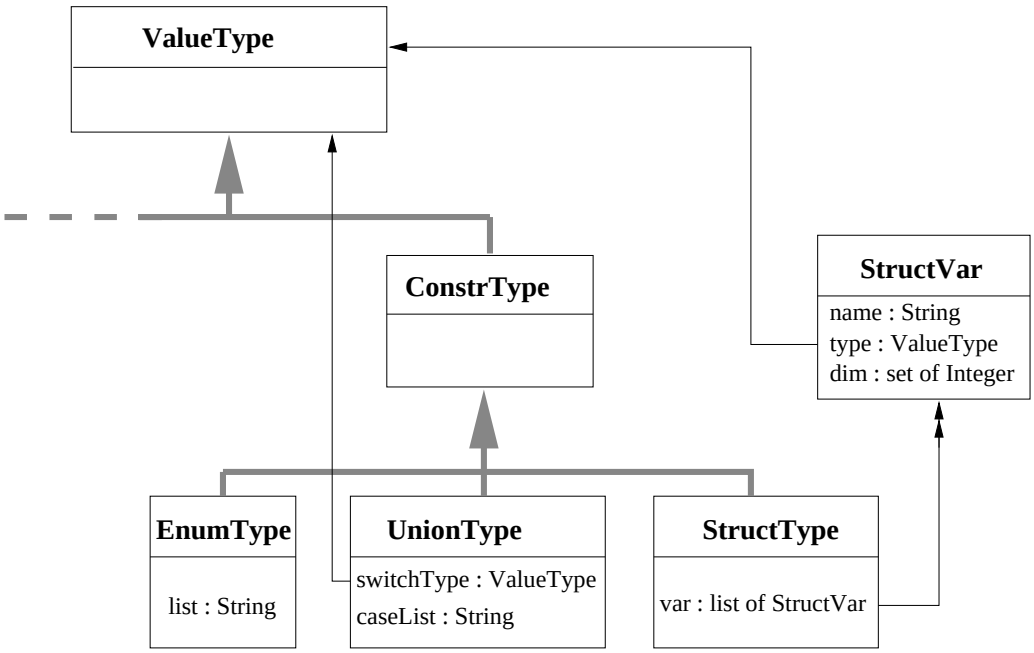



Figura 4.4: ConstrType

Per semplicità di rappresentazione e poiché attualmente non è ritenuto importante nell'ambito del progetto di integrazione, la classe che rappresenta il tipo Union è molto semplice: a parte il tipo della variabile di switch, tutto ciò che riguarda la definizione della union non viene interpretata, ma archiviata così come è stata definita. Naturalmente deve comunque essere garantita la correttezza sintattica della definizione.

Il tipo Enum si discosta da Struct e Union, perché rappresenta un semplice elenco finito di valori: tutte le variabili di questo tipo possono assumere solamente uno dei valori elencati nella definizione dello stesso.

### 4.4.3 I tipi classe

Le classi (o **Interface**) sono rappresentate da un'insieme di diverse dichiarazioni.

#### Ereditarietà

Il primo aspetto da considerare riguarda la collocazione della classe nello schema di figura 4.5. Ogni classe può avere una o più superclassi, cioè è ammessa l'ereditarietà multipla. Nello schema questa proprietà si manifesta attraverso la collezione complessa **inheritance** che mappa sulla stessa classe **Interface**.

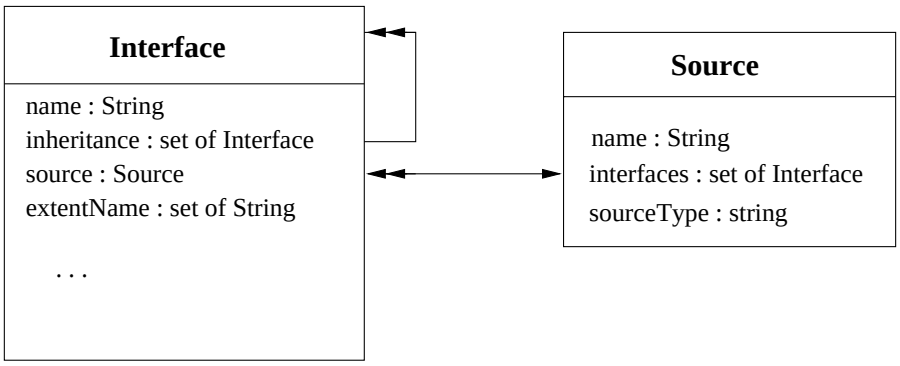



Figura 4.5: Ereditarietà e Source

### Source

Ogni classe appartiene ad un sorgente, le cui caratteristiche sono nome e tipo (relazionale, a oggetti, file, semistrutturato), come in figura 4.5.
Per consentire una migliore navigazione nella struttura dati è stato più che opportuno mantenere nell'oggetto **Source** la lista di tutte le classi che ne fanno parte. Infatti, dal momento che classi omonime possono esistere in sorgenti differenti, l'identificazione univoca di una classe è data dalla coppia $nomeSource - nomeInterface$.
Il costrutto per la dichiarazione del sorgente è stato aggiunto in ODL$_{I^3}$.

### Extent

L'Extent rappresenta l'insieme di tutte le istanze della classe all'interno di un particolare database. Se la classe è globale, possono essere dichiarati più *Extent* associati alla stessa, pertanto, in generale, la memorizzazione di questa parte della classe avviene attraverso l'uso di un set di nomi (stringhe).
La dichiarazione di un'estensione non è obbligatoria, una classe può non averne, tuttavia la sua presenza facilita il reperimento indicizzato (quindi rapido) delle informazioni da parte del DBMS.

### Chiavi e Foreign key

Si possono dichiarare delle chiavi candidate, semplici o composte. Ad ogni istanza dell'oggetto **KeyList** corrisponde una chiave candidata, che mappa un attributo (chiave semplice) o più attributi (chiave composta).
La possibilità di avere anche delle Foreign Key è una prerogativa del linguaggio ODL$_{I^3}$. Infatti le informazioni provenienti da uno schema relazionale sarebbero

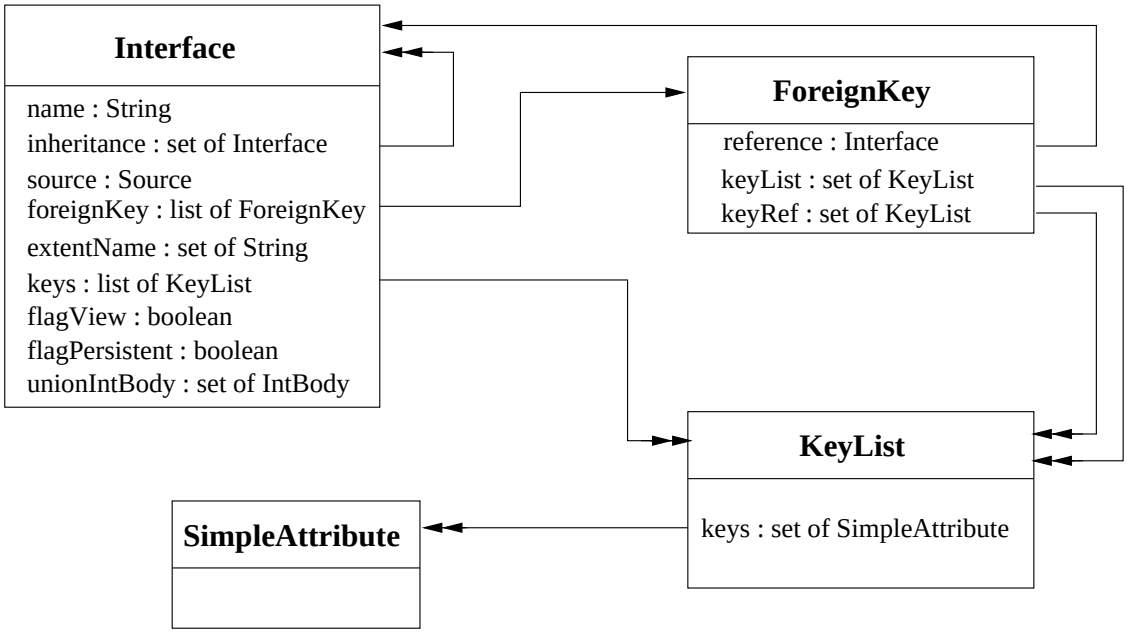



Figura 4.6: Extent, Chiavi e Foreign key

incomplete se si trascurasse questo aspetto, dal momento che nel modello relazionale le gerarchie di aggregazione si realizzano solo attraverso le foreign key.
L'oggetto **ForeignKey** che mantiene tale informazione contiene la lista degli attributi componenti, la lista dei medesimi attributi appartenenti alla classe riferita, e la stessa classe riferita.

**Esempio 2** Consideriamo l'esempio di riferimento relativo al contesto di un ospedale: la prima parte di definizione della classe `ID.Patient` fornita dal *wrapper* associato al sorgente relazionale `ID`, è la seguente:

```
interface Patient (
   source relational Intensive_Care_Department
   extent Patients
   key code
   foreign_key (test) references Test (number)
   foreign_key (doctor_id) references Doctor (id)
)
{  ...  }
```

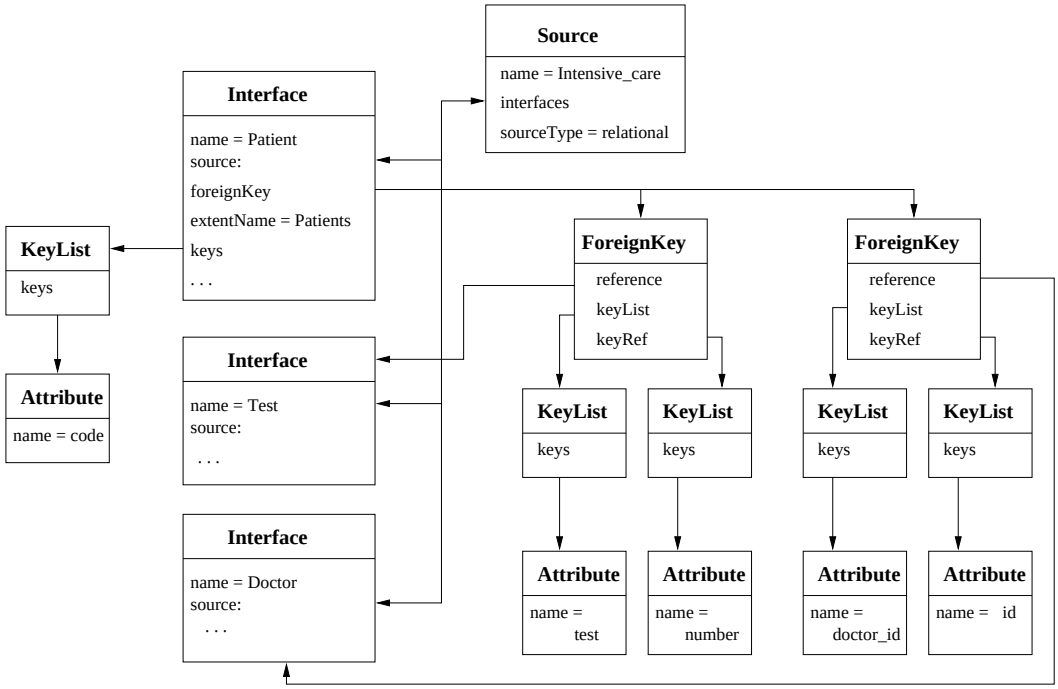



Figura 4.7: Esempio di rappresentazione della classe `ID.Patient`

Da questa descrizione ODL$_{I^3}$ si possono notare alcune caratteristiche tipiche di entità relazionali:

- la dichiarazione del sorgente CD, di tipo relazionale;
- la dichiarazione di un'unica estensione (infatti la classe è locale);
- la presenza di almeno una chiave candidata, obbligatoria per le tabelle relazionali;
- la presenza di due *foreign key* che producono aggregazione fra la tabella descritta e le tabelle `Test` e `Doctor` rispettivamente.

La figura 4.7 mostra come la struttura dati del parser ospita le informazioni di questo esempio.

### Interface body

Il corpo dell'interfaccia di una classe (o vista) è costituito dall'insieme di attributi e metodi che ne fanno parte. A differenza dell'ODL standard, nel lin-

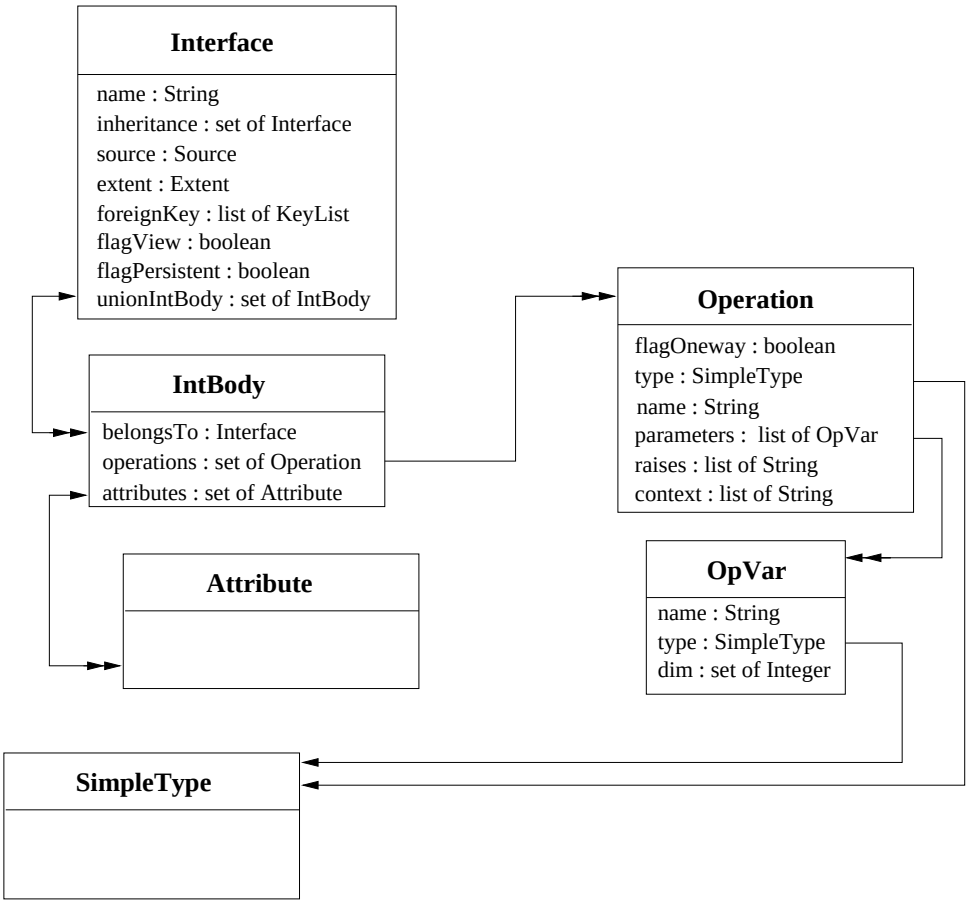



Figura 4.8: Interface body

guaggio ODL$_{I^3}$ è prevista la definizione di più implementazioni per una stessa classe. Questa caratteristica consente l'acquisizione di dati semistrutturati che, per definizione, non sono tenuti a seguire una rigida formattazione, come invece avviene per i dati strutturati.
Per chiarire meglio il concetto supponiamo di avere oggetti semistrutturati diversi che rappresentano lo stesso aspetto della realtà attraverso contenuti fortemente differenziati a livello strutturale, proprio perché non esiste una struttura cui tutte le istanze devono attenersi; un tipico esempio riguarda l'attributo *data*: in certi casi una data può essere rappresentata come l'insieme dei campi *giorno, mese* ed *anno*; in altri casi, semplicemente come una stringa.
Analogo discorso vale per altre informazioni, come ad esempio *ora*, *indirizzo*, ecc...

Il linguaggio descrittivo $ODL_{I^3}$ fornisce lo speciale costrutto *union* che consente di definire più specifiche alternative per una stessa interfaccia, in modo tale che possano essere messi a confronto oggetti che rappresentano due istanze della medesima realtà, ma che sono descritti attraverso scelte strutturali differenti.

**Esempio 3** Riprendendo l'esempio su come rappresentare l'attributo `date`, supponiamo di avere la seguente descrizione:

```
interface Appointment
{
  ...
  attribute struct {
      unsigned int year;
      unsigned short month;
      unsigned short day; } date;
  ...
};
union
{
  ...
  attribute string date;
  ...
};
```

Alla luce di ciò, è evidente che ad una stessa classe possano appartenere più attributi omonimi, uno per ogni implementazione. Stesso discorso vale per le operazioni.

### Operazioni

Le operazioni sono i metodi della classe, o meglio la loro signature: infatti tra le informazioni contenute nella classe **Operation** figurano il nome e il tipo restituito dal metodo, una lista di parametri passati (oggetti di tipo **OpVar**) ognuno avente un nome, un tipo e un elenco di eventuali dimensioni di array, come in figura 4.8. Notare che i tipi ammessi per i parametri passati e il tipo restituito dal metodo, devono essere istanze di **SimpleType**.

### Attributi

Esistono vari tipi di attributi: ci sono gli attributi semplici e quelli complessi: la differenza è tutta nel tipo: tipo-valore nel primo caso e tipo-classe nel secondo.

In ODL esiste anche la possibilità di dichiarare delle relationship, ovvero attributi complessi dei quali esiste la relazione inversa; infine il linguaggio $ODL_{I^3}$ fornisce un costrutto per definire regole di mapping che legano grandezze appartenenti allo schema integrato e grandezze degli schemi locali. Quando alla dichiarazione di un attributo seguono un insieme di regole di mapping, lo stesso è automaticamente considerato globale.
Nello schema di figura 4.10 sono isolati tre tipi di attributi: i **GlobalAttribute**, i **SimpleAttribute** e le **Relationship**.

I **SimpleAttribute** hanno un tipo generico (**Type**), un flag booleano che indica se sono di sola lettura e un set di eventuali indici d'array.

Le **Relationship** mappano una interfaccia e contengono le informazioni sull'inversa attraverso un puntatore alla stessa classe **Relationship**, inoltre *type* indica se l'attributo è o meno una collezione, mentre *orderBy* indica un criterio di ordinamento.

I **GlobalAttribute** hanno le stesse caratteristiche degli attributi locali, più un collegamento all'oggetto **MappingRule**

**Esempio 4** Consideriamo la classe `CD.Patient`, appartenente al sorgente semistrutturato `CD`; la definizione scritta in $ODL_{I^3}$ è la seguente:

```
interface Patient (
   source semistructured Cardiology_Department
)
{
   attribute string name;
   attribute string address;
   attribute set<Exam> exam*;
   attribute integer room
   attribute integer bed
   attribute string therapy*
   attribute set<Physician> physician*;
}
```

In questo esempio non sono presenti estensioni, né chiavi candidate o foreign key, a differenza di quanto avverrebbe nella descrizione di una classe relazionale. Si può inoltre notare dalla figura 4.9 la presenza di collezioni di attributi complessi.

