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

MOMIS: Il componente Query Manager

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Andrea Zaccaria

Correlatore
Ing. Maurizio Vincini

Controrelatore
Chiar.mo Prof. Paolo Tiberio

Anno Accademico 1997 - 98

Parole chiave:
Intelligent Information Integration
Integrazione Semantica
Mediatore
Database eterogenei
Query Processing
Query Manager

RINGRAZIAMENTI

Ringrazio la Professoressa Sonia Bergamaschi, l'Ing. Maurizio Vincini e l'Ing. Alberto Corni per l'aiuto fornito durante la realizzazione della presente tesi.
Ringrazio inoltre tutti gli amici e colleghi che mi sono stati vicino, rendendo sicuramente più piacevoli questi anni di studi.
Un ringraziamento speciale va poi ai miei genitori, che hanno reso possibile tutto ciò, e alla mia fidanzata, Tania, che mi ha sempre sostenuto e sopportato.

Indice

Introduzione **1**

1 MOMIS: Progetto di un Sistema Intelligente di Integrazione **5**
1.1 Architettura di riferimento per sistemi I^3 7
1.1.1 A cosa serve la tecnologia I^3 e quali problemi deve risolvere 8
1.1.2 Servizi di Coordinamento 10
1.1.3 Servizi di Amministrazione 11
1.1.4 Servizi di Integrazione e Trasformazione Semantica . . . 12
1.1.5 Servizi di Wrapping 13
1.1.6 Servizi Ausiliari 13
1.2 Il sistema MOMIS 14
1.2.1 Scelte implementative 16
1.2.2 Il Modello dei dati 20
1.2.3 L'architettura di MOMIS 22
1.3 ODB-Tools Engine 24
1.3.1 La Logica Descrittiva **OLCD** 24
1.3.2 Le regole **OLCD** e l'espansione semantica di un tipo . . . 26
1.3.3 Validazione e Sussunzione 27
1.3.4 Ottimizzazione semantica delle interrogazioni 28
1.3.5 Architettura di ODB-Tools 29

2 Integrazione intensionale di schemi **31**
2.1 Processo di Integrazione 34
2.2 Estrazione di Relazioni Terminologiche 36
2.3 Analisi di Affinità delle classi ODL_{I^3} 42
2.4 Generazione dei Cluster di classi ODL_{I^3} 43
2.5 Costruzione dello Schema Globale di mediatore 45

3 Integrazione estensionale di schemi **53**
3.1 Le relazioni estensionali 53
3.1.1 Definizione degli assiomi estensionali 58

3.1.2 Traduzione degli assiomi estensionali in proprietà intensionali . . . 60
3.1.3 Verifica di congruenza e individuazione delle Base Extension . . . 62
3.1.4 Generazione della gerarchia estensionale . . . 66

4 Il modulo Query Manager 69
4.1 Ottimizzazione semantica globale . . . 70
4.2 Individuazione delle sorgenti . . . 72
4.2.1 Gestione di query complesse . . . 76
4.3 Generazione delle query locali . . . 79
4.4 Ottimizzazione semantica locale . . . 81
4.5 Composizione della risposta . . . 82

5 Progetto e Realizzazione del Query Manager 85
5.1 L'ambiente di sviluppo . . . 86
5.2 Query Manager . . . 87
5.2.1 Query Parser and Validator . . . 92
5.2.2 Query Translator . . . 94
5.3 Il package "oql" . . . 97
5.4 Il package "queryman" . . . 99
5.4.1 La classe "QueryManager" . . . 99
5.4.2 La classe "Query" . . . 101
5.4.3 Le classi "Plan" e "Data" . . . 104
5.5 Il package "globalschema" . . . 105
5.5.1 La classe "MappingTable" . . . 105
5.5.2 La classe "BaseExtension" . . . 109
5.5.3 La classe "ExtensionalHierarchy" . . . 110
5.5.4 La classe "TransOutput" . . . 111
5.6 Il package "utility" . . . 112
5.6.1 La classe "parser" . . . 113
5.6.2 Utility aggiuntive . . . 114
5.7 Il software . . . 115

6 Confronto con altri lavori 117
6.1 TSIMMIS . . . 117
6.1.1 Il modello OEM . . . 119
6.1.2 Il linguaggio MSL . . . 119
6.1.3 Il generatore di Wrapper . . . 120
6.1.4 Il generatore di Mediatori . . . 121
6.1.5 Il Linguaggio LOREL . . . 123