### Mapping rule

Come già visto nella sezione 3.5, ci sono vari tipi di corrispondenza fra grandezze globali e locali, come mostrato in figura 4.10.
Ad un attributo globale, con riferimento ad una classe locale, può corrispondere:

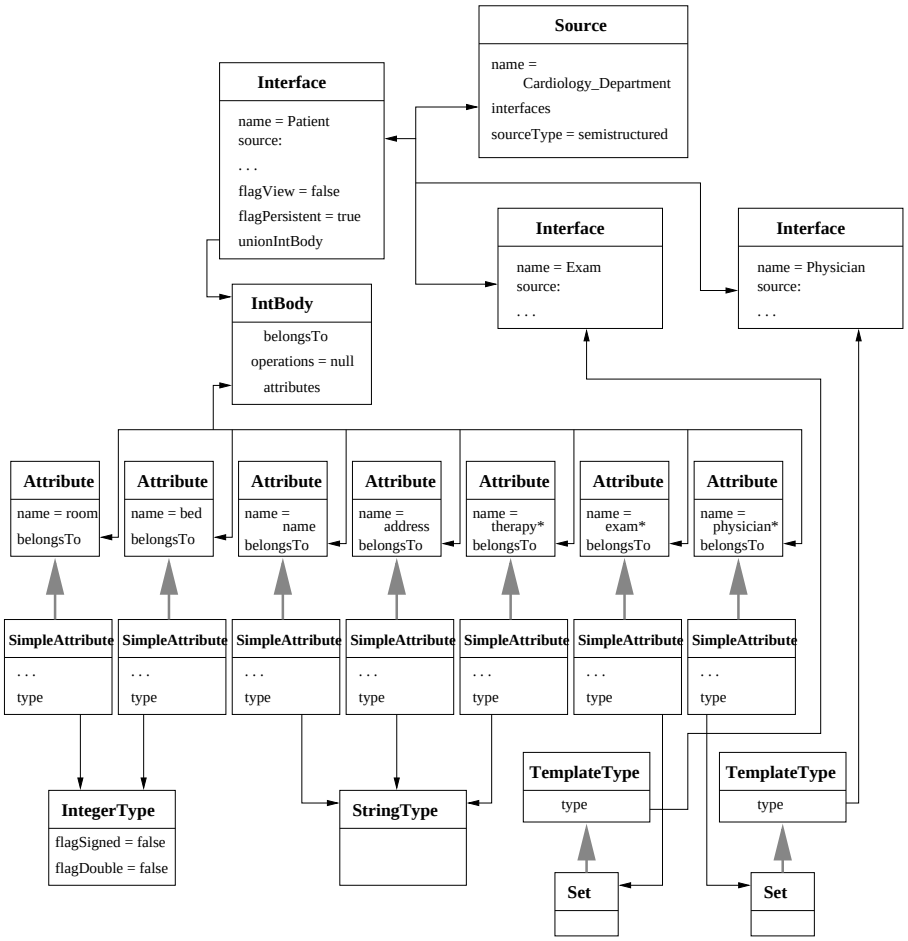



Figura 4.9: Esempio di rappresentazione della classe `CD.Patient`

- Un solo attributo locale; è il caso più semplice
- Un insieme di attributi locali in $and$ tra loro; e' il caso di $name$ che mappa localmente negli attributi $firstNameandlastName$. L'operatore $and$ assume il significato di concatenazione.
- Un insieme di attributi locali in $union$ tra loro; in questo caso all'attributo globale corrisponde uno solo alla volta tra quelli in $union$; il criterio che determina la scelta di quale attributo locale scegliere è dato dal valore di un terzo attributo locale, rappresentato nella struttura da **onParameter**.
- Un valore di default. È il tipico caso in cui si rende necessario esprimere un metaconcetto; riprendendo l'esempio ospedaliero, si presenta

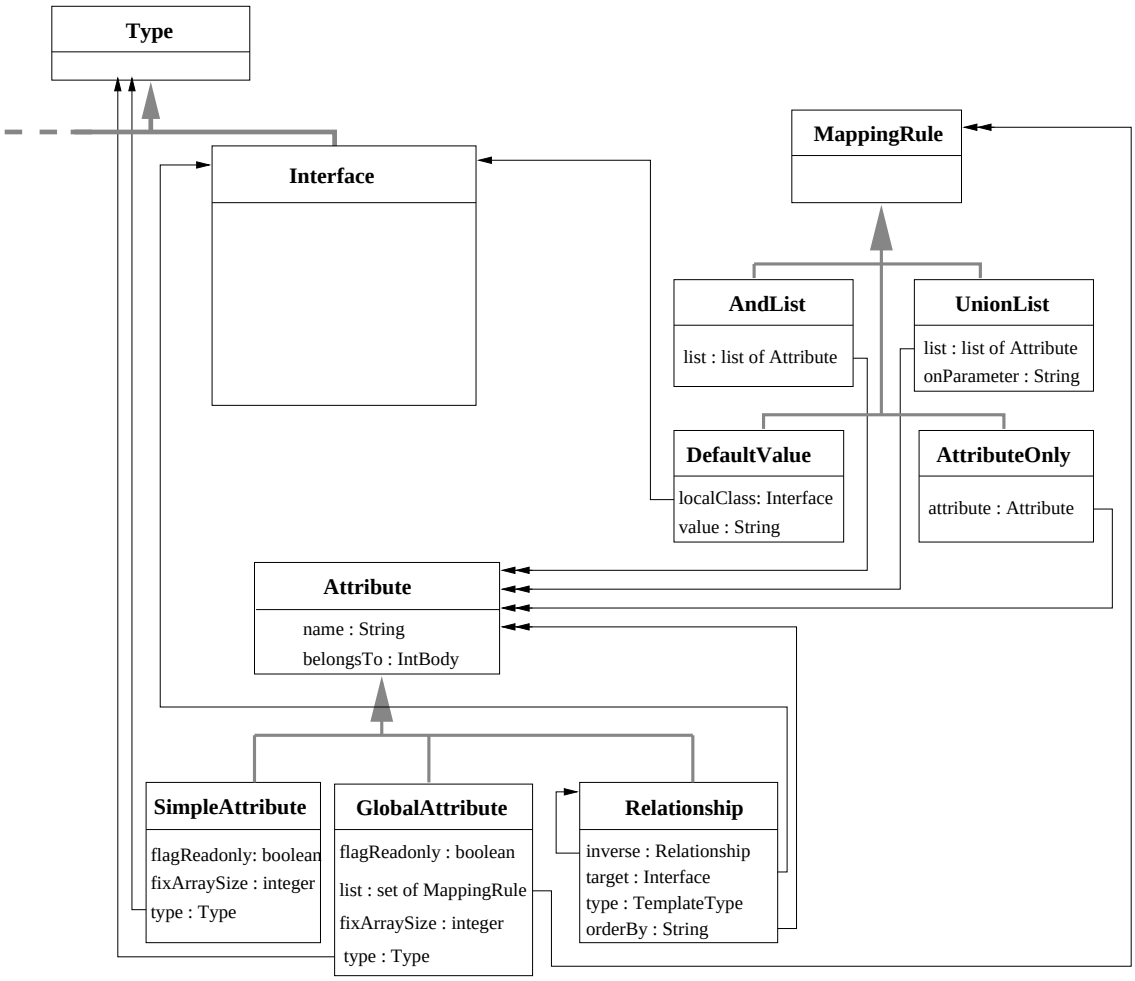



Figura 4.10: Attributi e Mapping rule

l'esigenza di imporre valori di default per l'attributo globale aggiunto `Hospital_Patient.dept`, poiché localmente la conoscenza sul reparto è implicita nel nome dei sorgenti.

`Hospital_Patient.dept = 'Cardiology'` associato a `CD.Patient`

`Hospital_Patient.dept = 'Intensive Care'` per `ID.Patient`

`Hospital_Patient.dept = 'Intensive Care'` per `ID.Dis_Patient`

L'esempio che segue mostra come si esprimono in $ODL_{I^3}$ alcune regole di mapping per la classe globale `Hospital_Patient`.

**Esempio 5** Al termine del processo di integrazione riguardante l'esempio di riferimento sono state ottenute tre classi globali: `Hospital_Patient`, `Medical_Staff` ed `Exam`; la descrizione in linguaggio $ODL_{I^3}$ della prima è la seguente:

```
interface Hospital_Patient
{
   attribute long code
        mapping_rule ID.Patient.code,
                     ID.Dis_Patient.code;
   attribute string name
        mapping_rule ( ID.Patient.first_name and
                       ID.Patient.last_name ),
                     CD.Patient.name;
   attribute string address
        mapping_rule ID.Patient.address,
                     CD.Patient.address;
   attribute set<Exam> exam*
        mapping_rule ID.Patient.test,
                     CD.Patient.exam*;
   attribute short room
        mapping_rule CD.Patient.room;
   attribute short bed
        mapping_rule CD.Patient.bed;
   attribute string therapy*
        mapping_rule CD.Patient.therapy*;
   attribute set<Medical_Staff> physician*
        mapping_rule CD.Patient.physician*,
                     ID.Patient.doctor_id;
   attribute string dept
        mapping_rule ID.Patient = 'Intensive Care',
                     ID.Dis_Patient = 'Intensive Care',
                     CD.Patient = 'Cardiology';
}
```

In questo esempio, rappresentato graficamente in figura 4.11, si notano alcune caratteristiche valide per ogni classe globale .

- non ci sono superclassi: lo schema integrato ha gerarchie di aggregazione, ma non ereditarietà;
- non viene specificato alcun sorgente, nessuna estensione e nemmeno chiavi o foreign key per la classe `Hospital_Patient`; questo vale per tutte le

classi globali, essendo l'informazione sul sorgente tipica delle grandezze locali;

- ogni attributo globale ha almeno una regola di mapping, cioè un corrispondente locale, sia esso un attributo o un valore di default, o entrambe le cose;
- poiché gli attributi locali `CD.Patient.physician` e `ID.Patient.doctor_id` sono complessi, il dominio del corrispondente globale `physician` deve a sua volta essere complesso: in particolare conterrà oggetti della classe globale `Medical_Staff`.

La figura 4.12 mostra tutta la porzione dell'**Object model** che riguarda i *tipi-classe*.

### 4.4.4 Costanti

La sintassi delle costanti in ODL è praticamente identica a quella dell'ANSI C.
La classe **Constant** che le rappresenta in figura 4.13 ha tre attributi:

- Nome della costante
- Tipo della costante, deve essere un oggetto **BaseType**
- Valore della costante. Per semplicità il valore viene archiviato in formato stringa: sarà cura di chi consulta la struttura dati eseguire il cast opportuno.

### 4.4.5 Relazioni terminologiche

Nel linguaggio $ODL_{I^3}$ è stata aggiunta la possibilità di inserire relazioni fra attributi ed attributi, classi e classi, e relazioni miste fra classi ed attributi. Il parser si preoccupa di interpretarle e di capire quali classi e/o quali attributi sono posti in relazione tra loro.
Come già scritto nella sezione 3.1, i tipi di relazione sono: sinonimia (SYN), ipernimia (BT), iponimia(NT), e associazione (RT). La figura 4.14 mostra come vengono archiviate della struttura dati, secondo i seguenti casi:

- La relazione coinvolge due classi: viene creato un oggetto **InterfaceRel**
- La relazione coinvolge due attributi: viene creato un oggetto **AttributeRel**
- La relazione coinvolge una classe e un attributo: viene creato un oggetto **AttrIntRel**

### 4.4.6 Regole di integrità

È data, dal linguaggio ODL$_{I^3}$, la possibilità di definire regole di integrità di tipo *if-then*, che devono essere verificate per ogni istanza nel database.
La corretta sintassi di una regola è la seguente:

**rule** *nomerule*
**forall** *iteratore* **in** *collezione* **:** *antecedente* **then** *conseguente*

oppure:
**rule** *nomerule*
[{ **case of** *identifier* **:** *caselist* }]

I termini che compaiono in questa parte di grammatica e che rappresentano gli attributi della classe **Rule** in figura 4.15, sono i seguenti:

- *nomerule*: è il nome della regola
- *iteratore*: è in identificatore che rappresenta l'istanza.
- *collezione*: è un insieme di istanze; può essere una classe o parte di essa.
- *antecedente*: è la condizione di *if*: si ottiene ponendo in *AND* una serie di predicati booleani.
- *conseguente*: è l'affermazione valida per tutte le istanze che verificano l'*antecedente*. La sintassi di *antecedente* e *conseguente* è la stessa.

La dichiarazione alternativa di *case*, fornisce il criterio di scelta fra più attributi locali che sono mappati in *union* da un unico attributo globale: nel paragrafo 3.5.2 si è trattato questo tipo di mapping, nel quale emerge l'esigenza, attraverso una regola, di sapere in base a quali valori di un dato parametro l'attributo globale corrisponde via via ad ognuno degli attributi locali.

Le condizioni *antecedente* e *conseguente* sono formate da un unico predicato booleano o da più condizioni in $and$ tra loro. Ogni predicato è un oggetto della classe **RuleBody**.
Come mostrato dallo schema in figura 4.16, ci sono varie possibilità.

#### In

Condizione vera se *dotName1* (ad esempio un oggetto) è contenuto nella collezione *dotName2*. Gli attributi si chiamano *dotName* e contengono una lista di stringhe perché la sintassi ODL$_{I^3}$ prevede che siano cammini espressi in *dot-notation*.

### Forall e Exist

La sintassi per questo predicato è la stessa della prima parte della regola, infatti prevede una chiamata ricorsiva a predicati booleani **RuleBody**

**Forall** *iteratore* **in** *collezione* **:** *condizioni_in_and*
Il predicato è vero se per tutte le istanze valgono tutte le condizioni in $and$.

**Exist** *iteratore* **in** *collezione* **:** *condizioni_in_and*
Il predicato è vero se esiste almeno un'istanza per cui valgono tutte le condizioni in $and$.

### Compare

Questo è il più classico dei predicati booleani: è vero o falso in base al risultato del confronto tra il valore di una variabile (ad esempio un attributo) e il valore di un letterale. Gli operatori sono: $=$, $<$ , $\leq$ , $>$ , $\geq$.
L'attributo *castType* contiene, se specificato, il tipo su cui fare il cast del valore letterale per rendere possibile il confronto con la variabile.

### RuleOperation

Questa condizione è vera se risulta verificato il confronto di uguaglianza fra una variabile (*dotName*) e il valore ritornato da una funzione invocata (chiamata operazione):

- *name*: è il nome dell'operazione.
- *opType*: è il tipo ritornato dall'operazione.
- *args*: è una lista di parametri (oggetti **RuleOpArg**), di ognuno dei quali si conosce:
  - *type*: il tipo del parametro;
  - *dotName* o *value* a seconda che il parametro sia rispettivamente una variabile/attributo/costante o un letterale;

### 4.4.7 L’Object Model

Riporto in figura 4.17 l’intero modello ad oggetti, così come è stato costruito ed utilizzato dal parser ODL$_{I^3}$.

## 4.5 Controlli

Compito del parser non è soltanto quello di acquisire informazioni e caricarle nella struttura dati presentata nella sezione precedente; uno dei compiti principali che gli spettano riguarda i controlli sulla correttezza non solo sintattica, ma anche semantica. Infatti rispettare la sintassi di un particolare linguaggio non è condizione sufficiente per affermare che l'informazione è corretta: esistono regole di coerenza che possono essere garantite solamente da ulteriori controlli. Nel caso trattato in questa tesi sono:

- Controllo dei sorgenti di ogni interfaccia
- Controllo delle superclassi di ogni interfaccia
- Controllo delle chiavi
- Controllo delle foreign key
- Controllo delle mapping rule
- Controllo delle relationship
- Controllo delle relazioni terminologiche

Un criterio di classificazione dei controlli distingue quelli che vengono effettuati in tempo reale, cioè nel momento stesso in cui si fa l'analisi del file sorgente, e quelli che sono effettuati al termine dell'acquisizione dei dati.

Al primo tipo appartiene non solamente il controllo grammaticale ma anche, ad esempio, quello sui tipi-valore:

**Esempio 6** *Supponiamo di incontrare la seguente serie di dichiarazioni:*

```
typedef double t1;
...
t1  g;
```

*In questo caso in tempo reale viene verificata l'esistenza del tipo definito "t1", perché il linguaggio* $ODL_{I^3}$*, come del resto l'ANSI C, prevede che nella dichiarazione di una variabile si possa richiamare un tipo-valore solo se questo è già stato definito in precedenza. Alla luce di ciò, la definizione seguente dà luogo ad errore di* tipo non definito*:*

```
t1  g;
...
typedef double t1;
```

È sempre preferibile il controllo durante l'acquisizione, perché, evidentemente, è più veloce nell'individuare eventuali errori. Tuttavia nella maggior parte dei casi l'unico modo per verificare correttamente la coerenza delle informazioni è farlo a posteriori; un esempio tipico è quello delle Relationship:

**Esempio 7** *Consideriamo la seguente definizione di relationship:*

```
interface Classe1 {
...
relationship Classe2 rel1
   inverse Classe2::rel2;
...
}

interface Classe2 {
...
relationship Classe1 rel2
   inverse Classe1::rel1;
...
}
```

*Per come è definita la sintassi delle relationship, le classi coinvolte devono fare riferimento l'una all'altra vicendevolmente, di conseguenza, qualsiasi sia l'ordine in cui vengono definite, la prima delle due richiama l'altra che ancora non esiste, rendendo così impossibile stabilire in fase di parsing la correttezza di tale definizione.*

### 4.5.1 Controllo della sorgente

Nel linguaggio $ODL_{I^3}$ non esiste una parte dedicata alla definizione delle sorgenti; l'esistenza di un determinato Source si esprime all'interno della definizione di ogni classe che ne fa parte, in modo ridondante.
Poiché il nome di una sorgente è sufficiente ad individuarla univocamente, non possono esistere due oggetti Source distinti con lo stesso nome.

Questo tipo di controllo viene fatto in fase di parsing, infatti nel momento stesso in cui una classe dichiara nome e tipo della sorgente di appartenenza, il compito del software consiste in:

- Costruire un nuovo oggetto Source se non ne esiste uno con quel nome
- Se ne esiste uno con quel nome, verificare che sia identico, cioè che anche il tipo coincida.

### 4.5.2 Controllo delle superclassi

Ogni interfaccia dichiara, eventualmente, una o più classi genitrici. A posteriori bisogna verificare che, nello stesso schema sorgente, le presunte superclassi esistano come interfacce.
A questo livello sarebbero necessari anche altri controlli non implementati: ad esempio che una classe non sia superclasse di se stessa, o che due interfacce non siano mutuamente superclasse e sottoclasse, più in generale che la gerarchia di ereditarietà non presenti dei cicli.

### 4.5.3 Controllo delle chiavi candidate

La definizione delle eventuali chiavi viene fatta separatamente rispetto a quella degli attributi, pertanto si rende necessario controllare che tutte i componenti le chiavi siano anche attributi della medesima interfaccia.

### 4.5.4 Controllo delle foreign key

Il controllo sulle foreign key è simile a quello fatto sulle chiavi: tutti i componenti la foreign key devono essere attributi della classe. Inoltre occorre verificare l'esistenza della classe riferita e, in essa, l'esistenza di attributi aventi i nomi coincidenti con quelli dati agli attributi riferiti. Il Data Flow Diagram di questa verifica è riportato in figura 4.18.

### 4.5.5 Controllo delle mapping rule

Sugli attributi appartenenti allo schema globale possono essere imposte relazioni di mapping con attributi locali. Un attributo viene individuato attraverso la dot-notation *nomeSource.nomeClasse.nomeAttributo*. La verifica di esistenza dello stesso, operazione propria di questo controllo, passa attraverso la successione ordinata di tre controlli: prima il sorgente, poi l'interface ed infine l'attributo.
Come già spiegato in precedenza nel paragrafo 4.4.3, ci sono quattro tipi di mapping: tre di questi fanno riferimento ad attributi locali, mentre il quarto (mapping "Default value") richiama una classe locale: in tal caso la verifica si conclude al reperimento dell'oggetto Interface.

### 4.5.6 Controllo delle relationship

Certamente quella delle Relationship è la verifica più elaborata: il tipo di aggregazione è binario, pertanto, come già visto nell'esempio 7, le informazioni che devono essere dichiarate sono:

- nome della relazione (nell'esempio `rel1`)
- nome della relazione inversa (nell'esempio `rel2`)
- nome della classe target (nell'esempio `Classe2`)

Dopo aver controllato che esista `Classe2` e, in essa, `rel2`, la verifica non è terminata, perché occorre che i dati dichiarati in `rel2` siano coerenti; in particolare:

- la classe target di `rel2` deve essere `Classe1`
- la relazione inversa di `rel2` deve essere `rel1`

### 4.5.7 Controllo delle relazioni terminologiche

Il compito del software consiste nell'individuare classi e/o attributi i cui nomi, espressi in *dot notation*, sono posti in relazione. Al termine del controllo, l'oggetto **ThesRelation** conterrà i relativi collegamenti.

## 4.6 Il software

Per realizzare la struttura dati atta a contenere tutte le informazioni dei vari schemi che devono essere integrati, la scelta implementativa è ricaduta sul linguaggio ad oggetti **Java** versione 1.2beta4 [50].
La decisione non è stata facile, considerato il fatto che java mi era totalmente sconosciuto; una possibile scelta alternativa poteva essere la programmazione nel più conosciuto **ANSI C**. Tuttavia le motivazioni che mi hanno spinto alla scelta più radicale sono essenzialmente le seguenti:

- Java è un linguaggio ad oggetti che si presta bene ad implementare una struttura a sua volta organizzata secondo uno schema ad oggetti.
- In Ansi C sarebbe stato necessario implementare un'architettura dinamica e piuttosto articolata di liste, records e puntatori; java nasconde la presenza di puntatori ed effettua in modo del tutto trasparente l'allocazione e deallocazione di spazio di memoria dinamica (Garbage collection), sollevando pertanto il programmatore da rischi di errori a *run-time* difficili da individuare.
- La programmazione ad oggetti permette una strategia di programmazione di tipo bottom-up e una maggiore modularità del software, consentendo, di conseguenza, una migliore lettura dello stesso e soprattutto una maggiore riusabilità.

- Le specifiche del linguaggio ODL$_{I^3}$ negli ultimi tempi hanno subito delle modifiche, pertanto sarebbe stato comunque necessario apportare cambiamenti al software già realizzato dai miei colleghi, operazione di per sé piuttosto difficoltosa, dal momento che ognuno ha il proprio stile di programmazione.

- Non ultimo, la possibilità di acquisire padronanza di un nuovo linguaggio, ma soprattutto di un nuovo modo di pensare la programmazione (a oggetti, appunto) ha rappresentato un'occasione importante, soprattutto dal punto di vista di prospettive professionali.

Il modulo software che implementa il parser è stato realizzato in due fasi: la prima consistente in acquisizione, caricamento della struttura dati e controllo sintattico, nonché esecuzione di tutti i controlli di coerenza eseguibili durante l'analisi; la seconda relativa a controlli semantici e di coerenza messi in atto a posteriori.