6.1.6 Pregi e difetti di TSIMMIS . . . 124
6.2 GARLIC . . . 125
6.2.1 Il linguaggio GDL . . . 127
6.2.2 Query Planning . . . 129
6.2.3 Pregi e difetti di GARLIC . . . 130
6.3 SIMS . . . 130
6.3.1 Integrazione delle sorgenti . . . 132
6.3.2 Query Processing . . . 133
6.3.3 Pregi e difetti di SIMS . . . 135
6.4 Osservazioni . . . 135

Conclusioni **137**

A Glossario I^3 **139**
A.1 Architettura . . . 139
A.2 Servizi . . . 141
A.3 Risorse . . . 144
A.4 Ontologia . . . 146

B Esempio in ODL$_{I^3}$ **149**

C Esempio di produzione di documentazione **153**

D Grammatica OQL **163**

E Restrizione del OQL per le BasicQuery **167**

Elenco delle figure

1.1 Diagramma dei servizi I^3 10
1.2 Servizi I^3 ppresenti nel mediatore 15
1.3 Architettura del sistema MOMIS 22
1.4 Architettura di ODB-Tool 29

2.1 Esempio di riferimento 35
2.2 Thesaurus comune per le sorgenti $\mathbf{S_1}$, $\mathbf{S_2}$, e $\mathbf{S_3}$. 42
2.3 Albero di affinità 44
2.4 Esempio di classe globale in ODL$_{I^3}$ 49
2.5 Mapping table di University_Person e Workplace 49
2.6 Fasi dell' Integrazione Intensionale 51

3.1 Estensione della classe di entità 55
3.2 57
3.3 Esempio di gerarchia di ereditarietà 63
3.4 Esempio di Verifica di Congruenza 64
3.5 esempio di "$baseextensin$" 65
3.6 Rappresentazione della gerarchia estensionale 67

5.1 Schema funzionale del Mediatore MOMIS 88
5.2 Definizione del piano di esecuzione di una Query 89
5.3 Esecuzione del piano 91
5.4 Schema di acquisizione ed ottimizzazione della query 92
5.5 Trasformazione di una query 96
5.6 Gerarchia di classi per la rappresentazione delle query oql 98
5.7 Modello ad oggetti del modulo queryman 100
5.8 Modello ad oggetti della classe MappingTable 106
5.9 Modello ad oggetti della classe MappingElement 107
5.10 Modello ad oggetti della classe BaseExtension 110
5.11 Modello ad oggetti della classe ExtensionalHierarchy 111
5.12 Schema della classe TransOutput 112

6.1 Architettura TSIMMIS 118
6.2 Oggetti esportati da **CS** in OEM 122
6.3 Oggetti esportati da **WHOIS** in OEM 122
6.4 Oggetti esportati da **MED** 123
6.5 Architettura GARLIC 126
6.6 GDL schema 128
6.7 Esempio di query SIMS 132
6.8 Mapping tra *domain model* e modello locale 133
6.9 Query Processing 134

Introduzione

Lo sviluppo delle tecnologie telematiche, tanto per i sistemi di elaborazione, quanto per le reti di calcolatori, ha portato ad una sempre maggiore presenza di sorgenti informative determinando un vera e propria esplosione nella quantità e varietà di dati accessibili.
Poter gestire in modo efficace questa mole di dati è diventato quindi un fattore cruciale per il successo aziendale ma, paradossalmente, l'aumento nell'offerta di informazione fatica a tradursi in un effettivo vantaggio per l'utente. Tale situazione è la conseguenza di una crescita irregolare che ha portato ad avere una grande varietà di sorgenti disomogenee e quindi difficilmente integrabili.

Il problema di base è appunto l'eterogeneità dei sistemi, la quale può presentarsi in diversi modi, a partire dalle piattaforme Hardware e software su cui una sorgente è basata (ad esempio diversi DBMS e linguaggi di interrogazione), fino ad arrivare ai modelli dei dati (relazionale, object-oriented, ecc...) e agli schemi usati per la rappresentazione della struttura logica dei dati memorizzati.
In un contesto di questo tipo, risulta evidente che, per poter reperire le informazioni desiderate, sarebbe necessario avere familiarità con il contenuto, le strutture ed i linguaggi di interrogazione propri delle singole sorgenti. L'utente dovrebbe quindi essere in grado di scomporre la propria interrogazione in una sequenza di sotto-interrogazioni rivolte alle sorgenti di informazioni provvedendo poi a "$trattare$" i risultati parziali, in modo da ottenere una risposta unificata. Tutto ciò dovrebbe essere fatto tenendo presente le possibili trasformazioni che possono subire i dati, le relazioni che li legano, le proprietà che possono avere in comune e le discrepanze sussistenti tra le diverse rappresentazioni.
Disponendo di un numero sempre maggiore di sorgenti e di dati da manipolare diviene difficile trovare persone che posseggano tutte le conoscenze necessarie, perciò risulta indispensabile avere un processo che automatizzi l'intera fase di reperimento ed integrazione delle informazioni.

Questa tesi si inserisce, appunto, in un sistema più ampio denominato **MOMIS** (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources) [1, 2, 3, 4], sviluppato con l'obiettivo di realizzare l'integrazione di sorgenti eterogenee e distri-

buite.
MOMIS adotta un'architettura a tre livelli con un *Mediatore* che ne occupa la parte centrale ed avente lo scopo di fornire una visione integrata degli schemi locali. Questa vista integrata deve permettere all'utente la formulazione di interrogazioni svincolandolo dal dover conoscere la struttura ed il contenuto delle singole sorgenti. Il *Mediatore* rappresenta dunque il cuore del sistema ed ha il compito di realizzare l'integrazione degli schemi e di provvedere alla gestione delle interrogazioni.

Elementi indubbiamente innovativi di questo progetto sono l'impiego di un approccio *semantico* e di Logiche Descrittive. Questi elementi introducono, infatti comportamenti intelligenti che permettono di sfruttare al meglio le conoscenze intensionali, semantiche ed estensionali sia inter-schema sia intra-schema, per generare una vista globale il più possibile espressiva. Oltre ad una migliore integrazione delle sorgenti si ottiene dunque un processo di gestione delle interrogazioni più efficiente e funzionale.
Di fondamentale importanza è quindi l'impiego di ODB-Tools, un ambiente software sviluppato presso l'Università di Modena, in grado di realizzare la validazione di schemi ad oggetti e l'espansione semantica delle interrogazioni.

Obiettivo della presente tesi è stata l'analisi e la progettazione del modulo Query Manager, cioè il modulo del Mediatore preposto alla gestione delle interrogazioni, si è dunque individuata e formalizzata la sequenza di attività necessarie alla generazione della risposta a partire da una generica query globale posta dall'utente.
I risultati conseguiti sono stati, oltre alla definizione ed implementazione delle principali classi necessarie alla gestione delle interrogazioni, lo sviluppo del modulo software per la trasformazione della query globale in interrogazioni rivolte alle sorgenti e di un modulo per il parsing ed acquisizione di query espresse in linguaggio OQL.

La tesi è organizzata nel seguente modo. Nel Capitolo 1 viene dapprima introdotta l'architettura di riferimento I^3 per i sistemi di Integrazione di Informazioni, per poi illustrare le scelte implementative fatte in MOMIS. Nell'ultima parte del capitolo viene anche presentato il componente ODB-Tools usato al fine di introdurre, nel sistema, comportamenti intelligenti.
I Capitoli 2 e 3 illustrano l'approccio usato nell'integrazione intensionale ed estensionale degli schemi di sorgenti eterogenee.
Nel Capitolo 4 viene presentato il modulo Query Manager, mostrando le fasi che lo caratterizzano e le informazioni che devono essere utilizzate.
Il Capitolo 5 descrive il progetto del Query Manager ed il software sviluppato. Il codice non è allegato alla tesi ma è comunque reperibile all'indirizzo *http://sparc20.dsi.unimo.it/tesi/index.html*.
Il Capitolo 6 rappresenta lo stato dell'arte, riporta, cioè, una descrizione dei Query

Manager di altri sistemi I^3 basati su Mediatore e sviluppati in laboratori di ricerca internazionali.
Sono inoltre presenti cinque appendici. In particolare in Appendice A viene riportato un glossario dei termini usati in ambito I^3, in Appendice B è descritto lo schema ODL$_{I^3}$ completo dell'esempio di riferimento, in Appendice C viene riportato un esempio di documentazione ottenibile con il componente Javadoc ed infine, nelle Appendici D e E, vengono mostrate le rappresentazioni BNF della grammatica OQL e della versione ristretta per le Basic Query.