**Acquisizione e controllo sintattico** La prima parte di questa fase consiste nella realizzazione di una funzione ($yylex$) che, attraverso un diagramma a stati finiti, è in grado di analizzare ed interpretare stream di caratteri e di restituire, in base al riconoscimento di sequenze preimpostate, opportuni valori di *token*.
Successivamente mi sono avvalso di uno strumento software, $Byacc$ [51], che permette di implementare direttamente le regole di sintassi scritte in BNF; accanto ad ogni possibile regola sintattica, il programmatore deve preoccuparsi di scrivere le cosiddette *actions*, ovvero le istruzioni che memorizzano le informazioni nella struttura dati e i comandi che svolgono la parte di verifiche che si possono effettuare in tempo reale. $Byacc$ realizza automaticamente il codice java che effettua i controlli sintattici del testo specificati in BNF e lascia inalterate le *actions*. Il risultato finale è il parser.

**Controlli finali di coerenza** Al termine dell'acquisizione il main lancia in successione un metodo di controllo definito per ogni classe che possiede oggetti da controllare e aggiornare.

**Disponibilità del software** Il software che implementa il parser è composto dal file *parser.class*, il cui sorgente (*parser.java*) è generato da $Byacc$ a partire dal file *parser.y*; la struttura dati è contenuta nel package denominato *Odli3*. Il parser è accessibile all'indirizzo *http://sparc20.dsi.unimo.it/tesi/index.html*.

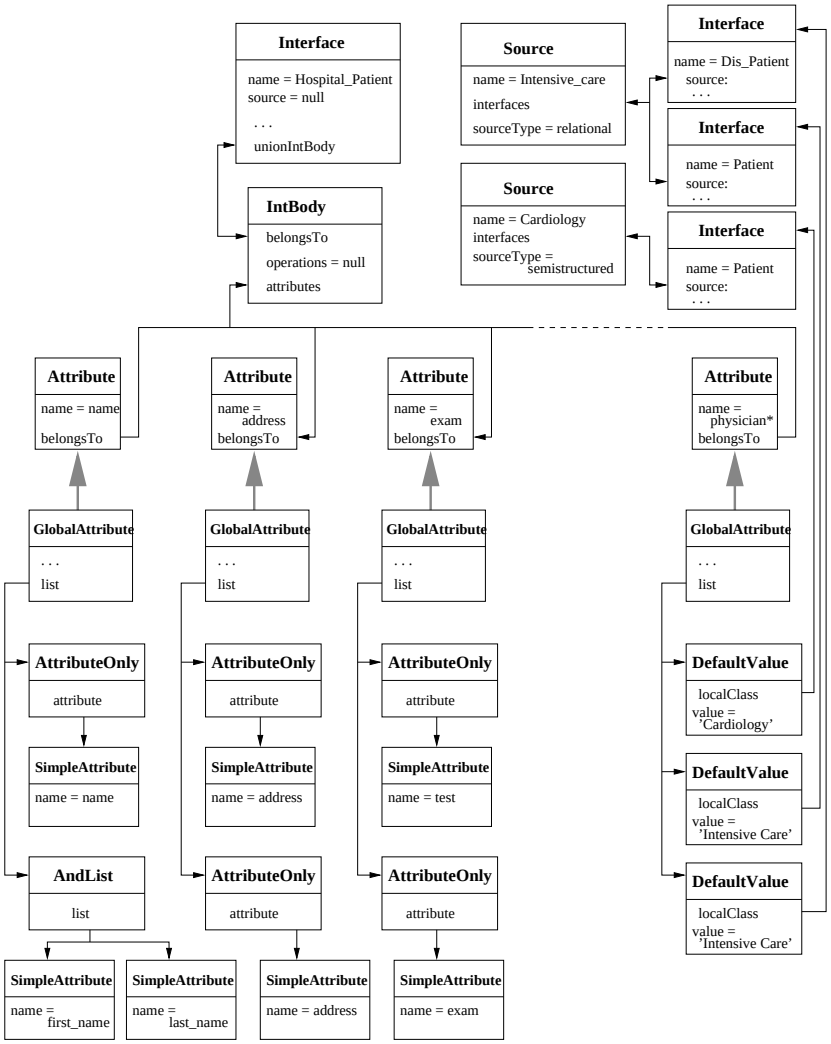



Figura 4.11: Esempio di *Mapping Rule* fra attributi globali e locali

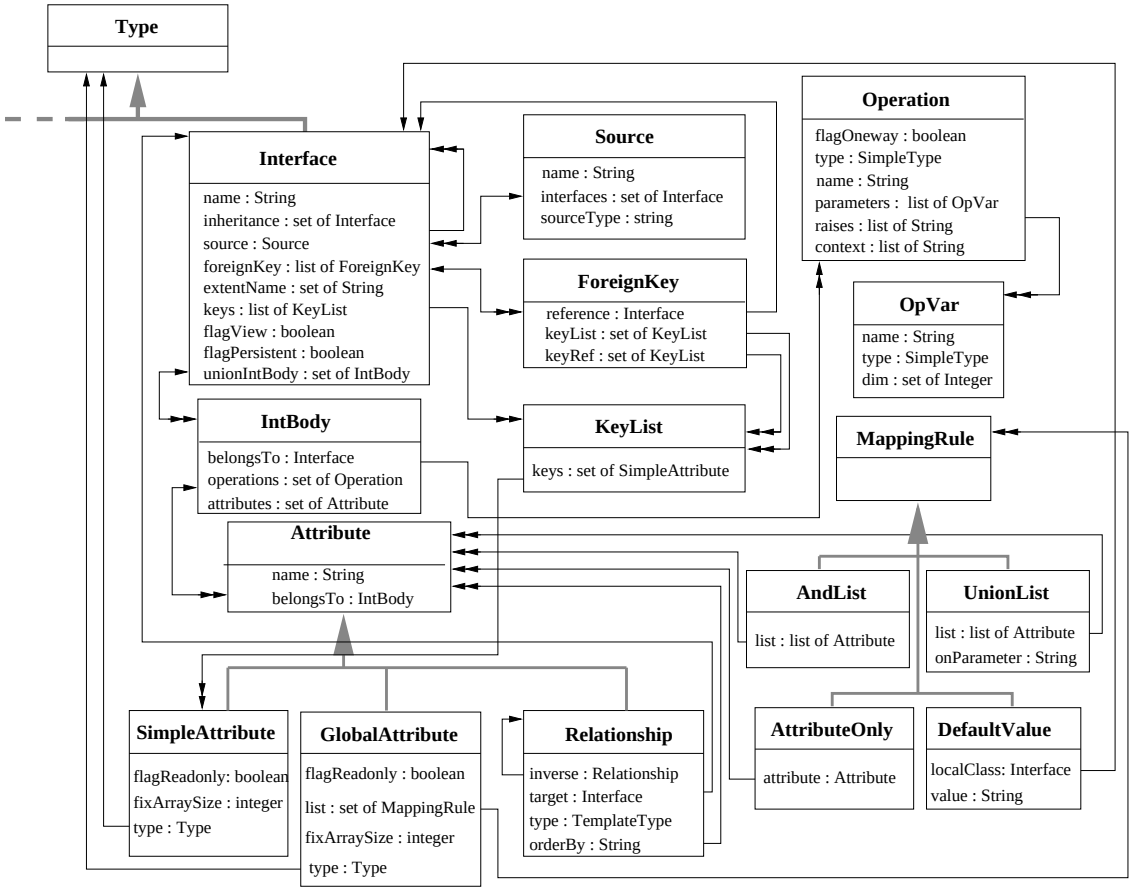



Figura 4.12: Schema completo della rappresentazione dei tipi-classe

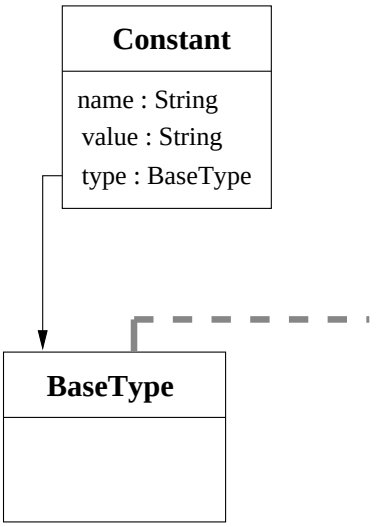



Figura 4.13: Le costanti

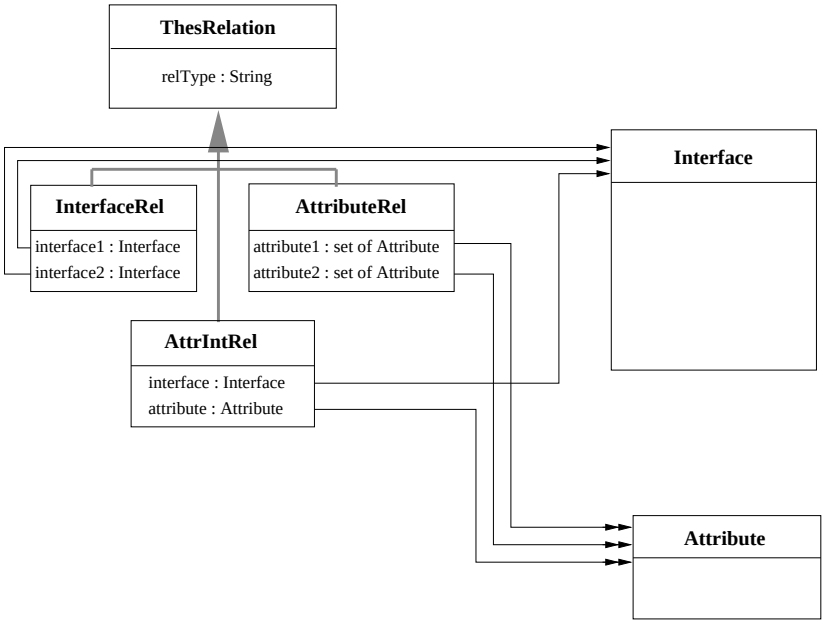



Figura 4.14: Relazioni terminologiche

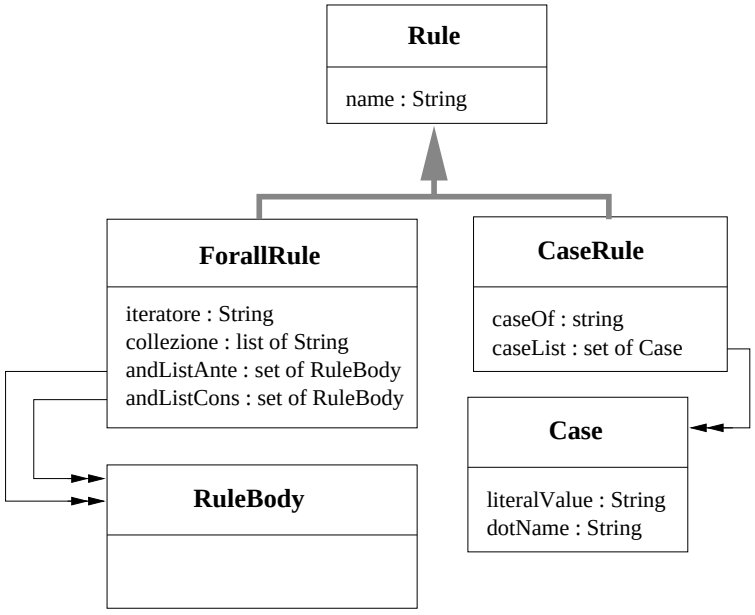



Figura 4.15: Regole di integrità

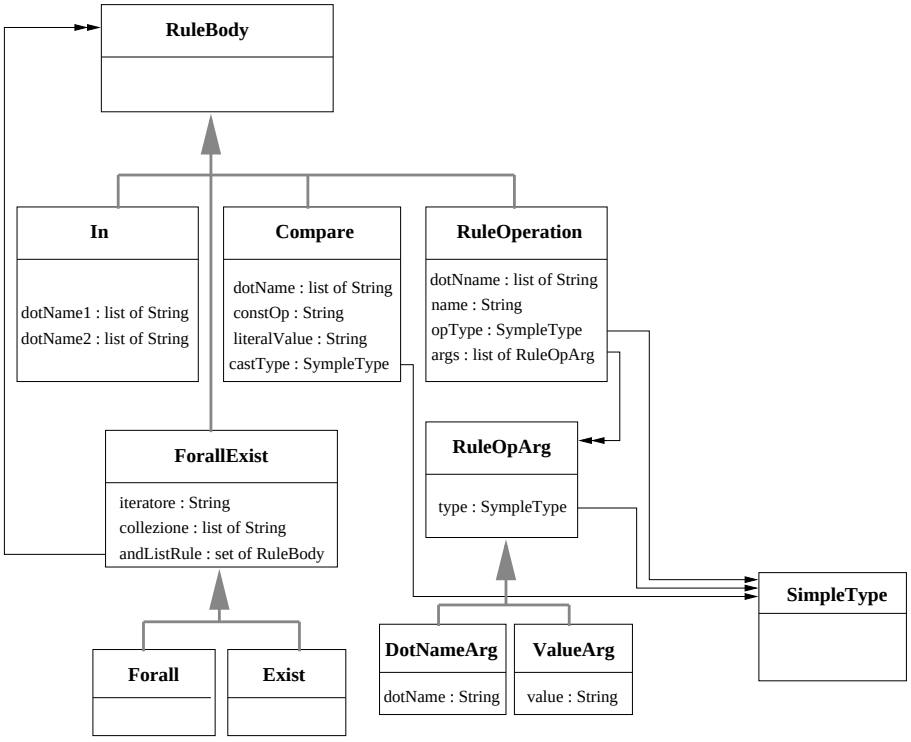



Figura 4.16: Predicati booleani componenti le condizioni di *antecedente* o *conseguente*

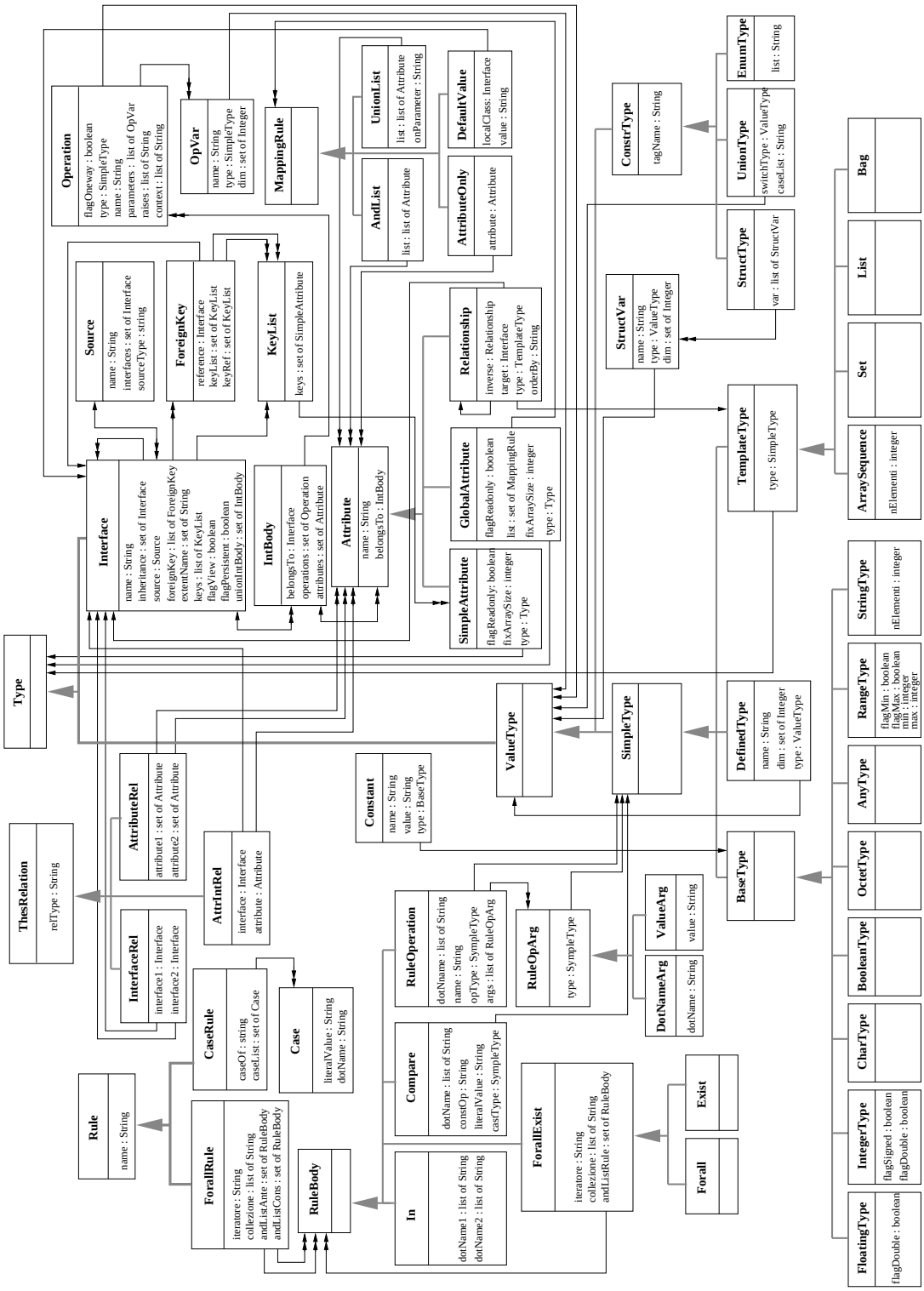



Figura 4.17: L'object model del linguaggio ODL$_{I^3}$

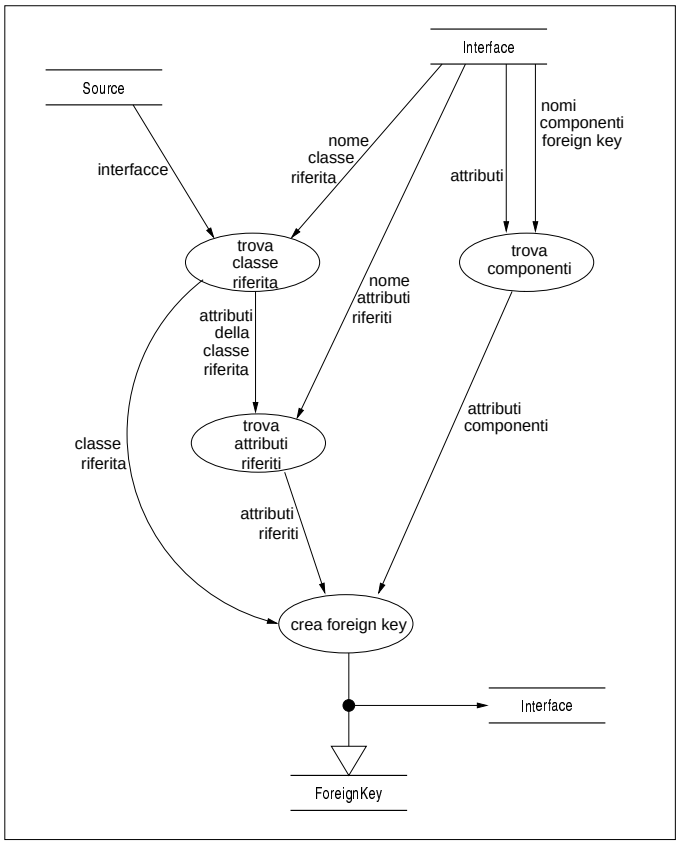



Figura 4.18: Verifica delle foreign key

# Capitolo 5

# SI-Designer

In questo capitolo viene descritto il componente software **SI-Designer**, che permette al progettista di effettuare tutti i passi relativi alla integrazione di schemi $ODL_{I^3}$ provenienti da basi di dati eterogenee: Le fasi da seguire, già trattate nel capitolo 3, sono le seguenti:

1. Generazione del thesaurus comune;
2. Calcolo delle affinità;
3. Generazione dei Cluster;
4. Definizione della Mapping Table

In questo contesto, il lavoro svolto dal componente SI-Designer è duplice: da un lato deve fornire al progettista un'interfaccia amichevole che lo assista per tutta la durata delle suddette fasi; dall'altro si preoccupa di mandare in esecuzione i componenti che di volta in volta sono interessati a svolgere le funzioni pertinenti, coordinandone quindi il lavoro.

## 5.1 Generazione del thesaurus comune

Nel sistema Momis, il thesaurus comune rappresenta una delle basi di conoscenza attraverso cui si perviene allo schema globale integrato.
Il processo di definizione del thesaurus è semiautomatico, in quanto il sistema non è in grado da solo di estrarre tutte le relazioni esistenti fra le classi rappresentate nei database sorgenti; poiché quindi l'intervento del progettista è indispensabile, questa fase è caratterizzata da una forte interazione fra il sistema e il progettista stesso.

I passi necessari alla generazione del dizionario sono i seguenti:

1. Acquisizione degli schemi sorgenti
2. Estrazione automatica delle relazioni (strutturali e terminologiche)
3. Integrazione/Revisione delle relazioni
4. Validazione delle relazioni fra attributi
5. Inferenza di nuove relazioni fra classi

### 5.1.1 Acquisizione degli schemi sorgenti

È richiesta al progettista la lista dei file che descrivono, in linguaggio $ODL_{I^3}$, gli schemi sorgenti, o quantomeno la parte di schemi che si intende rendere disponibile in forma integrata all'utente finale.

La figura 5.1 mostra l'interfaccia amichevole fornita da **SI-Designer**; la finestra grafica è composta da:

- un campo riga denotato da *Nome del file che descrive la sorgente*, editabile dal progettista e sul quale lo stesso inserisce il nome del file che descrive una sorgente (il path completo);
- un pulsante *Aggiungi*, attraverso il quale si aggiunge all'elenco la nuova sorgente/file da acquisire;
- una lista (*Lista file delle sorgenti da integrare*) non editabile contenente l'elenco delle sorgenti/file finora inserite;
- Il pulsante *Acquisizione sorgenti ed estrazione relazioni* la cui pressione comporta la terminazione delle descrizioni delle sorgenti da integrare e attiva automaticamente il modulo $\mathbf{SIM}_1$.

All'atto dell'inserimento, provocato da *Aggiungi*, il sistema depura la stringa contenuta nel campo da eventuali spazi e tabulazioni iniziali e finali; successivamente, prima di inserirne il contenuto nella lista, controlla se la stringa è non vuota e, in questo caso, se il suo contenuto non è già presente nell'elenco; in tal modo la lista non contiene mai duplicati o righe vuote.
Per cancellare o modificare il nome di un file, lo si seleziona nella lista: il suo valore viene riportato nel campo *Nome del file che descrive la sorgente*, diventando così modificabile e cancellabile.

Il pulsante *Acquisizione sorgenti ed estrazione relazioni* è attivo solo se l'elenco contiene almeno due file da acquisire. Infatti, perché l'integrazione abbia significato è obbligatorio che gli schemi sorgenti siano almeno due.

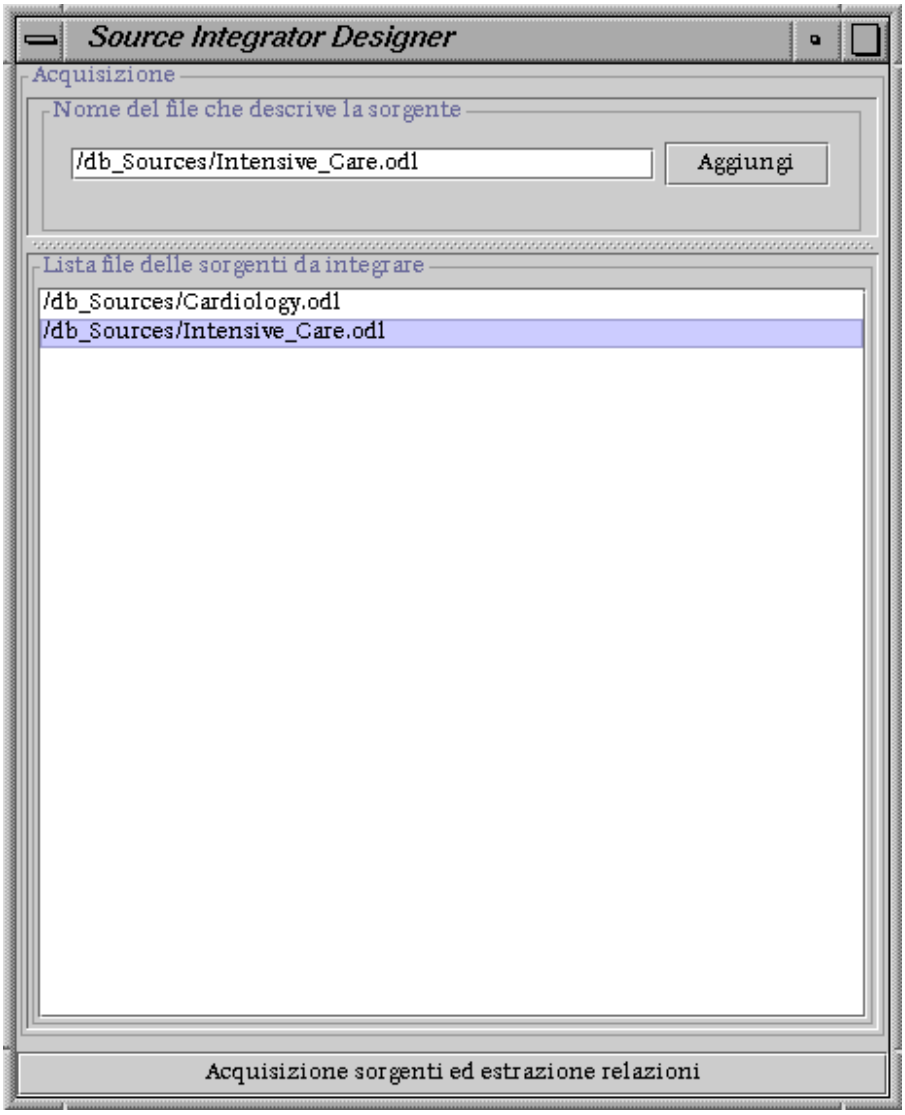



Figura 5.1: Acquisizione degli schemi sorgenti

### 5.1.2 Estrazione automatica delle relazioni

Alla pressione del pulsante *Acquisizione sorgenti ed estrazione relazioni* il sistema controlla che tutti i file presenti nell'elenco esistano; successivamente manda in esecuzione il modulo software **SIM**$_1$ (descritto in [25]), che ha il compito, interagendo con **ODB-Tools**, di estrarre le relazioni intraschema dalla struttura dei singoli file sorgenti.
Inoltre, tutti i termini utilizzati per dare nome alle classi locali e agli attributi, vengono analizzati da **WordNet**, un modulo software descritto brevemente nella sezione 2.3.2 e più ampiamente in [22], che si preoccupa di estrarre relazioni puramente terminologiche tra i concetti espressi.

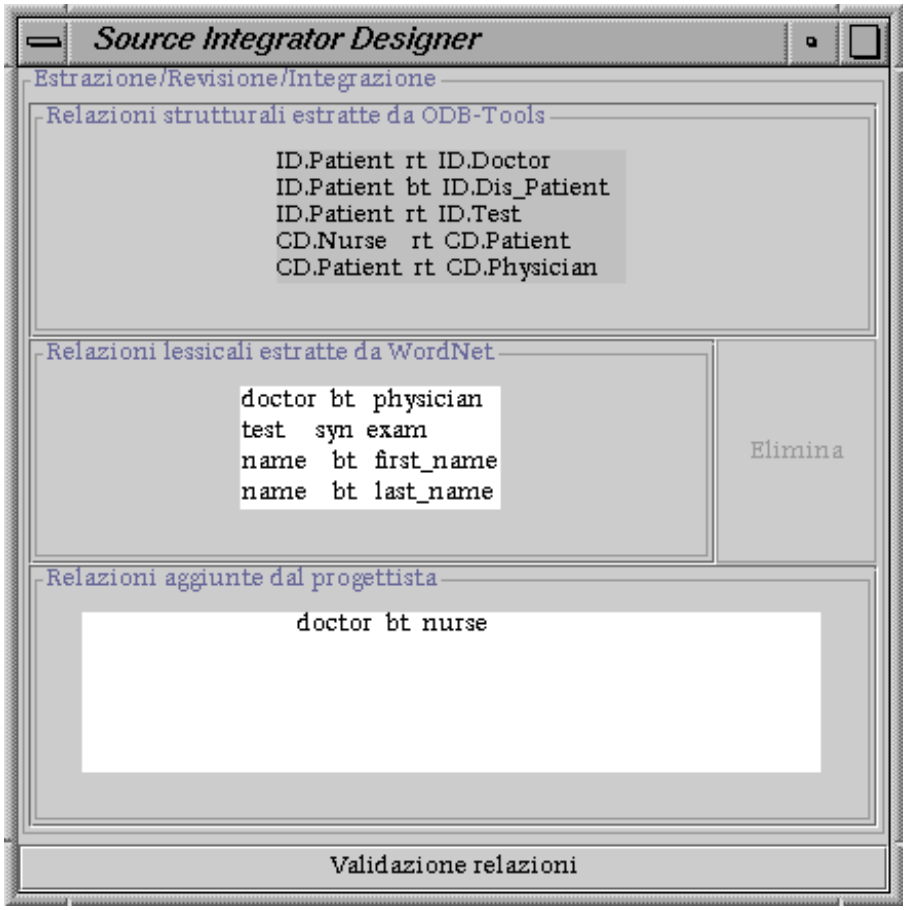



Figura 5.2: Estrazione e integrazione delle relazioni

I risultati di queste due operazioni di estrazione vengono visualizzati in due

aree distinte della seconda schermata, così come appare in figura 5.2. Nella parte alta della finestra sono mostrate le relazioni estratte dalla struttura degli schemi sorgenti. Poiché si assume che la rappresentazione locale sia corretta, queste relazioni sono a loro volta esatte, e non possono essere modificate.
Lo stesso non si può dire delle relazioni derivate dal dizionario lessicale: in questo caso il progettista ha l'ultima parola: può, a sua discrezione, cancellare quelle che ritiene non corrette. Per cancellare una relazione è sufficiente selezionarla e premere il pulsante *Elimina*.

### 5.1.3 Integrazione/Revisione delle relazioni

Così come l'operazione di estrazione di relazioni da parte del dizionario lessicale può mettere a confronto concetti che in questo particolare dominio applicativo il progettista ritiene non essere correlati, allo stesso modo è presumibile che concetti affini non siano tutti estratti automaticamente da **WordNet**.
L'area editabile posta nella parte bassa di figura 5.2, denotada dall'etichetta *Relazioni aggiunte dal progettista*, ha lo scopo di fornire al progettista la possibilità integrare la parte di thesaurus fin qui ottenuta, mediante l'aggiunta di ulteriori relazioni terminologiche.

### 5.1.4 Validazione

Con la pressione del pulsante *Validazione relazioni* si passa alla fase successiva: **SI-Designer** raccoglie tutte le relazioni puramente terminologiche, quelle cioè estratte dal dizionario e non cancellate, e quelle aggiunte, e le passa al modulo **SIM**$_1$, il quale provvede a confrontarle con i nomi delle classi e degli attributi locali e, dopo aver scartato quelle inutili (nelle quali uno dei due termini, o entrambi, non è né nome di classe né nome di attributo), le divide in tre liste:

1. *relazioni tra attributi*: vengono analizzate ed elaborate al fine di stabilire, in base al criterio di confronto fra i domini illustrato al paragrafo 3.1.4, quali fra esse sono da considerare *valide*.

2. *relazioni tra classi*: le relazioni che coinvolgono coppie di classi saranno analizzate in un secondo momento; vengono posizionate in una apposita lista

3. *relazioni miste* Qualora siano presenti relazioni miste (per relazione mista si intende una relazione tra il nome di una classe e il nome di un attributo) il modulo SIM$_1$ non le elabora: queste faranno parte direttamente del Thesaurus definitivo.

Nella parte alta di figura 5.3 è visualizzato il risultato della validazione delle relazioni appartenenti alla prima lista.

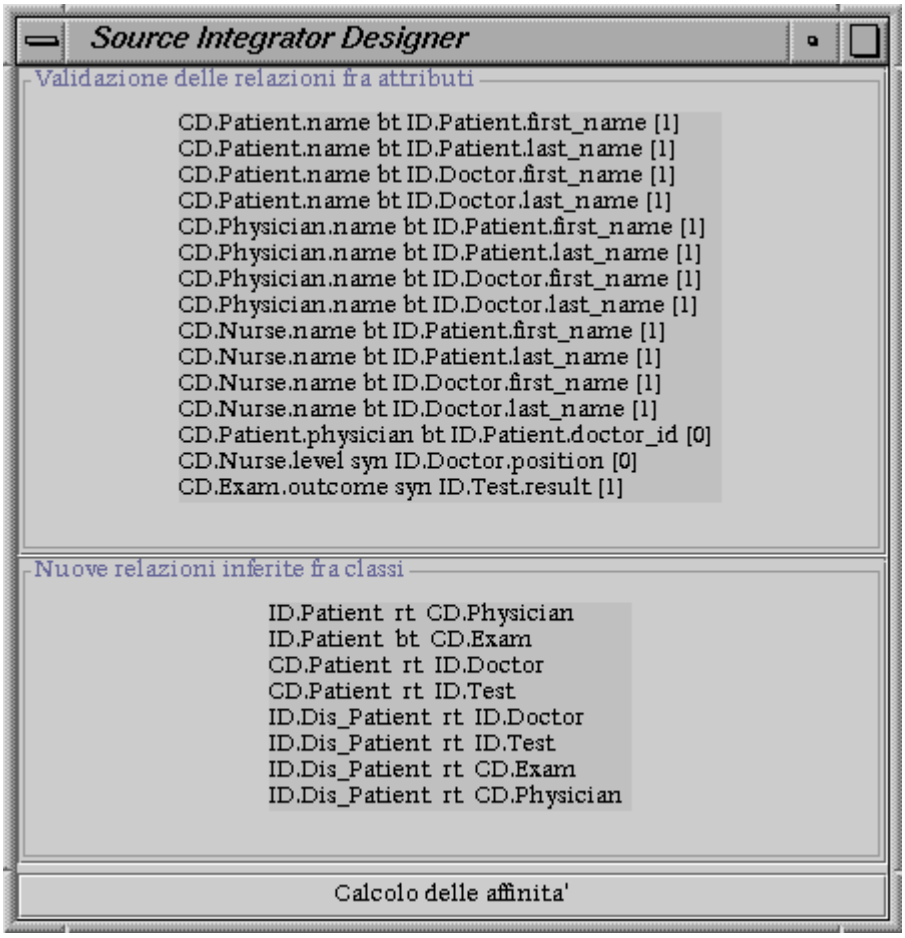



Figura 5.3: Validazione delle relazioni fra attributi e deduzione di nuove relazioni fra classi locali

### 5.1.5 Inferenza di nuove relazioni

Come già spiegato nel paragrafo 3.1.5, **SIM**$_1$, in base alle relazioni fin qui valide, apporta alcune modifiche temporanee ai nomi e ai domini degli attributi, e alle gerarchie fra le classi. In virtù di queste operazioni, si ottiene uno schema unico provvisorio, la cui descrizione viene prima tradotta in linguaggio OLCD, e poi passata al modulo **OLCD-Designer** che, facendo uso di tecniche di intelligenza artificiale, in particolare dell'algoritmo di sussunzione, è in grado di inferire

nuove relazioni fra classi locali. **SI-Designer** visualizza il risultato dell'inferenza insieme al quello della validazione, come in figura 5.3. Notare che in questa videata sono mostrate relazioni non genericamente terminologiche, ma fra ben precise entità locali (classi e attributi) degli schemi in questione.

A questo punto del processo il thesaurus comune è completo.

## 5.2 Calcolo delle affinità

Il calcolo delle affinità rappresenta la seconda fase di progettazione della vista integrata. Il componente software che si occupa dello svolgimento di questa operazione è **Artemis** [20, 21, 45].

Sulla base delle relazioni facenti parte del Thesaurus comune, viene costruito un grafo non orientato che mette in connessione i nomi delle entità locali, siano esse classi o attributi. Ad ogni relazione valida corrisponde un arco del grafo, il cui peso dipende da quanto la relazione è vincolante. Ad ogni tipo di relazione è dunque associato un peso.

Attraverso un duplice calcolo, descritto più in dettaglio nella sezione 3.2, Artemis associa ad ogni coppia di classi due coefficienti di affinità: il *Name Affinity Coefficient* che rappresenta il grado di affinità secondo la dimensione semantica, e lo *Structural Affinity Coefficient* secondo la dimensione strutturale.

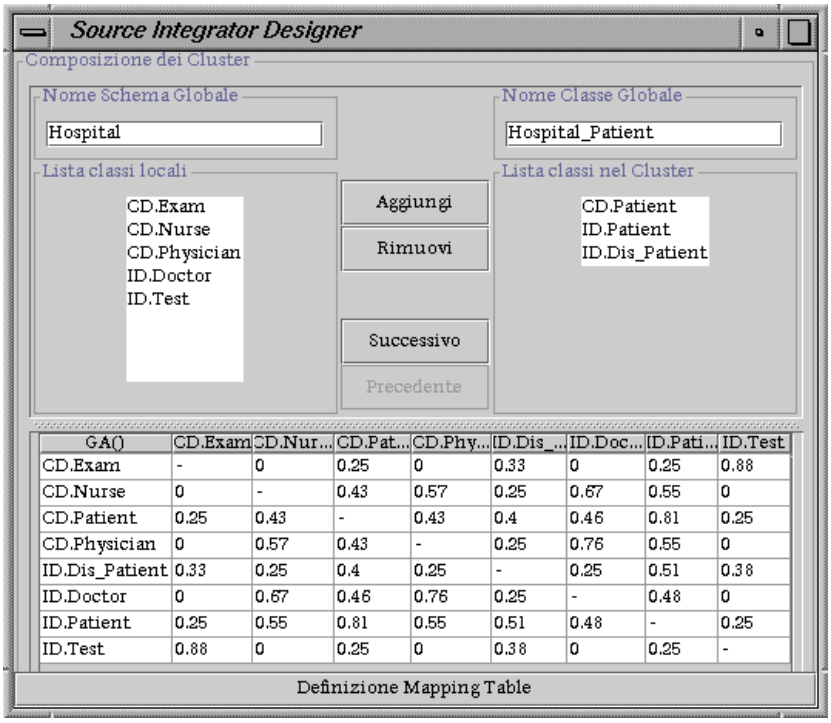
La media pesata dei due coefficienti fornisce il *Global Affinity Coefficient*, che rappresenta il vero parametro di affinità.

## 5.3 Generazione dei Cluster

Terminato il calcolo delle affinità esistente per ogni coppia di classi, il raggruppamento in cluster si ottiene attraverso un procedimento iterativo di progressiva riduzione della tabella dei coefficienti di affinità.
L'algoritmo di clusterizzazione è descritto in dettaglio nella sezione 3.3; l'unico dato che il progettista deve fornire è il valore di soglia che rappresenta il criterio di raggruppamento delle classi.

Attualmente **SI-Designer** delega al progettista la responsabilità di creare i Cluster. In figura 5.4 viene mostrata la schermata relativa a questa fase; la finestra è divisa in due settori:

- nella parte inferiore viene visualizzata una tabella contenente la matrice di affinità; nella prima riga e nella prima colonna sono scritti i nomi di tutte le




| GA() | CD.Exam | CD.Nur... | CD.Pat... | CD.Phy... | ID.Dis_... | ID.Doc... | ID.Pati... | ID.Test |
|---|---|---|---|---|---|---|---|---|
| CD.Exam | - | 0 | 0.25 | 0 | 0.33 | 0 | 0.25 | 0.88 |
| CD.Nurse | 0 | - | 0.43 | 0.57 | 0.25 | 0.67 | 0.55 | 0 |
| CD.Patient | 0.25 | 0.43 | - | 0.43 | 0.4 | 0.46 | 0.81 | 0.25 |
| CD.Physician | 0 | 0.57 | 0.43 | - | 0.25 | 0.76 | 0.55 | 0 |
| ID.Dis_Patient | 0.33 | 0.25 | 0.4 | 0.25 | - | 0.25 | 0.51 | 0.38 |
| ID.Doctor | 0 | 0.67 | 0.46 | 0.76 | 0.25 | - | 0.48 | 0 |
| ID.Patient | 0.25 | 0.55 | 0.81 | 0.55 | 0.51 | 0.48 | - | 0.25 |
| ID.Test | 0.88 | 0 | 0.25 | 0 | 0.38 | 0 | 0.25 | - |

Figura 5.4: Composizione dei cluster

classi locali: ogni cella, incrocio di una riga e di una colonna, contiene il *Global Affinity Coefficient* tra le due classi interessate. Poiché il coefficiente di affinità gode della proprietà di simmetria ($GA(c_i, c_j) = GA(c_j, c_i)$), la matrice è a sua volta simmetrica; inoltre non ha significato il coefficiente di affinità di una classe locale con se stessa, di conseguenza la diagonale principale della matrice non contiene valori.[1]

- nel settore superiore sono contenute due liste: quella a sinistra contiene l'elenco generale di tutte le classi locali, mentre quella di destra, che rappresenta il primo cluster, è inizialmente vuota.
  Il progettista ha il compito di comporre manualmente i singoli cluster: per prima cosa deve dare un nome alla classe globale associata, scrivendolo nel campo editabile posto sopra alla lista di destra; successivamente seleziona una classe dalla lista di sinistra e la sposta nel cluster premendo il pulsante *Aggiungi*. La pressione del pulsante *Rimuovi* toglie una classe dal cluster e la torna a mettere nella lista generale.

[1]Per comodità la matrice di affinità viene rappresentata come descritto, anche se sarebbe sufficiente una matrice triangolare.

I pulsanti *Successivo* e *Precedente* consentono di passare da un cluster all'altro: pertanto il contenuto della parte destra di questo settore inferiore rimane continuamente allineato al cluster corrente, mentre la tabella di affinità e la lista generale delle classi restano sempre visibili. Procedendo in questo modo, man mano che vengono composti i cluster, la lista generale si svuota progressivamente; quando è completamente vuota i cluster sono formati, e si attiva il pulsante *Definizione Mapping Table* che consente di passare alla fase successiva.