Capitolo 1

MOMIS: Progetto di un Sistema Intelligente di Integrazione

La presenza di un numero sempre maggiore di fonti di informazione, all'interno di un azienda come sulla rete Internet, ha reso possibile oggi accedere ad un vastissimo insieme di dati, sparsi su macchine diverse come pure in luoghi diversi. Parallelamente quindi all'aumento delle probabilità di trovare un dato sulla rete informatica, in qualsivoglia fonte e formato, va costantemente aumentando la difficoltà di recuperare questo dato in tempi e modi accettabili, essendo tra di loro le fonti di informazione fortemente eterogenee, sia per quanto riguarda i tipi di dati (testuali, immagini . . .), sia per quanto riguarda il modo di descriverli, e quindi di *segnalarli* ai potenziali utenti. Contestualmente alla difficoltà di reperire un dato, pur nella sicurezza di ritrovarlo, si va inoltre delineando un altro tipo di problema, che paradossalmente nasce dall'abbondanza di informazioni, e che viene percepito dall'utente come *information overload* (sovraccarico di informazioni): il numero crescente di informazioni (e magari la loro replicazione) genera confusione, rendendo pressocché impossibile isolare efficientemente i dati necessari a prendere determinate decisioni.
In questo scenario, al momento fortemente studiato, e che coinvolge diverse aree di ricerca e di applicazione, si vanno oggi ad inserire i sistemi di supporto alle decisioni (DSS, Decision Support System), l'integrazione di basi di dati eterogenee, i *datawarehouse* (magazzino), fino ad arrivare ai sistemi distribuiti. I *decision maker* lavorano su fonti diverse (inclusi file system, basi di dati, librerie digitali, . . .) ma sono per lo più incapaci di ottenere e fondere le informazioni in un modo efficiente.
L'integrazione di basi di dati invece, e tutto ciò che va sotto il nome di data-warehouse, si occupa di materializzare presso l'utente finale delle viste, ovvero delle porzioni delle sorgenti, replicando però fisicamente i dati, ed affidandosi a complicati algoritmi di ”mantenimento” di questi dati, per assicurare la loro

consistenza a fronte di cambiamenti nelle sorgenti originali. Con *Integration of Information* invece, come è descritto in [5], si rappresentano in letteratura tutti quei sistemi in grado di combinare tra di loro dati provenienti intere sorgenti o parti selezionate di esse, senza fare uso della replicazione fisica delle informazioni, bensì basandosi sulle loro descrizioni. Quando inoltre questa integrazione utilizza tecniche di intelligenza artificiale, sfruttando le conoscenze acquisite, possiamo parlare di Intelligent Integration of Information (I^3), che si distingue quindi dalle altre forme di integrazione prefiggendosi non una semplice aggregazione di informazioni, bensì anche un aumento del loro valore, ottenendo nuove informazioni dai dati ricevuti.
Con questi obiettivi si è quindi inserita, nell'ambito dell'integrazione, l'Intelligenza Artificiale (IA), che già aveva dato buoni risultati in domini applicativi più limitati. Naturalmente, è ovvio come sia pressocché impossibile pensare ad un sistema che vada bene per tutti i domini applicativi, e che magari integri un numero altissimo di sorgenti. Per questo motivo, per realizzare sistemi molto ampi, è stata proposta una partizione delle risorse e dei servizi che questi sistemi devono supportare, e che si articola su due dimensioni:

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

2. verticalmente: molti domini, con un numero limitato (e minore di 10) di sorgenti.