## 5.4 Definizione della Mapping Table

Per completare lo schema globale, a questo punto, manca solo la definizione della *Mapping Table*, cioè della collezione di tabelle, una per classe globale, contenenti il mapping fra grandezze globali e grandezze locali. Il processo relativo a questa fase è semiautomatico: il sistema, in base alla composizione dei cluster e alle informazioni contenute nel thesaurus (in particolare quelle fra attributi) genera gli attributi globali, secondo quanto visto nella sezione. 3.4. Quella che viene visualizzata a questo livello è la mapping table sulla quale è auspicabile che il progettista effettui una revisione, trattata nei particolari nella sezione 3.5.

Anche in questo caso, tuttavia, non esiste ancora nel sistema Momis il componente software che svolga la parte automatica relativa a questa fase. Come nel passo precedente, pertanto, il componente **SI-Designer** affida al progettista l'onere di costruire le tabella di mapping, assistendolo nelle operazioni che questa fase comporta.

La finestra che compare a video (figura 5.5) è composta da varie cartelle (*Tabbed Pane*), una per ogni cluster, all'interno di ognuna delle quali è contenuta una tabella e un'area di testo.
Inizialmente la tabella è compilata soltanto nella sua prima colonna: il primo elemento è il nome della classe globale associata al cluster visualizzato, mentre le altre celle della colonna contengono i nomi delle classi locali del cluster.
Nella prima riga, accessibile in scrittura, devono essere inseriti i nomi di tutti gli attributi globali appartenenti alla classe.

Le celle rappresentano il mapping che l'attributo globale (indice di colonna) ha all'interno della classe locale (indice di riga). Non è consentito al progettista scrivere direttamente nelle celle: per creare un mapping il progettista seleziona la cella interessata e scrive nell'area sottostante, in linguaggio $\text{ODL}_{I^3}$, la parte di sintassi relativa a quel mapping specifico.
Poiché le celle sono troppo piccole per contenere tutte le informazioni sul mapping, il sistema si preoccupa semplicemente di marcare quelle per le quali

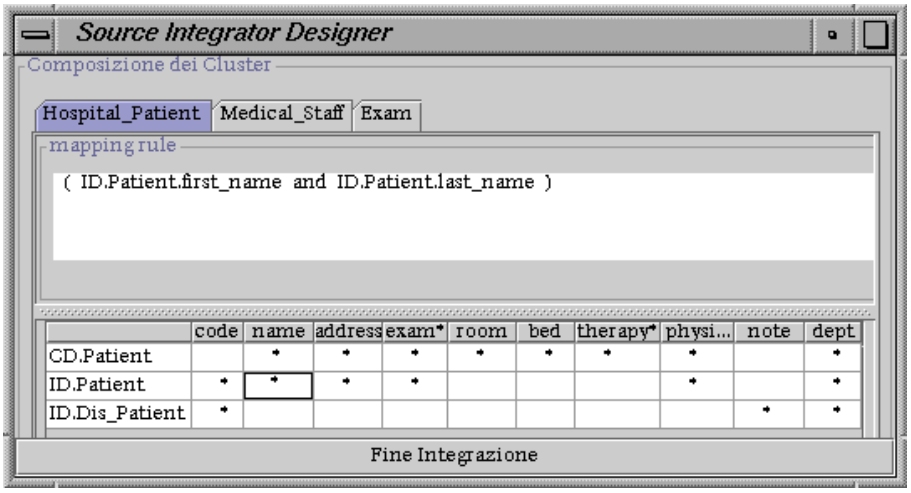



Figura 5.5: Compilazione della *Mapping Table*

è stata fissata una corrispondenza fra attributo globale e classe locale. Se il progettista seleziona una cella già marcata, l'area sottostante mostra il testo scritto in precedenza, che diventa così modificabile. L'operazione manuale appena descritta deve essere ripetuta per tutte le cartelle.

Completate tutte le tabelle il progettista preme il pulsante *Fine*. A questo punto il componente **SI-Designer** effettua le seguenti operazioni:

1. Genera, in linguaggio ODL$_{I^3}$, la descrizione dello schema globale;
2. manda in esecuzione il parser, che controlla la correttezza sintattica e semantica delle dichiarazioni di mapping inserite precedentemente dal progettista;
3. interroga la struttura dati caricata dal parser, preleva il contenuto informativo relativo alle grandezze globali e lo trasferisce nella struttura dati, appositamente creata dal query manager per rappresentare in memoria centrale le mapping table.

### 5.4.1 Generazione dello schema globale in linguaggio ODL$_{I^3}$

La descrizione di una generica classe globale è così fatta:

```
interface nome_classe_globale {
    attribute tipo1  nome_attributo_globale1
```

```
            mapping_rule
                 mapping_rule_con_una_classe_locale,
                 ...
                 mapping_rule_con_una_classe_locale;

          attribute tipo2 nome_attributo_globale2
            mapping_rule
                 ...
    }
```

**Procedure Descrizione_schema_globale**

Crea e apri in scrittura un nuovo file 'Global.odl'.
**for** ogni tabella {
  Appendi la stringa "`interface` *nome_classe_globale* {"
  Appendi la stringa "`( source object` *nome_schema_globale* `)`"
  **for** ogni colonna della tabella {
    Appendi la stringa "`attribute` *tipo_attributo_globale nome_attributo_globale*"
    Appendi la stringa "`mapping_rule`"
    **for** ogni elemento non vuoto della colonna {
      Appendi la mapping rule
      **if** ci sono ancora elementi non vuoti
        **then** appendi il carattere '`,`'
    }
    Appendi il carattere '`;`'
  }
  Appendi il carattere '}'
  **if** ci sono ancora tabelle
    **then** appendi il carattere '`;`'
}
Appendi il carattere '`.`'
**end procedure.**

Figura 5.6: Procedura di clustering

Il tipo di ogni attributo globale è fornito dal progettista, così come le stringhe che rappresentano le mapping rule con le classi globali, di conseguenza la descrizione dello schema globale si ottiene concatenando opportunamente le singole stringhe inserite dal progettista. L'algoritmo che descrive questa operazione è mostrato in figura 5.6.

### 5.4.2 Esecuzione del Parser

L'analisi, da parte del parser, del file ODL$_{I^3}$ appena generato, è un'operazione necessaria al fine di garantire la coerenza delle informazioni in esso contenute e la corretta sintassi. Mi riferisco in particolare alle regole di mapping scritte manualmente dal progettista.

A questo proposito il Parser verifica l'esistenza nei sorgenti di tutti gli attributi locali, sapendo che ognuno è denotato dalla sintassi *NomeSorgente.NomeClasse.nomeAttributo*. Inoltre viene verificato ogni riferimento a classi globali: il caso tipico riguarda la definizione di un attributo complesso che mappa, appunto, su una classe.

### 5.4.3 Caricamento della struttura dati della Mapping Table

La struttura dati utilizzata dal Parser, ampiamente spiegata nella sezione 4.4, è stata progettata con l'obiettivo di contenere tutte le possibili informazioni che uno u più file ODL$_{I^3}$ possono esprimere. In questo ambito la sua architettura è del tutto generale, non orientata in modo specifico all'uno o all'altro aspetto.

Il Query Manager del sistema Momis [47], tuttora in fase di progettazione, basa la sua conoscenza sul mapping esistente fra le grandezze globali e le grandezze locali presenti nei singoli schemi sorgenti, poiché il suo compito consiste nella traduzione delle query poste dall'utente (che usufruisce dello schema integrato) scritte in termini di classi e attributi globali, in query rivolte ai database locali.
Affinché il lavoro di traduzione sia efficiente, il Query Manager deve poter disporre di una struttura dati efficiente che implementi le Mapping Table in memoria centrale. Questa struttura è illustrata in figura 5.7.

Ogni oggetto della classe **MappingTable** contiene le informazioni su tutte le regole di mapping di una classe globale, nei seguenti attributi:

- *globalSchema*: stringa che contiene il nome dello schema globale;
- *gClassName*: nome della classe globale che l'oggetto rappresenta;
- *gAttributes*: elenco di oggetti **GlobalAttribute**. Nella rappresentazione tabellare sono i titoli delle colonne;
- *localClasses*: elenco di oggetti **LocalClass**. Nella rappresentazione tabellare sono i titoli delle righe;

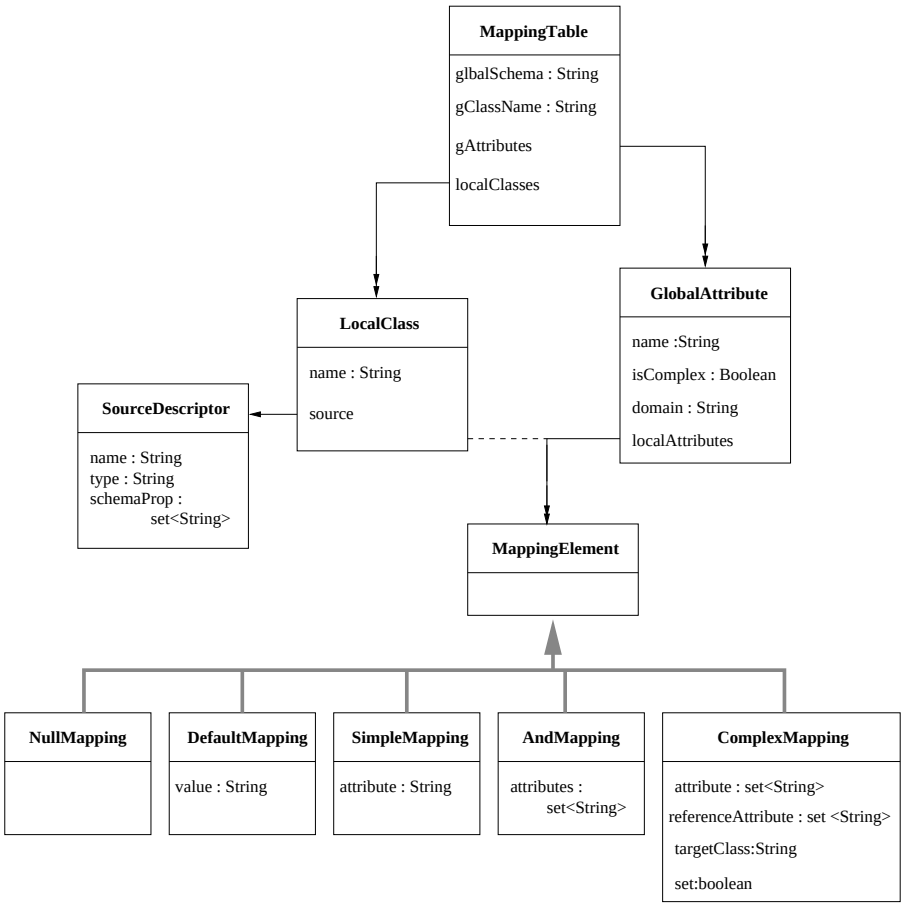



Figura 5.7: Modello a oggetti della Mapping Table

Gli oggetti **LocalClass** contengono i dati sulle classi locali utili al Query Manager, e cioè il nome e il sorgente, a sua volta descritto da **SourceDescriptor**, contenente nome e tipo del sorgente, nonché il set di tutti gli attributi di tutte le classi dello schema locale;

Gli oggetti **GlobalAttribute** contengono i dati sugli attributi globali, quali nome e dominio; l'attributo booleano *isComplex* dice se il dominio di questo attributo è un tipo-valore o un tipo-classe: in quest'ultimo caso *domain* contiene il nome di una classe globale.
I dati sui mapping sono contenuti negli oggetti **MappingElement**, raggiungibili attraverso l'attributo *localAttributes* che ne contiene un l'insieme relativo all'attributo globale di cui fa parte. I diversi mapping di un attributo globale si distinguono in base a un valore chiave dato dal nome della classe locale cui sono indirizzati. Per questo motivo, pur non essendoci connessione fra **LocalClass** e

**MappingElement**, nello schema compare una linea tratteggiata.

Ogni oggetto **MappingElement** può essere:

- **NullMapping** se non c'è mapping fra attributo globale e classe locale;
- **DefaultMapping** se l'attributo globale assume, nella classe locale, un valore di default, contenuto in *value*;
- **SimpleMapping** se il mapping è semplice: in tal caso è sufficiente conoscere il nome dell'attributo locale;
- **AndMapping** se il mapping riguarda più attributi della classe locale in questione: i loro nomi sono contenuti nel set di stringhe *attributes*;
- **ComplexMapping** se il mapping avviene con un attributo locale che realizza aggregazione: notare che se il sorgente è relazionale, l'aggregazione si manifesta mediante una foreign key, che può essere composta da una lista di attributi locali.
  - *set* è un attributo booleano che indica se l'aggregazione è data da un unico attributo locale (è il caso di un sorgente a oggetti o di una foreign key semplice) o da una lista di attributi (quando la foreign key è composta);
  - *attribute* contiene l'insieme degli attributi locali (al limite uno solo) coinvolti nell'aggregazione;
  - *referenceAttribute* contiene lo stesso insieme di attributi, così come si chiamano nella classe mappata. Nel caso di sorgente a oggetti questo campo è vuoto;
  - *targetClass* è il nome della classe locale target nella gerarchia di aggregazione.

## 5.5 Il software

**SI-Designer** è stato realizzato in linguaggio java, versione 1.2beta4 [50]. Per l'interfaccia grafica ho fatto uso dei package *Swing Components*, disponibili a partire dal JDK 1.2.

Le classi che compongono **SI-Designer** sono state raggruppate in un pacchetto di nome *SIDes*; il main è contenuto nel file *SIDesigner.java*, accessibile all'indirizzo *http://sparc20.dsi.unimo.it/tesi/index.html*.

# Conclusioni

In questa tesi è stato descritto il progetto di un sistema mediatore, MOMIS, il cui compito consiste nel fornire accesso integrato ad una molteplicità di sorgenti eterogenee di informazioni. In particolare, MOMIS va a collocarsi all'interno di una vasta area di ricerca, l'Integrazione Intelligente di Informazioni ($I^3$) che, pur essendo piuttosto recente, si pone come obiettivo l'utilizzo di tecniche ampiamente consolidate di Intelligenza Artificiale nell'ambito dell'integrazione dei dati.

Un aspetto positivo del mediatore MOMIS è dato dalla capacità che il sistema ha di assistere il progettista durante la fase di integrazione degli schemi. Non potendo infatti prescindere dalla conoscenza che il progettista stesso possiede, da un lato l'uso del database WordNet permette di suggerire la conoscenza semantica necessaria alla determinazione delle entità affini, limitando il compito manuale alla sola integrazione delle relazioni non automaticamente estratte, dall'altro l'uso di tecniche di Intelligenza Artificiale fornite dagli ODB-Tools consente di inferire automaticamente la conoscenza logicamente implicata da quella già acquisita. In questo modo, pur rimanendo inalterata la responsabilità propria del progettista, nei limiti del possibile il suo lavoro manuale viene minimizzato.

L' aspetto teorico del progetto MOMIS, nella parte che riguarda le fasi della costruzione dello schema globale, è stato definito prima che iniziasse il lavoro svolto in questa tesi: il contributo teorico offerto dal sottoscritto ha riguardato l'approfondimento della fase automatica di generazione della Mapping Table, in particolare l'unione ragionata degli attributi e il criterio di integrazione di quelli che realizzano aggregazione, posti in relazione semantica ma non validati.
Dal punto di vista del software, sono state realizzate precedentemente (ed indipendentemente l'una dall'altra) le fasi di costruzione del thesaurus (modulo $\text{SIM}_1$ [25] che si avvale degli ODB-Tools) e il calcolo delle affinità (modulo ESCA [45]): il mio lavoro è consistito nella realizzazione dell'interfaccia utente del Global Schema Builder, che ha anche il compito di coordinare l'esecuzione in successione dei moduli citati, nonché di interagire con il database lessicale WordNet. Infine è stato realizzato il componente *parser* del linguaggio $\text{ODL}_{I^3}$.
Attualmente, al completamento del Mediatore, manca la parte di Artemis che realizza la clusterizzazione e, dati i cluster, il calcolo automatico di definizione dello schema globale. Per quanto riguarda il gestore delle interrogazioni, (le cui problematiche sono state poco più che accennate), nell'ambito del progetto

MOMIS è stata svolta, parallelamente a questa tesi, una analisi approfondita ed è stato realizzato un prototipo di Query Manager [47].
Terminato il Mediatore, in futuro sarà necessario affrontare gli aspetti che riguardano i wrapper, posto a livello inferiore nell'architettura $I^3$, nonché effettuare la progettazione e realizzazione degli stessi.

L'apprendimento di Java, considerato attualmente uno dei più importanti standard nell'ambito della programmazione ad oggetti, ha comportato non poche difficoltà iniziali, soprattutto a causa della quasi totale mancanza di conoscenza di questo tipo di programmazione. Tuttavia ritengo positivo l'approccio avuto con questo linguaggio, pur rimanendo, allo stato attuale, la mia capacità di programmazione limitata alle conoscenze di base.
Considero infine estremamente formativa l'esperienza che ha portato a questo lavoro di tesi, non solo a causa dell'importanza e della grande attualità dei temi trattati, ma anche e soprattutto perché scaturita in un ambiente didattico sereno, dove ho riscontrato uno spirito di grande collaborazione da parte di chi aveva già un ottimo *know-how* degli argomenti in esame, e di chi ha condiviso con me questa nuova esperienza.

# Appendice A

# Glossario $I^3$

Questo glossario ed il vocabolario sul quale si basa sono stati originariamente sviluppati durante l'$I^3$ Architecture Meeting in Boulder CO, 1994, sponsorizzato dall'ARPA, e rifiniti in un secondo incontro presso l'Università di Stanford, nel 1995. Il glossario è strutturato logicamente in diverse sezioni:

- Sezione 1: Architettura
- Sezione 2: Servizi
- Sezione 3: Risorse
- Sezione 4: Ontologie

Nota: poiché la versione originaria del glossario usa una terminologia inglese, in alcuni casi è riportato, a fianco del termine, il corrispettivo inglese, quando la traduzione dal termine originale all'italiano poteva essere ambigua o poco efficace.

## A.1 Architettura

- Architettura = insieme di componenti.
- architettura di riferimento = linea guida ed insieme di regole da seguire per l'architettura.
- componente = uno dei blocchi sui quali si basa una applicazione o una configurazione. Incorpora strumenti e conoscenza specifica del dominio.
- applicazione = configurazione persistente o transitoria dei componenti, rivolta a risolvere un problema del cliente, e che può coprire diversi domini.

- configurazione = istanza particolare di una architettura per una applicazione o un cliente.
- collante (glue) = software o regole che servono per per collegare i componenti o per interoperare attraverso i domini.
- strato = grossolana categorizzazione dei componenti e degli strumenti in una configurazione. L'architettura $I^3$ distingue tre strati, ognuno dei quali fornisce una diversa categoria di servizi:
  1. Servizi di Coordinamento = coprono le fasi di scoperta delle risorse, distribuzione delle risorse, invocazione, scheduling . . .
  2. Servizi di Mediazione = coprono la fase di query processing e di trattamento dei risultati, nonché il filtraggio dei dati, la generazione di nuove informazioni, etc.
  3. Servizi di Wrapping = servono per l'utilizzo dei wrappers e degli altri strumenti simili utilizzati per adattarsi a standards di accesso ai dati e alle convenzioni adoperate per la mediazione e per il coordinamento.
- agente = strumento che realizza un servizio, sia per il suo proprietario, sia per un cliente del suo proprietario.
- facilitatore = componente che fornisce i servizi di coordinamento, come pure l'instradamento delle interrogazioni del cliente.
- mediatore = componente che fornisce i servizi di mediazione e che provvede a dare valore aggiunto alle informazioni che sono trasmesse al cliente in risposta ad una interrogazione.
- cliente (customer) = proprietario dell'applicazione che gestisce le interrogazioni, o utente finale, che usufruisce dei servizi.
- risorsa = base di dati accessibile, server ad oggetti, base di conoscenze . . .
- contenuto = risultato informativo ricavato da una sorgente.
- servizio = funzione fornita da uno strumento in un componente e diretta ad un cliente, direttamente od indirettamente.
- strumento (tool) = programma software che realizza un servizio, tipicamente indipendentemente dal dominio.
- wrapper = strumento utilizzato per accedere alle risorse conosciute, e per tradurre i suoi oggetti.

- regole limitative (constraint rules) = definizione di regole per l'assegnamento di componenti o di protocolli a determinati strati.
- interoperare = combinare sorgenti e domini multipli.
- informazione = dato utile ad un cliente.
- informazione azionabile = informazione che forza il cliente ad iniziare un evento.
- dato = registrazione di un fatto.
- testo = dato, informazione o conoscenza in un formato relativamente non strutturato, basato sui caratteri.
- conoscenza = metadata, relazione tra termini, paradigmi . . . , utili per trasformare i dati in informazioni.
- dominio = area, argomento, caratterizzato da una semantica interna, per esempio la finanza, o i componenti elettronici . . .
- metadata = informazione descrittiva relativa ai dati di una risorsa, compresi il dominio, proprietà, le restrizioni, il modello di dati, . . .
- metaconoscenza = informazione descrittiva relativa alla conoscenza in una risorsa, includendo l'ontologia, la rappresentazione . . .
- metainformazioni = informazione descritiiva sui servizi, sulle capacità, sui costi . . .