I domini nei vari livelli si scambieranno dati e informazioni tra di loro, ma non saranno strettamente collegati. Per esempio, in un sistema di recapito merci navale, le informazioni sulle navi saranno integrate da un modulo intermedio, quelle sul tempo nelle varie regioni da un altro modulo intermedio, ed un ulteriore modulo, ad un livello superiore, provvederà all'integrazione dei dati che gli verranno forniti dai *mediatori* (o *facilitatori*) sottostanti.
In questo quadro, dal 1992, si inserisce il progetto di ricerca I^3 fondato e sponsorizzato dall'ARPA, agenzia che fa capo al Dipartimento di Difesa americano [6]. I^3 si focalizza sul livello intermedio della partizione sopra descritta, livello che media tra gli utilizzatori e le sorgenti. All'interno di questo livello staranno diversi moduli (per una descrizione più dettagliata si rimanda al paragrafo successivo di questo capitolo), tra i quali i più importanti sono:

- *facilitator* e *mediator* (le differenze tra i due sono flebili ed ancora ambigue in letteratura), che ricercano le fonti "interessanti" e combinano i dati da esse ricevuti;

- *query processor*, che riformulano le query aumentando le probabilità di successo di quest'ultime;
- *data miner*, che analizzano i dati per estrarre informazioni intensionali implicite.

Nell'impostazione del progetto di Integrazione di Sorgenti Eterogenee presentato nella tesi, abbiamo seguito i principi ispiratori citati, sia per la loro completezza, sia per la riconosciuta validità del modello proposto. Oltre alla architettura di riferimento, muovendosi questo progetto in un campo di ricerca particolarmente giovane e in evoluzione, è riportato in appendice il glossario, a cui rifarsi per termini che risultino ambigui o poco chiari, definito nell'Appendice A.

1.1 Architettura di riferimento per sistemi I^3

L'architettura di riferimento presentata in questo paragrafo è stata tratta dal sito web [6], 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^3stanno 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.

1.1.1 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 [7];
  - 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 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

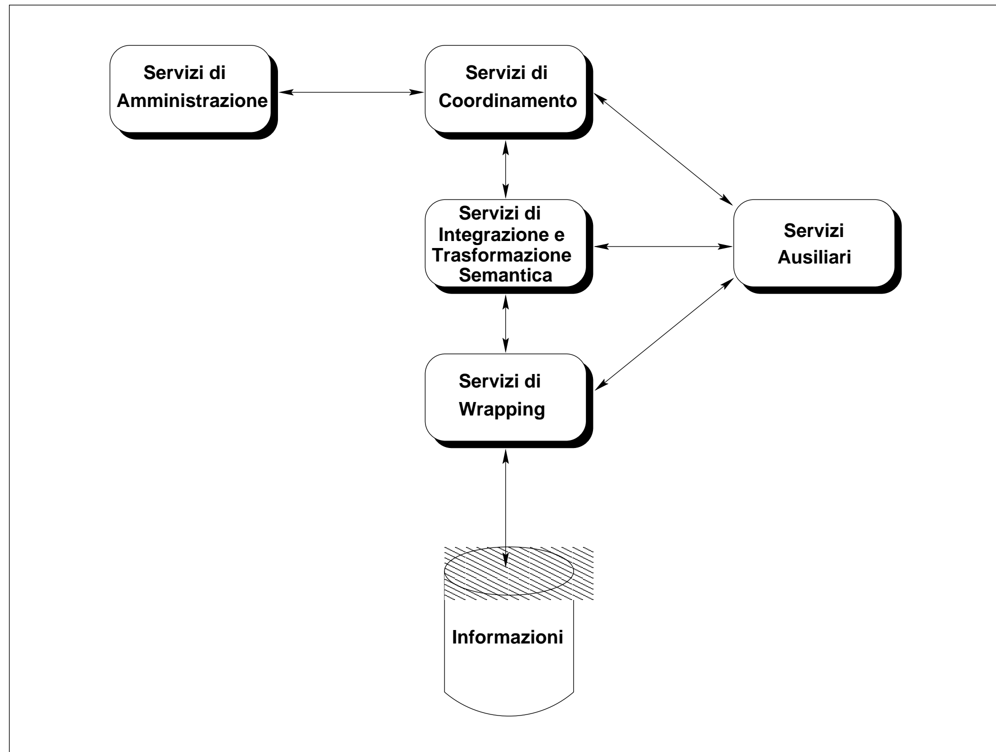

Figura 1.1: Diagramma dei servizi I^3

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.

1.1.2 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 sor-

genti (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.

1.1.3 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.

1.1.4 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 fon-

ti 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).

1.1.5 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.

1.1.6 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.