## A.2 Servizi

- Servizio = funzionalità fornita da uno o più componenti, diretta ad un cliente.
- instradamento (routing) = servizio di coordinamento per localizzare ed invocare una risorsa o un servizio di mediazione, o per creare una configurazione. Fa uso di un direttorio.
- scheduling = servizio di coordinamento per determinare l'ordine di invocazione degli accessi e di altri servizi; fa spesso uso dei costi stimati.
- accoppiamento (matchmaking) = servizio che accoppia i sottoscrittori di un servizio ai fornitori.

- intermediazione (brokering) = servizio di coordinamento per localizzare le risorse migliori.
- strumento di configurazione = programma usato nel coordinamento per aiutare a selezionare ed organizzare i componenti in una istanza particolare di una configurazione architetturale.
- servizi di descrizione = metaservizi che informano i clienti sui servizi, risorse . . .
- direttorio = servizio per localizzare e contattare le risorse disponibili, come le pagine gialle, pagine bianche . . .
- decomposizione dell'interrogazione (query decomposition) = determina le interrogazioni da spedire alle risorse o ai servizi disponibili.
- riformulazione dell'interrogazione (query reformulation) = programma per ottimizzare o rilassare le interrogazioni, tipicamente fa uso dello scheduling.
- contenuto = risultato prodotto da una risorsa in risposta ad interrogazioni.
- trattamento del contenuto (content processing) = servizio di mediazione che manipola i risultati ottenuti, tipicamente per incrementare il valore delle informazioni.
- trattamento del testo = servizio di mediazione che opera sul testo per ricerca, correzione . . .
- filtraggio = servizio di mediazione per aumentare la pertinenza delle informazioni ricevute in risposta ad interrogazioni.
- classificazione (ranking) = servizio di mediazione per assegnare dei valori agli oggetti ritrovati.
- spiegazione = servizio di mediazione per presentare i modelli ai clienti.
- amministrazione del modello = servizio di mediazione per permettere al cliente ed al proprietario del mediatore di aggiornare il modello.
- integrazione = servizio di mediazione che combina i contenuti ricevuti da una molteplicità di risorse, spesso eteorgenee.
- accoppiamento temporale = servizio di mediazione per riconoscere e risolvere differenze nelle unità di misura temporali utilizzate dalle risorse.

- accoppiamento spaziale = servizio di mediazione per riconoscere e risolvere differenze nelle unità di misura spaziali utilizzate dalle risorse.
- ragionamento (reasoning) = metodologia usata da alcuni componenti o servizi per realizzare inferenze logiche.
- browsing = servizio per permettere al cliente di spostarsi attraverso le risorse.
- scoperta delle risorse = servizio che ricerca le risorse.
- indicizzazione = creazione di una lista di oggetti (indice) per aumentare la velocità dei servizi di accesso.
- analisi del contenuto = trattamento degli oggetti testuali per creare informazioni.
- accesso = collegamento agli oggetti nelle risorse per realizzare interrogazioni, analisi o aggiornamenti.
- ottimizzazione = processo di manipolazione o di riorganizzazione delle interrogazioni per ridurne il costo o il tempo di risposta.
- rilassamento = servizio che fornisce un insieme di risposta maggiore rispetto a quello che l'interrogazione voleva selezionare.
- astrazione = servizio per ridurre le dimensioni del contenuto portandolo ad un livello superiore.
- pubblicità (advertising) = presentazione del modello di una risorsa o del mediatore ad un componente o ad un cliente.
- sottoscrizione = richiesta di un componente o di un cliente di essere informato su un evento.
- controllo (monitoring) = osservazione delle risorse o dei dati virtuali e creazione di impulsi da azionare ogniqualvolta avvenga un cambiamento di stato.
- aggiornamento = trasmissione dei cambiamenti dei dati alle risorse.
- instanziazione del mediatore = popolamento di uno strumento indipendente dal dominio con conoscenze dipendenti da un dominio.
- attivo (activeness) = abilità di un impulso di reagire ad un evento.

- servizio di transazione = servizio che assicura la consistenza temporale dei contenuti, realizzato attraverso l'amministrazione delle transzioni.
- accertamento dell'impatto = servizio che riporta quali risorse saranno interessate dalle interrogazioni o dagli aggiornamenti.
- stimatore = servizo di basso livello che stima i costi previsti e le prestazioni basandosi su un modello, o su statistiche.
- caching = mantenere le informazioni memorizzate in un livello intermedio per migliorare le prestazioni.
- traduzione = trasformazione dei dati nella forma e nella sintassi richiesta dal ricevente.
- controllo della concorrenza = assicurazione del sincronismo degli aggiornamenti delle risorse, tipicamente assegnato al sistema che amministra le transazioni.

## A.3 Risorse

- Risorsa = base di dati accessibile, simulazione, base di conoscenza, ... comprese le risorse "legacy".
- risorse "legacy" = risorse preesistenti o autonome, non disegnate per interoperare con una architettura generale e flessibile.
- evento = ragione per il cambiamento di stato all'interno di un componente o di una risorsa.
- oggetto = istanza particolare appartenente ad una risorsa, al modello del cliente, o ad un certo strumento.
- valore = contenuto metrico presente nel modello del cliente, come qualità, rilevanza, costo.
- proprietario = individuo o organizzazione che ha creato, o ha i diritti di un oggetto, e lo può sfruttare.
- proprietario di un servizio = individuo o organizzazione responsabile di un servizio.
- database = risorsa che comprende un insieme di dati con uno schema descrittivo.

- warehouse = database che contiene o dà accesso a dati selezionati, astratti e integrati da una molteplicità di sorgenti. Tipicamente ridondante rispetto alle sorgenti di dati.
- base di conoscenza = risorsa comprendente un insieme di conoscenze trattabili in modo automatico, spesso nella forma di regole e di metadata; permettono l'accesso alle risorse.
- simulazione = risorsa in grado di fare proiezioni future sui dati e generare nuove informazioni, basata su un modello.
- amministrazione della transazione = assicurare che la consistenza temporale del database non sia compromessa dagli aggiornamenti.
- impatto della transazione = riporta le risorse che sono state coinvolte in un aggiornamento.
- schema = lista dedlle relazioni, degli attributi e, quando possibile, degli oggetti, delle regole, e dei metadata di un database. Costituisce la base dell'ontologia della risorsa.
- dizionario = lista dei termini, fa parte dell'ontologia.
- modello del database = descrizione formalizzata della risorsa database, che include lo schema.
- interoperabilità = capacità di interoperare.
- eterogeneità = incompatibilità trovate tra risorse e servizi sviluppati autonamamente, che vanno dalla paittaforma utilizzata, sistema operativo, modello dei dati, alla semantica, ontologia, . . .
- costo = prezzo per fornire un servizio o un accesso ad un oggetto.
- database deduttivo = database in grado di utilizzare regole logiche per trattare i dati.
- regola = affermazione logica, unità della conoscenza trattabile in modo automatico.
- sistema di amministrazione delle regole = software indipendente dal dominio che raccoglie, seleziona ed agisce sulle regole.
- database attivo = database in grado di reagire a determinati eventi.
- dato virtuale = dato rappresentato attraverso referenze e procedure.

- stato = istanza o versione di una base di dati o informazioni.
- cambiamento di stato = stato successivo ad una azione di aggiornamento, inserimento o cancellazione.
- vista = sottoinsieme di un database, sottoposto a limiti, e ristrutturato.
- server di oggetti = fornisce dati oggetto.
- gerarchia = struttura di un modello che assegna ogni oggetto ad un livello, e definisce per ogni oggetto l'oggetto da cui deriva.
- network = struttura di un modello che fa uso di relazioni relativamente libere tra oggetti.
- ristrutturare = dare una struttura diversa ai dati seguendo un modello differente dall'originale.
- livello = categorizzazione concettuale , dove gli oggetti di un livello inferiore dipendono da un antenato di livello superiore.
- antenato (ancestor) = oggetto di livello superiore, dal quale derivano attributi ereditabili.
- oggetto root = oggetto da cui tutti gli altri derivano, all'interno di una gerarchia.
- datawarehouse = deposito di dati integrati provenienti da una molteplicità di risorse.
- deposito di metadata = database che contiene metadata o metainformazioni.

## A.4 Ontologia

- Ontologia = descrizione particolareggiata di una concettualizzazione, i.e. l'insieme dei termini e delle relazioni usate in un dominio, per indicare oggetti e concetti, spesso ambigui tra domini diversi.
- concetto = definisce una astrazione o una aggregazione di oggetti per il cliente.
- semantico = che si riferisce al significato di un termine, espresso come un insieme di relazioni.

- sintattico = che si riferisce al formato di un termine, espresso come un insieme di limitazioni.
- classe = definisce metaconoscenze come metodi, attributi, ereditarietà, per gli oggetti in essa istanziati.
- relazione = collegamento tra termini, come *is-a*, *part-of*, . . .
- ontologia unita (merged) = ontologia creata combinando diverse ontologie, ottenuta mettendole in relazione tra loro (mapping).
- ontologia condivisa = sottoinsieme di diverse ontologie condiviso da una molteplicità di utenti.
- comparatore di ontologie = strumento per determinare relazioni tra ontologie, utilizzato per determinare le regole necessarie per la loro integrazione.
- mapping tra ontologie = trasformazione dei termini tra le ontologie, attraverso regole di accoppiamento, utilizzato per collegare utenti e risorse.
- regole di accoppiamento (matching rules) = dichiarazioni per definire l'equivalenza tra termini di domini diversi.
- trasformazione dello schema = adattamento dello schema ad un'altra ontologia.
- editing = trattamneto di un testo per assicurarne la conformità ad una ontologia.
- algebra dell'ontologia = insieme delle operazioni per definire relazioni tra ontologie.
- consistenza temporale = è raggiunta se tutti i dati si riferiscono alla stessa istanza temporale ed utilizzano la stessa granularità temporale.
- specifico ad un dominio = relativo ad un singolo dominio, presuppone l'assenza di incompatibilità semantiche.
- indipendente dal dominio = software, strumento o conoscenza globale applicabile ad una molteplicità di domini.

# Appendice B

# Il linguaggio ODL dello standard ODMG-93

L'appendice riporta la descrizione della BNF del linguaggio ODL dello standard ODMG-93 descritta in [28].

| | | |
|---|---|---|
| ⟨specification⟩ | : : = | ⟨definition⟩ |
| | | \| ⟨definition⟩ ⟨specification⟩ |
| ⟨definition⟩ | : : = | ⟨type_dcl⟩ **;** |
| | | \| ⟨const_dcl⟩ **;** |
| | | \| ⟨except_dcl⟩ **;** |
| | | \| ⟨interface_dcl⟩ **;** |
| | | \| ⟨module⟩ **;** |
| ⟨module⟩ | : : = | **module** ⟨identifier⟩ { ⟨specification⟩ } |
| ⟨interface⟩ | : : = | ⟨interface_dcl⟩ |
| | | \| ⟨forward_dcl⟩ |
| ⟨interface_dcl⟩ | : : = | ⟨interface_header⟩ |
| | | [**:**⟨persistence_dcl⟩] {[⟨interface_body⟩]} |
| ⟨persistence_dcl⟩ | : : = | **persistent** \| **transient** |
| ⟨forward_dcl⟩ | : : = | **interface** ⟨identifier⟩ |
| ⟨interface_header⟩ | : : = | **interface** ⟨identifier⟩ |
| | | [⟨inheritance_spec⟩] |
| | | [⟨type_property_list⟩] |
| ⟨type_property_list⟩ | : : = | **(** [⟨extent_spec⟩] [⟨key_spec⟩] **)** |
| ⟨extent_spec⟩ | : : = | **extent** ⟨string⟩ |
| ⟨key_spec⟩ | : : = | **key**[**s**] ⟨key_list⟩ |

⟨key_list⟩ : : = ⟨key⟩ | ⟨key⟩ **,** ⟨key_list⟩
⟨key⟩ : : = ⟨property_name⟩ | **(**⟨property_list⟩**)**
⟨property_list⟩ : : = ⟨property_name⟩
| ⟨property_name⟩ **,** ⟨property_list⟩
⟨property_name⟩ : : = ⟨identifier⟩
⟨interface_body⟩ : : = ⟨export⟩ | ⟨export⟩ ⟨interface_body⟩
⟨export⟩ : : = ⟨type_dcl⟩ **;**
| ⟨const_dcl⟩ **;**
| ⟨except_dcl⟩ **;**
| ⟨attr_dcl⟩ **;**
| ⟨rel_dcl⟩ **;**
| ⟨op_dcl⟩ **;**
⟨inheritance_spec⟩ : : = **:**⟨scoped_name⟩ [**,**⟨inheritance_spec⟩]
⟨scoped_name⟩ : : = ⟨identifier⟩
| **::**⟨identifier⟩
| ⟨scoped_name⟩**::**⟨identifier⟩
⟨const_dcl⟩ : : = **const** ⟨const_type⟩ ⟨identifier⟩=⟨const_exp⟩
⟨const_type⟩ : : = ⟨integer_type⟩
| ⟨char_type⟩
| ⟨boolean_type⟩
| ⟨floating_pt_type⟩
| ⟨string_type⟩
| ⟨scoped_name⟩
⟨const_exp⟩ : : = ⟨or_expr⟩
⟨or_expr⟩ : : = ⟨xor_expr⟩
| ⟨or_expr⟩ | ⟨xor_expr⟩
⟨xor_expr⟩ : : = ⟨and_expr⟩
| ⟨xor_expr⟩ ^ ⟨and_expr⟩
⟨and_expr⟩ : : = ⟨shift_expr⟩
| ⟨and_expr⟩ **&** ⟨shift_expr⟩
⟨shift_expr⟩ : : = ⟨add_expr⟩
| ⟨shift_expr⟩ >> ⟨add_expr⟩
| ⟨shift_expr⟩ << ⟨add_expr⟩
⟨add_expr⟩ : : = ⟨mult_expr⟩
| ⟨add_expr⟩ **+** ⟨mult_expr⟩
| ⟨add_expr⟩ **-** ⟨mult_expr⟩
⟨mult_expr⟩ : : = ⟨unary_expr⟩
| ⟨mult_expr⟩ ***** ⟨unary_expr⟩
| ⟨mult_expr⟩ **/** ⟨unary_expr⟩
| ⟨mult_expr⟩ **%** ⟨unary_expr⟩
⟨unary_expr⟩ : : = ⟨primary_expr⟩
| ⟨unary_operator⟩ | ⟨primary_expr⟩
⟨unary_operator⟩ : : = **-** | **+** | ~

⟨primary_expr⟩ ::= ⟨scoped_name⟩
| ⟨literal⟩
| **(**| ⟨const_exp⟩**)**

⟨literal⟩ ::= ⟨integer_literal⟩
| ⟨string_literal⟩
| ⟨character_literal⟩
| ⟨floating_pt_literal⟩
| ⟨boolean_literal⟩

⟨boolean_literal⟩ ::= **TRUE** | **FALSE**

⟨positive_int_const⟩ ::= ⟨const_exp⟩

⟨type_dcl⟩ ::= **typedef** ⟨type_declarator⟩
| ⟨struct_type⟩
| ⟨union_type⟩
| ⟨enum_type⟩

⟨type_declarator⟩ ::= ⟨type_spec⟩ ⟨declarators⟩

⟨type_spec⟩ ::= ⟨simple_type_spec⟩
| ⟨constr_type_spec⟩

⟨simple_type_spec⟩ ::= ⟨base_type_spec⟩
| ⟨template_type_spec⟩
| ⟨scoped_name⟩

⟨base_type_spec⟩ ::= ⟨floating_pt_type⟩
| ⟨integer_type⟩
| ⟨char_type⟩
| ⟨boolean_type⟩
| ⟨octet_type⟩
| ⟨any_type⟩

⟨template_type_spec⟩ ::= ⟨array_type⟩
| ⟨string_type⟩
| ⟨coll_type⟩

⟨coll_type⟩ ::= ⟨coll_spec⟩ **¡**⟨simple_type_spec⟩**¿**

⟨coll_spec⟩ ::= **set** | **list** | **bag**

⟨constr_type_spec⟩ ::= ⟨struct_type⟩
| ⟨union_type⟩
| ⟨enum_type⟩

⟨declarators⟩ ::= ⟨declarator⟩
| ⟨declarator⟩ **,** ⟨declarators⟩

⟨declarator⟩ ::= ⟨simple_declarator⟩
| ⟨complex_declarator⟩

⟨simple_declarator⟩ ::= ⟨identifier⟩

⟨complex_declarator⟩ ::= ⟨array_declarator⟩

⟨floating_pt_type⟩ ::= **float** | **double**

⟨integer_type⟩ ::= ⟨signed_int⟩
⟨unsigned_int⟩
⟨signed_int⟩ ::= ⟨signed_long_int⟩
⟨signed_short_int⟩
⟨signed_long_int⟩ ::= **long**
⟨signed_short_int⟩ ::= **short**
⟨unsigned_int⟩ ::= ⟨unsigned_long_int⟩
⟨unsigned_short_int⟩
⟨unsigned_long_int⟩ ::= **unsigned long**
⟨unsigned_short_int⟩ ::= **unsigned short**
⟨char_type⟩ ::= **char**
⟨boolean_type⟩ ::= **boolean**
⟨octet_type⟩ ::= **octet**
⟨any_type⟩ ::= **any**
⟨struct_type⟩ ::= **struct** ⟨identifier⟩ {⟨member_list⟩}
⟨member_list⟩ ::= ⟨member⟩
| ⟨member⟩ ⟨member_list⟩
⟨member⟩ ::= ⟨type_spec⟩ ⟨declarators⟩ **;**
⟨union_type⟩ ::= **union** ⟨identifier⟩ **switch**
**(**⟨switch_type_spec⟩**)** {⟨switch_body⟩}
⟨switch_type_spec⟩ ::= ⟨integer_type⟩
| ⟨char_type⟩
| ⟨boolean_type⟩
| ⟨enum_type⟩
| ⟨scoped_name⟩
⟨switch_body⟩ ::= ⟨case⟩ | ⟨case⟩ ⟨switch_body⟩
⟨case⟩ ::= ⟨case_label_list⟩ ⟨element_spec⟩ **;**
⟨case_label_list⟩ ::= ⟨case_label⟩
| ⟨case_label⟩ ⟨case_label_list⟩
⟨case_label⟩ ::= **case** ⟨const_exp⟩**:**
| **default:**
⟨element_spec⟩ ::= ⟨type_spec⟩ ⟨declarator⟩
⟨enum_type⟩ ::= **enum** ⟨identifier⟩ {⟨enumerator_list⟩}
⟨enumerator_list⟩ ::= ⟨enumerator⟩
| ⟨enumerator⟩**,**⟨enumerator_list⟩
⟨enumerator⟩ ::= ⟨identifier⟩
⟨array_type⟩ ::= ⟨array_spec⟩
<⟨simple_type_spec⟩**,**⟨positive_int_const⟩>
| ⟨array_spec⟩ <⟨simple_type_spec⟩>
⟨array_spec⟩ ::= **array** | **sequence**
⟨string_type⟩ ::= **string** | **string** ⟨positive_int_const⟩

⟨array_declarator⟩ ::= ⟨identifier⟩ ⟨array_size_list⟩
⟨array_size_list⟩ ::= ⟨fixed_array_size⟩
| ⟨fixed_array_size⟩ ⟨array_size_list⟩
⟨fixed_array_size⟩ ::= **[** ⟨positive_int_const⟩ **]**
⟨attr_dcl⟩ ::= [**readonly**] **attribute** ⟨domain_type⟩
⟨attribute_name⟩ [⟨fixed_array_size⟩]
⟨domain_type⟩ ::= ⟨simple_type_spec⟩
| ⟨struct_type⟩
| ⟨enum_type⟩
⟨rel_dcl⟩ ::= **relationship** ⟨target_of_path⟩ ⟨identifier⟩
**inverse** ⟨inverse_trasversal_path⟩
⟨target_of_path⟩ ::= ⟨identifier⟩
| ⟨rel_collection_type⟩ <⟨identifier⟩>
⟨inverse_trasversal_path⟩ ::= ⟨identifier⟩ **::** ⟨identifier⟩
⟨attribute_list⟩ ::= ⟨scoped_name⟩
| ⟨scoped_name⟩**,**⟨attribute_list⟩
⟨rel_collection_type⟩ ::= **set** | **list** | **bag** | **array**
⟨except_dcl⟩ ::= **exception** ⟨identifier⟩ {[⟨member_list⟩]}
⟨op_dcl⟩ ::= [⟨op_attribute⟩] ⟨op_type_spec⟩ ⟨identifier⟩
⟨parameter_dcls⟩ [⟨raises_expr⟩] [⟨context_expr⟩]
⟨op_attribute⟩ ::= **oneway**
⟨op_type_spec⟩ ::= ⟨simple_type_spec⟩ | **void**
⟨parameter_dcls⟩ ::= **(**[⟨param_dcl_list⟩]**)**
⟨param_dcl_list⟩ ::= ⟨param_dcl⟩
| ⟨param_dcl⟩**,**⟨param_dcl_list⟩
⟨param_dcl⟩ ::= ⟨param_attribute⟩ ⟨simple_type_spec⟩ ⟨declarator⟩
⟨param_attribute⟩ ::= **in** | **out** | **inout**
⟨raises_expr⟩ ::= **raises (**⟨scoped_name_list⟩**)**
⟨scoped_name_list⟩ ::= ⟨scoped_name⟩
| ⟨scoped_name⟩**,**⟨scoped_name_list⟩
⟨context_expr⟩ ::= **context (**⟨string_literal_list⟩**)**
⟨string_literal_list⟩ ::= ⟨string_literal⟩
| ⟨string_literal⟩**,**⟨string_literal_list⟩

# Appendice C

# Il linguaggio ODL$_{I^3}$

L'appendice riporta la descrizione della BNF del linguaggio ODL$_{I^3}$. Viene descritta solo la parte si sintassi relativa alle estensioni compiute rispetto alla grammatica ODL dello standard ODMG-93, presentata nell'appendice B.