1.2 Il sistema MOMIS

Recependo l'esigenza di avere ambienti software "*intelligenti*" e funzionali, in grado non solo di fornire accesso a grosse moli di dati ma soprattutto capaci di aumentare la qualità ed il valore delle informazioni ottenibili, i gruppi operativi delle Università di Modena e Reggio e di Milano hanno avviato la realizzazione del sistema **MOMIS** (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources). **MOMIS** si colloca all'interno del progetto **MURST 40%** ***INTERDATA*** ed ha come fine la realizzazione di uno strumento che, seguendo le linee guida tracciate nell'ambito I^3, illustrate nel capitolo 1, permetta la reale integrazione di sorgenti distribuite, eterogenee sia strutturate sia semistrutturate.

In particolare il componente di **MOMIS** obiettivo di questa tesi è il **mediatore**, ovvero il modulo intermedio dell'architettura I^3 che si pone tra l'utente e le sorgenti di informazioni per realizzare quell'interazione tra dati e conoscenza necessaria a trasformare semplici dati in informazioni [8].
Secondo la definizione proposta da Wiederhold in [9] "un mediatore è quindi 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.

L'approccio architetturale adottato è stato quello *classico* caratterizzato dalla presenza di tre livelli distinti che interagiscono mediante interfacce standard. La validità di questo approccio è ormai riconosciuta a livello internazionale in quanto permette il conseguimento del necessario grado di astrazione e modularità. I tre livelli che compongono l'architettura sono:

1. utente: attraverso un'interfaccia grafica l'utente pone delle query su uno schema globale e riceve un'unica risposta, come se stesse interrogando una sola sorgente di informazioni;
2. mediatore: il mediatore gestisce l'interrogazione dell'utente, combinando, integrando ed eventualmente arricchendo i dati ricevuti dalle sorgenti. Perché ciò sia possibile verrà impiegato un modello (e quindi un linguaggio di interrogazione) comprensibile da tutte le fonti;