⟨specification⟩ : : = ⟨definition⟩
| ⟨definition⟩ ⟨specification⟩
⟨definition⟩ : : = ⟨type_dcl⟩ **;**
| ⟨const_dcl⟩ **;**
| ⟨except_dcl⟩ **;**
| ⟨interface_dcl⟩ **;**
| ⟨relationships_dcl⟩ **;**
| ⟨rule_dcl⟩ **;**
| ⟨module⟩ **;**

Parte relativa alla definzione del tipo-valore range.

⟨base_type_spec⟩ : : = ⟨floating_pt_type⟩
| ⟨integer_type⟩
| ⟨char_type⟩
| ⟨boolean_type⟩
| ⟨octet_type⟩
| ⟨range_type⟩
| ⟨any_type⟩
⟨range_type⟩ : : = **range [** ⟨range_specifier⟩ **]**
⟨range_specifier⟩ : : = ⟨const_exp⟩ **,** ⟨const_exp⟩
| ⟨const_exp⟩ **, +inf**
| **-inf ,** ⟨const_exp⟩

Parte relativa alla specificazione delle classi, in cui è possibile definire più di un interface_body.

⟨interface_dcl⟩ : : = ⟨interface_header⟩ {[⟨interface_body⟩]
[**union** ⟨interface_body⟩]}
⟨interface_header⟩ : : = **interface** ⟨identifier⟩
[⟨inheritance_spec⟩]
[⟨type_property_list⟩]
⟨inheritance_spec⟩ : : = **:** ⟨scoped_name⟩ [ **,** ⟨inheritance_spec⟩]

Parte relativa alla specificazione delle classi, in cui occorre specificare il tipo e il nome della sorgente informativa.

⟨type_property_list⟩ : : = **(**[⟨source_spec⟩] [⟨extent_spec⟩] [⟨key_spec⟩] [⟨f_key_list⟩]**)**
⟨source_spec⟩ : : = **source** ⟨source_type⟩ ⟨source_name⟩
⟨source_type⟩ : : = **relational**|**nfrelational**|**object**|**file**|**semistructured**
⟨source_name⟩ : : = ⟨identifier⟩
⟨extent_spec⟩ : : = **extent** ⟨extent_list⟩
⟨extent_list⟩ : : = ⟨string⟩ | ⟨string⟩ **,** ⟨extent_list⟩
⟨key_spec⟩ : : = **key**[**s**] ⟨key_list⟩
⟨f_key_list⟩ : : = ⟨f_key_spec⟩ | ⟨f_key_spec⟩ **,** ⟨f_key_list⟩
⟨f_key_spec⟩ : : = **foreign_key (**⟨key_list⟩**) references** ⟨identifier⟩
[,**(**⟨key_list⟩**)**]
. . .

Regole di definizione del mapping tra attributi della classe globale dello schema del mediatore, ed i corrispondenti nelle sorgenti locali.

⟨attr_dcl⟩ : : = **[readonly] attribute**
⟨domain_type⟩ ⟨attribute_name⟩ **[*]**
[⟨fixed_array_size⟩] [⟨mapping_rule_dcl⟩]
⟨mapping_rule_dcl⟩ : : = **mapping_rule** ⟨rule_list⟩
⟨rule_list⟩ : : = ⟨rule⟩ | ⟨rule⟩ **,** ⟨rule_list⟩
⟨rule⟩ : : = ⟨local_attr_name⟩ | **‘**⟨identifier⟩**’**
⟨and_expression⟩ | ⟨union_expression⟩
⟨and_expression⟩ : : = **(** ⟨local_attr_name⟩ **and** ⟨and_list⟩ **)**
⟨and_list⟩ : : = ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **and** ⟨and_list⟩
⟨union_expression⟩ : : = **(** ⟨local_attr_name⟩ **union** ⟨union_list⟩ **on** ⟨identifier⟩ **)**
⟨union_list⟩ : : = ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **union** ⟨union_list⟩
⟨local_attr_name⟩ : : = ⟨source_name⟩**.**⟨class_name⟩**.**⟨attribute_name⟩
. . .

Terminological relationships usate per la definizione del Common Thesaurus.

⟨relationships_dcl⟩ : : = ⟨local_entity⟩ ⟨relationship_type⟩ ⟨local_entity⟩
⟨local_entity⟩ : : = ⟨local_class_name⟩ | ⟨local_attr_name⟩
⟨local_class_name⟩ : : = ⟨source_name⟩**.**⟨class_name⟩
⟨relationship_type⟩ : : = **SYN** | **BT** | **NT** | **RT**
. . .

Definizione degli **OLCD** integrity constraint: le regole sono definite tramite la logica di *if then* e sono valide per ogni istanza del database.
Definizione delle regole di *or* e *union*.

⟨rule_dcl⟩ : : = **rule** ⟨identifier⟩ ⟨rule_spec⟩
⟨rule_spec⟩ : : = ⟨rule_pre⟩ **then** ⟨rule_post⟩ | { ⟨case_dcl⟩ }
⟨rule_pre⟩ : : = ⟨forall⟩ ⟨identifier⟩ **in** ⟨identifier⟩ **:** ⟨rule_body_list⟩
⟨rule_post⟩ : : = ⟨rule_body_list⟩
⟨case_dcl⟩ : : = **case of** ⟨identifier⟩ **:** ⟨case_list⟩
⟨case_list⟩ : : = ⟨case_spec⟩ | ⟨case_spec⟩ ⟨case_list⟩
⟨case_spec⟩ : : = ⟨identifier⟩ **:** ⟨identifier⟩ **;**
⟨rule_body_list⟩ : : = **(** ⟨rule_body_list⟩ **)**
| ⟨rule_body⟩
| ⟨rule_body_list⟩ **and** ⟨rule_body⟩
| ⟨rule_body_list⟩ **and (** ⟨rule_body_list⟩ **)**
⟨rule_body⟩ : : = ⟨dotted_name⟩ ⟨rule_const_op⟩ [⟨rule_cast⟩] ⟨literal_value⟩
| ⟨dotted_name⟩ **in** ⟨dotted_name⟩
| ⟨forall⟩ ⟨identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩
| **exists** ⟨identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩
⟨rule_const_op⟩ : : = = | ≥ | ≤ | > | <
⟨rule_cast⟩ : : = **(**⟨simple_type_spec⟩**)**
⟨dotted_name⟩ : : = ⟨identifier⟩ | ⟨identifier⟩**.**⟨dotted_name⟩
⟨forall⟩ : : = **for all** | **forall**

# Appendice D

# Il diagramma sintattico del linguaggio ODL$_{I^3}$

L'appendice riporta il diagramma sintattico del linguaggio ODL$_{I^3}$.
Il diagramma sintattico ha il vantaggio di essere intuitivo e facilmente leggibile, pur conservando lo stesso potere espressivo del BNF; in alcuni casi (vedi figure D.27 e D.28) è capace addirittura di esprimere vincoli in modo più conciso del BNF.

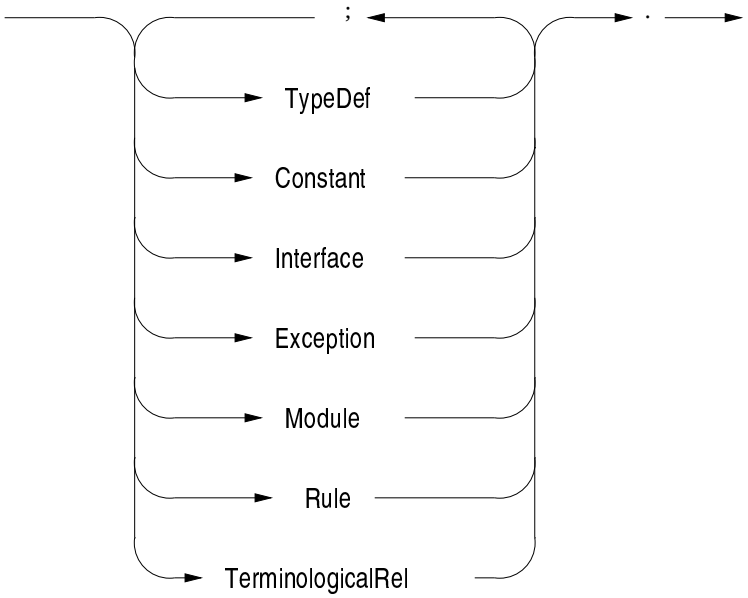



Figura D.1: Specification

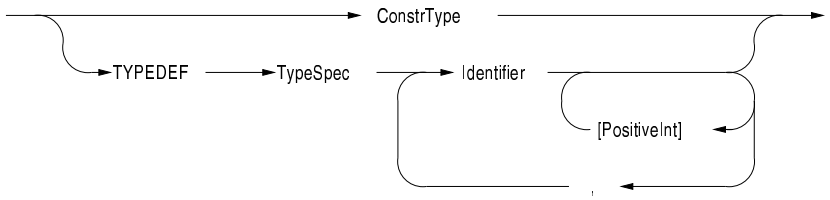



Figura D.2: TypeDef

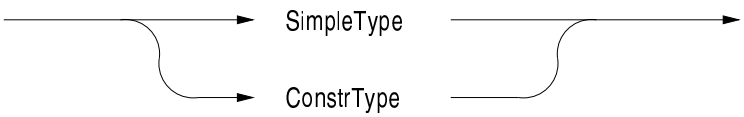



Figura D.3: TypeSpec

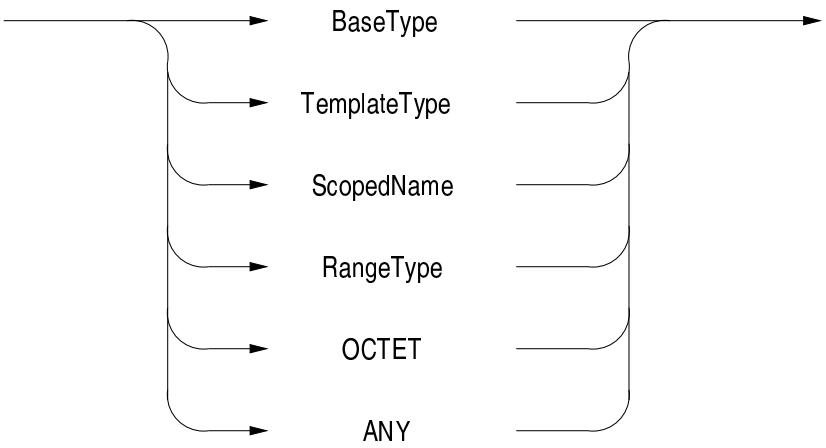



Figura D.4: SimpleType

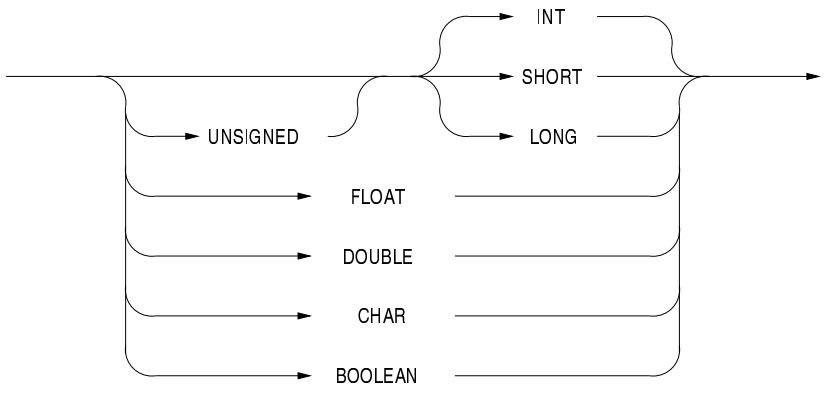



Figura D.5: BaseType

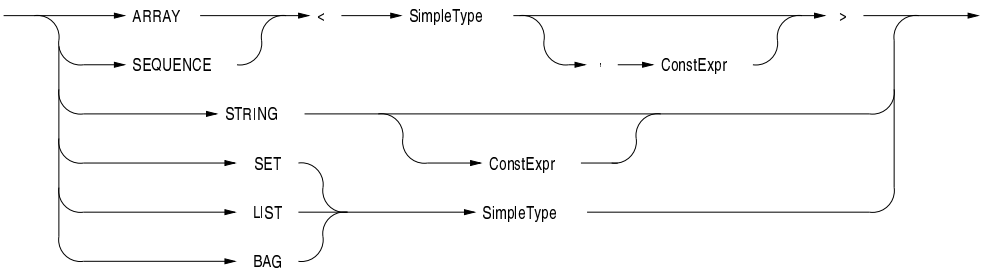



Figura D.6: TemplateType


Figura D.7: ScopedName

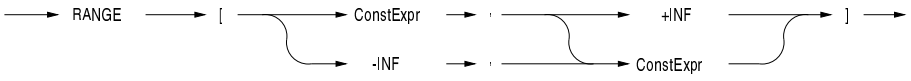



Figura D.8: RangeType

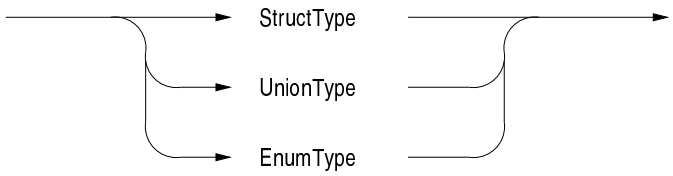



Figura D.9: ConstrType

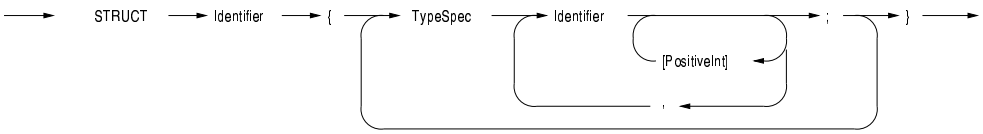



Figura D.10: StructType

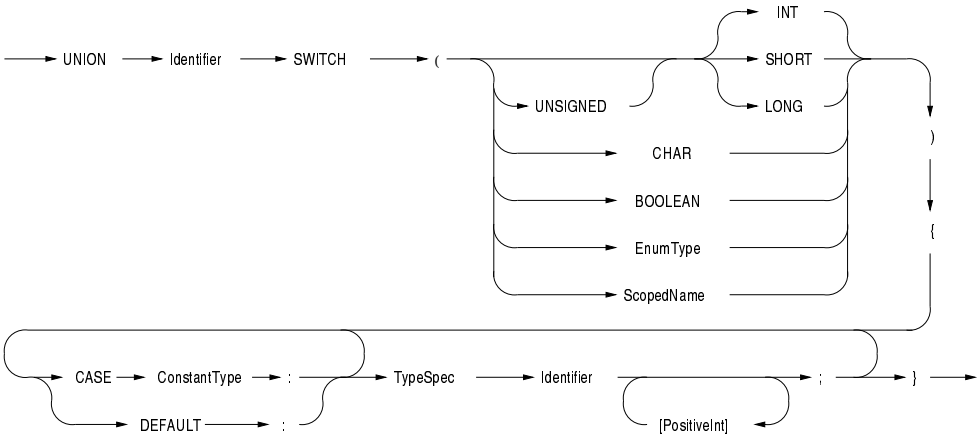



Figura D.11: UnionType

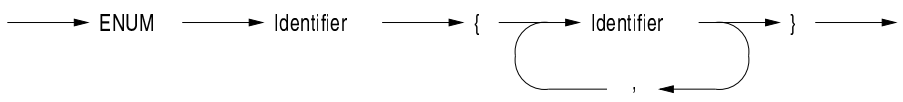



Figura D.12: EnumType

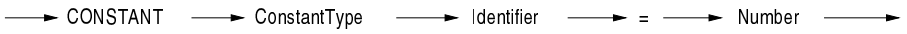



Figura D.13: Constant

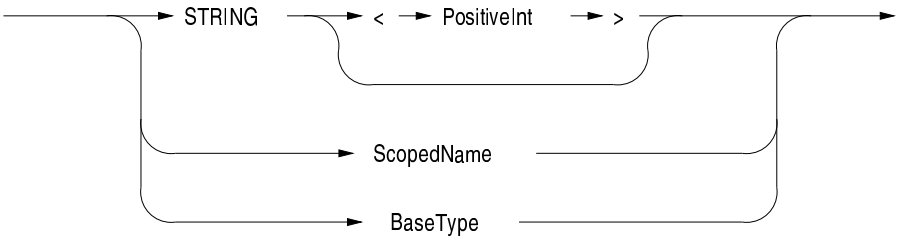



Figura D.14: ConstantType

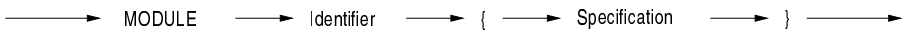



Figura D.15: Module

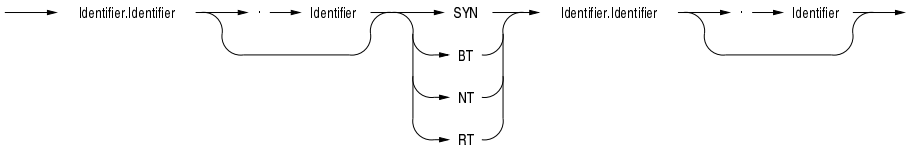



Figura D.16: TerminologicalRel

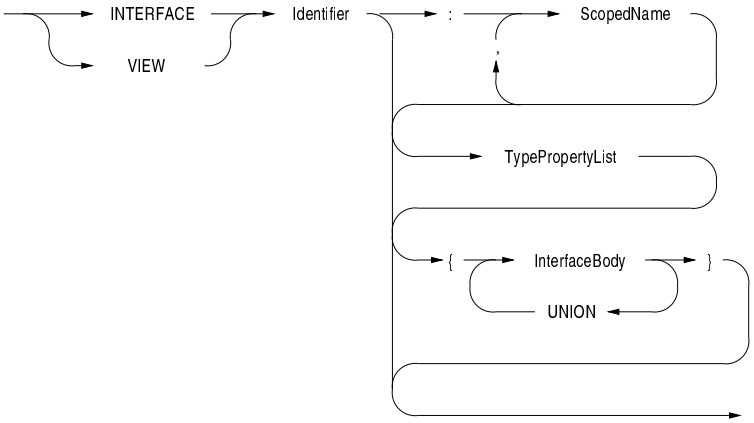



Figura D.17: Interface

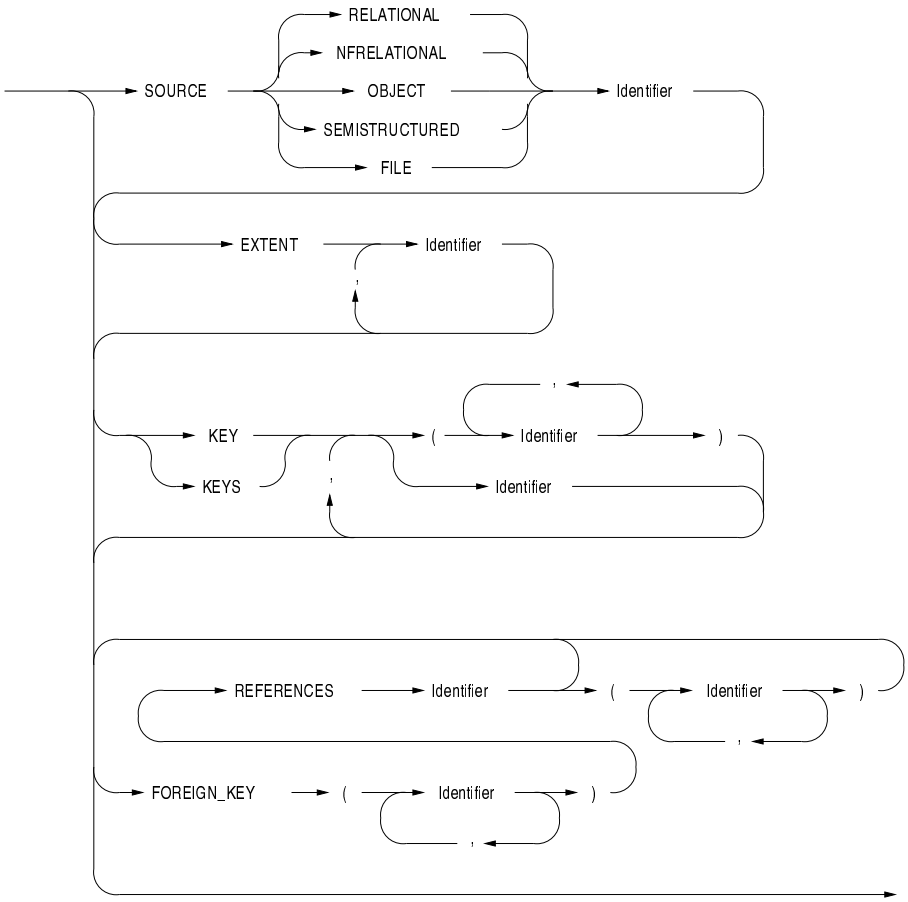



Figura D.18: PropertyList

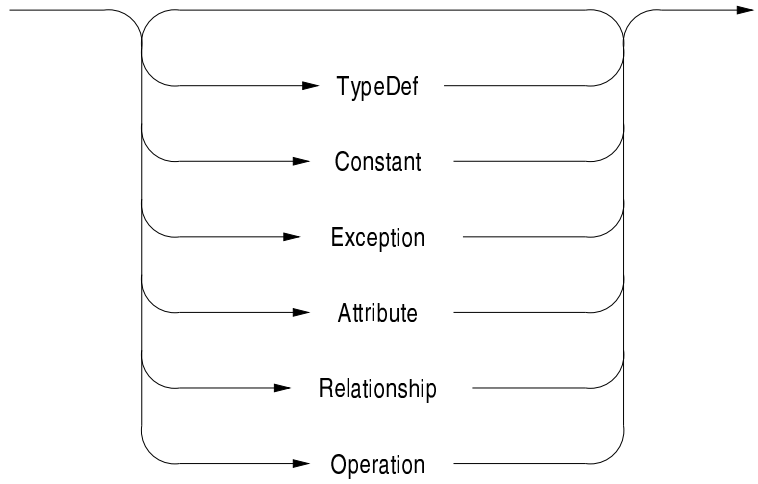



Figura D.19: InterfaceBody

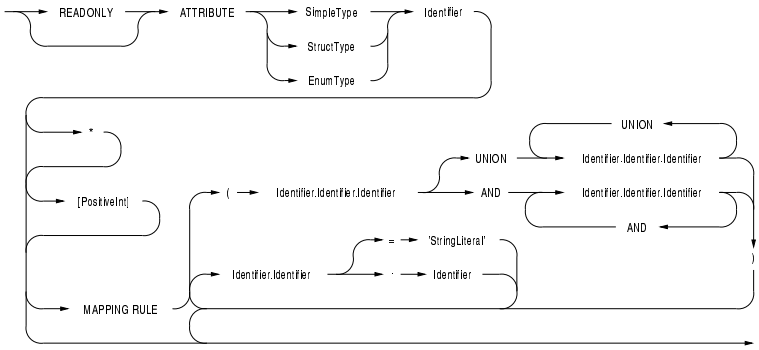



Figura D.20: Attribute

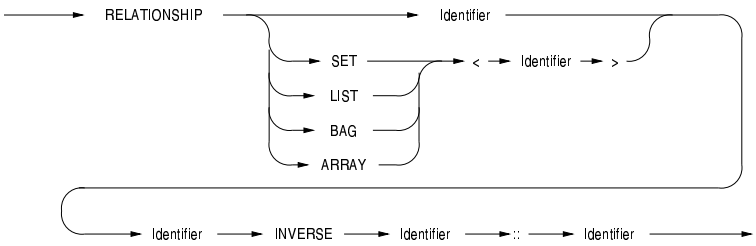



Figura D.21: Relationship

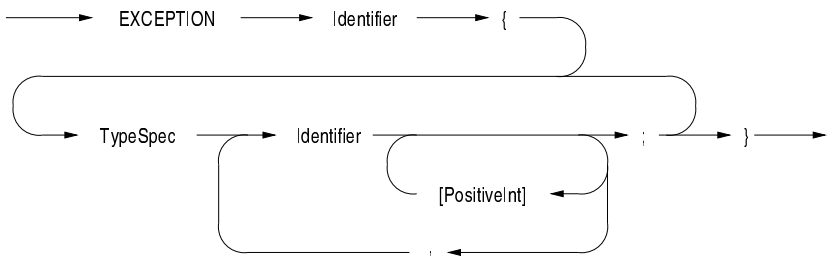



Figura D.22: Exception

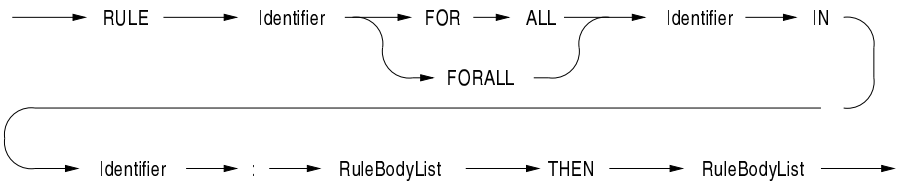



Figura D.23: Rule

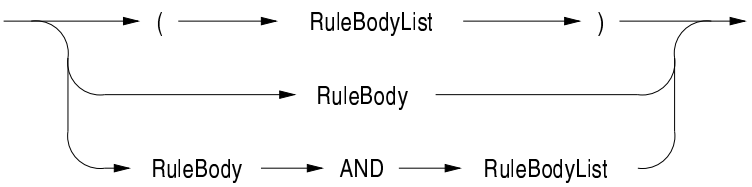



Figura D.24: RuleBodyList

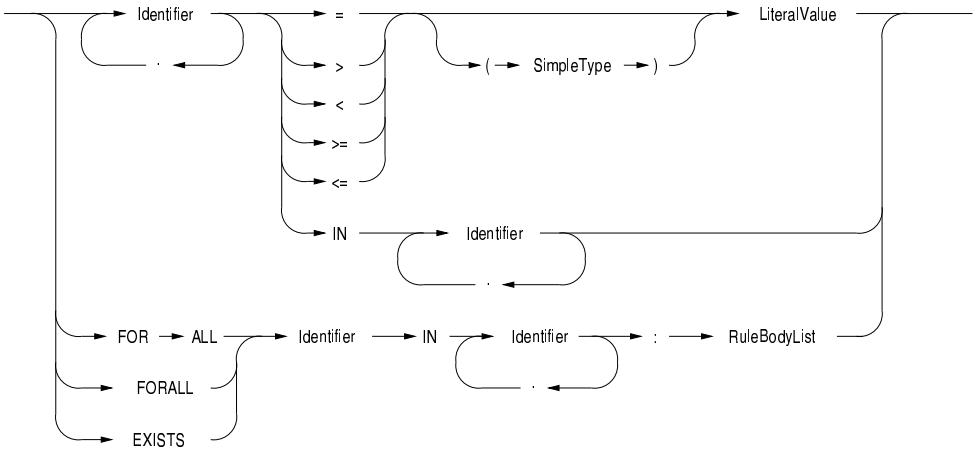



Figura D.25: RuleBody

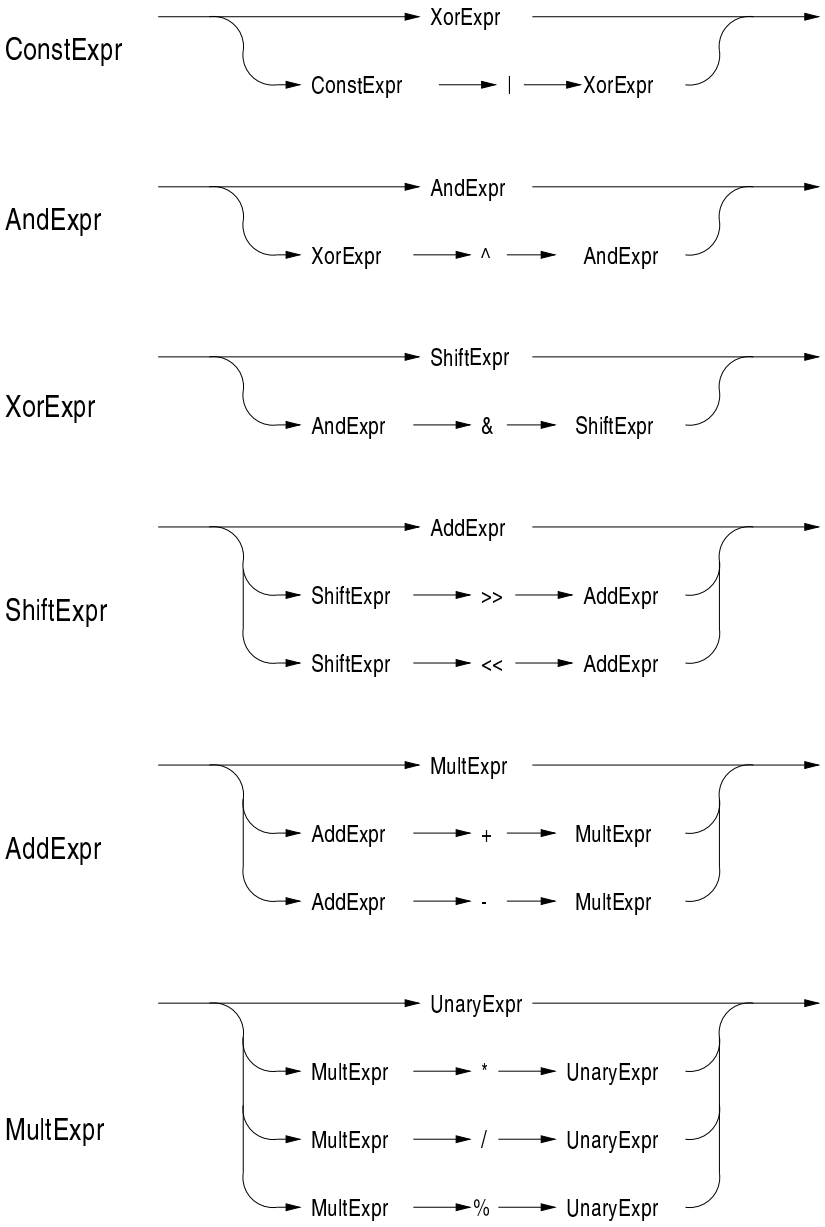



Figura D.26: Espressioni costanti

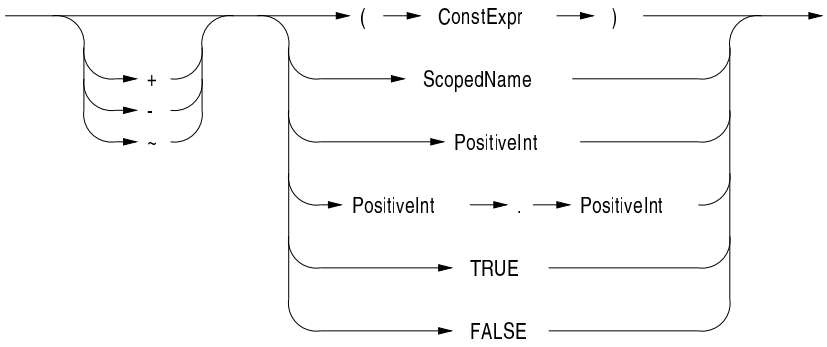



Figura D.27: UnaryExpr

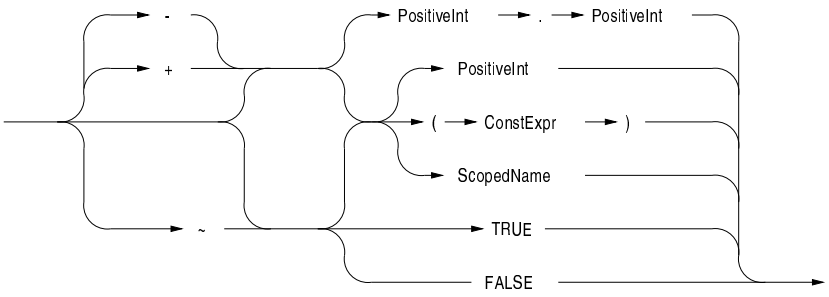



Figura D.28: UnaryExpr (versione più vincolante)

# Appendice E

# Sorgenti di esempio in linguaggio $\text{ODL}_{I^3}$

Di seguito è riportata la descrizione, attraverso il linguaggio $\text{ODL}_{I^3}$, dell'esempio di riferimento di Sezione 2.5.

```
Cardiology_Department (CD):

interface Patient                          interface Physician
 ( source semistructured                    ( source semistructured
      Cardiology_Department )                    Cardiology_Department )
{ attribute string         name;           {  attribute string  name;
  attribute string         address;           attribute string  address;
  attribute set<Exam>      exam*;             attribute integer phone;
  attribute integer        room;              attribute string specialization
  attribute integer        bed;
  attribute string         therapy*;
  attribute set<Physician> physician*; };


interface Nurse                            interface Exam
 ( source semistructured                   ( source semistructured
      Cardiology_Department )                    Cardiology_Department )
{ attribute string       name;             {  attribute integer date;
  attribute string       address;             attribute string  type;
  attribute integer      level;               attribute string  outcome;};
  attribute set<Patient> patient; };
```

```
Intensive_Care_Department (ID):

interface Patient
( source relational Intensive_Care
  extent Patients
  key  code
  foreign_key(test)
           references Test (number),
  foreign_key(doctor_id)
           references Doctor (id) )
{ attribute string  code;
  attribute string  first_name;
  attribute string  last_name;
  attribute string  address;
  attribute string  test;
  attribute string  doctor_id; };

interface Doctor
( source relational Intensive_Care
  extent Medical_Staffers
  key id )
{ attribute string  id;
  attribute string  first_name;
  attribute string  last_name;
  attribute integer phone;
  attribute string  address;
  attribute string  availability;
  attribute string  position; };

interface Test
( source relational Intensive_Care
  extent Tests
  key number )
{ attribute integer number;
  attribute string  type;
  attribute integer date;
  attribute string  laboratory;
  attribute string  result; };

interface Dis_Patient
( source relational Intensive_Care
  extent Dis_Patients
  key code
  foreign_key(code)
         references Patient )
{ attribute string  code;
  attribute integer date;
  attribute string  note; };
```

# Bibliografia


[1] R. Hull and R. King et al. Arpa $i^3$ reference architecture, 1995. Available at http://www.isse.gmu.edu/I3_Arch/index.html.

[2] Gio Wiederhold et al. *Integrating Artificial Intelligence and Database Technology*, volume 2/3. Journal of Intelligent Information Systems, June 1996.

[3] F. Saltor and E. Rodriguez. On intelligent access to heterogeneous information. In *Proceedings of the 4th KRDB Workshop,* Athens, Greece, August 1997.

[4] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.

[5] S. Chawathe, Garcia Molina, H., J. Hammer, K.Ireland, Y. Papakostantinou, J.Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *IPSJ Conference, Tokyo, Japan*, 1994. ftp://db.stanford.edu /pub/chawathe/1994/ tsimmis-overview.ps.

[6] H. Garcia-Molina et al. The TSIMMIS approach to mediation: Data models and languages. In *NGITS workshop*, 1995. ftp://db.stanford.edu /pub/garcia/1995/tisimmis-models-languages.ps.

[7] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specification. Technical report, Stanford University, 1995. ftp://db.stanford.edu /pub/papakonstantinou/1995/ medmaker.ps.

[8] Y.Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *VLDB Int. Conf.*, Bombay, India, September 1996.

[9] N.Guarino. Semantic matching: Formal ontological distinctions for information organization, extraction, and integration. Technical report, Summer School on Information Extraction, Frascati, Italy, July 1997.

[10] N.Guarino. Understanding, building, and using ontologies. A commentary to 'Using Explicit Ontologies in KBS Development', by van Heijst, Schreiber, and Wielinga.

[11] M.J.Carey, L.M. Haas, P.M. Schwarz, M. Arya, W.F. Cody, R. Fagin, M. Flickner, A.W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J.H. Williams, and E.L. Wimmers. Object exchange across heterogeneous information sources. Technical report, Stanford University, 1994.

[12] M.T. Roth and P. Scharz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources. In *Proc. of the 23rd Int. Conf. on Very Large Databases*, Athens, Greece, March 1995.

[13] Y. Arens, C.Y. Chee, C. Hsu, and C. A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[14] Y. Arens, C. A. Knoblock, and C. Hsu. Query processing in the sims information mediator. *Advanced Planning Technology*, 1996.

[15] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency checking in complex object database schemata with integrity constraints. *IEEE Transactions on Knowledge and Data Engineering*, 10:576–598, July/August 1998.

[16] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[17] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. ODB-QOPTIMIZER: A tool for semantic query optimization in oodb. In *Int. Conference on Data Engineering - ICDE97*, 1997. http://sparc20.dsi.unimo.it.

[18] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. Odb-tools: a description logics based tool for schema validation and semantic query optimization in object oriented databases. In *Sesto Convegno AIIA - Roma*, 1997.

[19] Odb-tools Project. Available at http://sparc20.dsi.unimo.it/index.html.

[20] S. Castano and V. De Antonellis. Semantic dictionary design for database interoperability. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, 1997.

[21] S. Castano and V. De Antonellis. Deriving global conceptual views from multiple information sources. In *preProc. of ER'97 Preconference Symposium on Conceptual Modeling, Historical Perspectives and Future Directions*, 1997.

[22] A.G. Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

[23] T. Catarci and M. Lenzerini. Representing and using interschema knowledge in cooperative information systems. *Journal of Intelligent and Cooperative Information Systems*, 2(4):375–398, 1993.

[24] B. Everitt. *Computer-Aided Database Design: the DATAID Project.* Heinemann Educational Books Ltd, Social Science Research Council, 1974.

[25] S. Montanari. Un approccio intelligente all'Integrazione di Sorgenti Eterogenee di Informazione. Tesi di Laurea, Univeristà di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1998.

[26] D. Beneventano, A. Corni, S. Lodi, and M. Vincini. ODB: validazione di schemi e ottimizzazione semantica on-line per basi di dati object oriented. In *Quinto Convegno Nazionale su Sistemi Evoluti per Basi di Dati - SEBD97, Verona*, pages 208–225, 1997.

[27] W. A. Woods and J. G. Schmolze. The KL-ONE family. In F. W. Lehman, editor, *Semantic Networks in Artificial Intelligence*, pages 133–178. Pergamon Press, 1992. Pubblished as a Special issue of *Computers & Mathematics with Applications*, Volume 23, Number 2-9.

[28] R. G. G. Cattell, editor. *The Object Database Standard: ODMG93*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[29] R.G.G. Cattell and others., editors. *The Object Data Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[30] 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.

[31] J. J. King. QUIST: a system for semantic query optimization in relational databases. In *7th Int. Conf. on Very Large Databases*, pages 510–517, 1981.

[32] J. J. King. *Query optimization by semantic reasoning.* PhD thesis, Dept. of Computer Science, Stanford University, Palo Alto, 1981.

[33] M. M. Hammer and S. B. Zdonik. Knowledge based query processing. In *6th Int. Conf. on Very Large Databases*, pages 137–147, 1980.

[34] 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.

[35] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured data. In *23rd VLDB Int. Conf.*, 1997.

[36] J. D Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, Maryland - USA, 1989.

[37] S. Abiteboul. Querying semi-structured data. In *Proceedings of the International Conference on Database Theory, ICDT97*, pages 1–18, Athens, Greece, 1997.

[38] P. Buneman. Semistructured data. In *Proc. of 1997 Symposium on Principles of Database Systems (PODS97)*, pages 117–121, Tucson, Arizona, 1997.

[39] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proc. of the ACM SIGMOD International Conference*, pages 505–516. ACM Press, 1996.

[40] Y.Papakonstantinou, H.Garcia-Molina, and J.Widom. Object exchange across heterogeneous information sources. In *Proc. of ICDE95*, Taipei, Taiwan, 1995.

[41] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The lorel query language for semistructured data. *Journal of Digital Libraries*, 1(1), 1996.

[42] S. Bergamaschi and C. Sartori. On taxonomic reasoning in conceptual design. *ACM Trans. on Database Systems*, 17(3):385–422, September 1992.

[43] 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.

[44] D. Calvanese, G. De Giacomo, and M. Lenzerini. Structured objects: Modeling and reasoning. In *Proc. of Int. Conference on Deductive and Object-Oriented Databases*, 1995.

[45] G. P. Grifa. Analisi di Affinità Strutturali fra classi ODL$_{I^3}$ nel Sistema MOMIS. Tesi di Laurea, Univeristà di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999.

[46] N.V. Findler, editor. *Associative Networks.* Academic Press, 1979.

[47] A. Zaccaria. MOMIS: Il componente Query Manager. Tesi di Laurea, Univeristà di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999.

[48] Domenico Beneventano, Sonia Bergamaschi, and Claudio Sartori. Semantic query optimization by subsumption in OODB. In H. Christiansen, H. L. Larsen, and T. Andreasen, editors, *Flexible Query Answering Systems*, volume 62 of *Datalogiske Skrifter - ISSN 0109-9799*, Roskilde, Denmark, 1996.

[49] A. Rabitti. Architettura di un Mediatore per un Sistema di Sorgenti Distribuite ed Autonome. Tesi di Laurea, Univeristà di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1998.

[50] Kathy Walrath Mary Campione. *The Java Tutorial : Object-Oriented Programming for the Internet (Java Series).* Addison-Wesley, 1998.

[51] Byacc/Java. Available at http://www.lincom-asg.com/~rjamison/byacc.