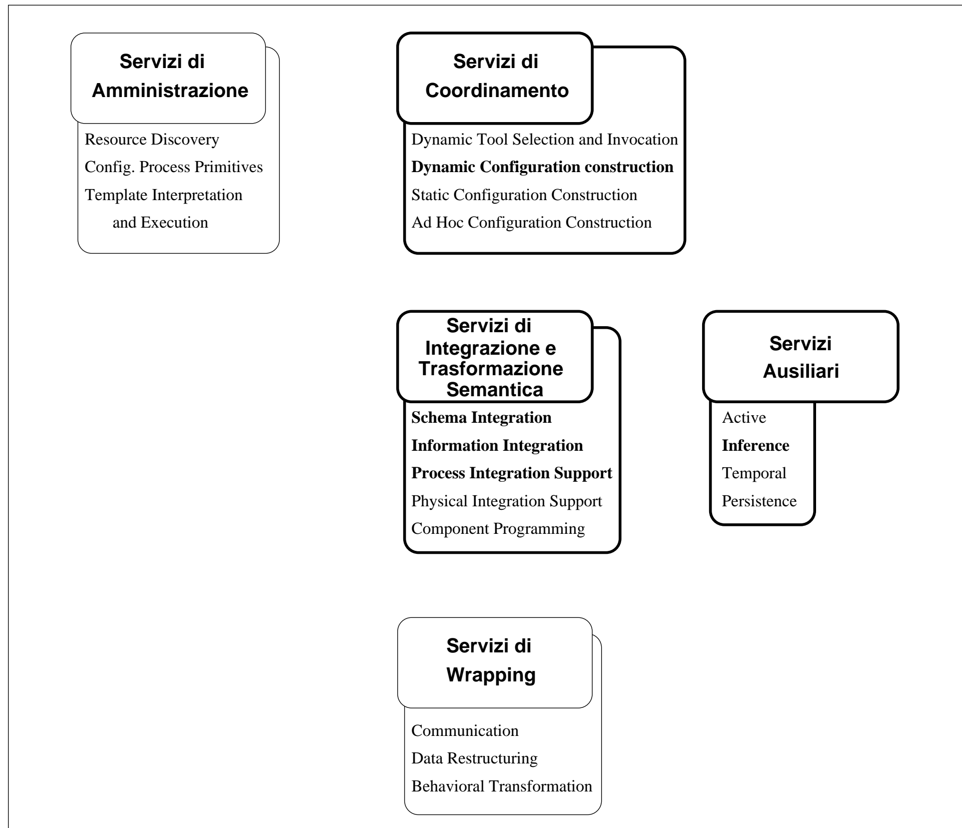

Figura 1.2: Servizi I^3 ppresenti nel mediatore

3. sorgenti: sono le fonti di informazioni che devono essere integrate dal sistema. Esse costituiscono il livello più basso della struttura e si prevede che possono essere dei database tradizionali (sia ad oggetti, sia basati sul modello relazionale), oppure dei semplici file system. Le sorgenti non saranno accedute in modo diretto ma verranno gestite da moduli software (Wrapper) in grado di convertire le richieste del mediatore in una forma comprensibile dalla sorgente, e le informazioni da essa estratte 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 le funzionalitèsaminate, in questo e nei precedenti lavori, sono le seguenti:

- 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 (Query Decompo-

sition), ed eventualmente di scomporre la richiesta in un insieme di sotto-interrogazioni diverse da inviare alle differenti fonti di informazione (Query Transformation);

- 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 interrogazione (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;

Sebbene in un primo momento l'attenzione fosse esclusivamente rivolta alla gestione sorgenti tradizionali o strutturate, successivamente si è cercato di estendere il contesto applicativo di **MOMIS** in modo da poter trattare anche dati "*multimediali*".
La necessità di gestire dati semistrutturati [10] deriva dall'incredibile aumento dei formati in cui i dati possono essere rappresentati, tuttavia la loro trattazione apre problematiche di complessa soluzione. Questi dati sono infatti caratterizzati dall'avere una struttura estremamente irregolare, non riconducibile ad uno schema preciso, quindi si rende necessario individuare un modello idoneo alla loro trattazione.

1.2.1 Scelte implementative

In letteratura sono stati presentati diversi "*approcci*" al problema dell'integrazione di database convenzionali, come pure applicabili all'integrazione di dati semistrutturati. Per comprendere come MOMIS possa essere collocato rispetto ad altri sistemi esistenti, o in fase di sviluppo, è opportuno fare un'analisi di tali approcci.

Le proposte di integrazione Di tali approcci, seguendo quanto esposto in [11], si può realizzare una prima categorizzazione sulla base del diverso approccio utilizzato per la risoluzione dei conflitti semantici.
Ad un'estremo dello spettro di soluzioni troviamo una proposta di standardizzazione dei database e della rappresentazioni dei dati, come ad esempio vien fatto in SAP [12] . Tale proposta si basa quindi sulla definizione di un *modello globale dei dati* mediante il quale deve essere fatta un reingegnerizzazione dei sistemi locali. Questa strada comporta ingenti investimenti ovviamente ed un forte grado di collaborazione e interazione, presupposti questi che possono essere trovati solo in contesti caratterizzati da un forte accentramento amministrativo.

All'estremo opposto troviamo sistemi che risolvono le eterogeneità presenti nelle sorgenti fornendo strumenti in grado di fornire all'utente esterno una visione omogenea degli schemi e delle informazioni. Questa soluzione preserva quindi l'autonomia delle sorgenti ed è l'unica strada percorribile quando è necessario un elevato grado di indipendenza dei singoli database.

Concentrando l'attenzione sul secondo tipo di approccio è possibile raffinare la classificazione sulla base del modo in cui vengono descritte le sorgenti ed i dati in esse contenuti. Come descritto in [13] è possibile distinguere tra approcci *semantici* e *strutturali*.

Per quanto riguarda l'approccio *strutturale* (e tra questi l'esempio piú importante è senza dubbio costituito dal progetto TSIMMIS [1]) possiamo sottolineare l'impiego di un self-describing model per rappresentare gli oggetti da integrare, limitando l'uso delle informazioni semantiche a delle 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. I punti caratterizzanti di tale approccio sono quindi:

- utilizzo di un modello autodescrittivo per trattare tutti i singoli oggetti presenti nel sistema, sopperendo all'eventuale mancanza degli schemi concettuali delle diverse sorgenti;
- inserimento delle informazioni semantiche in modo esplicito attraverso l'impiego di regole dichiarative (ed in particolare, in TSIMMIS, attraverso le MSL rule);
- utilizzo di un linguaggio self-describing che facilita l'integrazione anche e soprattutto di dati semi-strutturati;

Come è facile intuire, in questo modo si ha la possibilità di integrare in modo completamente trasparente al mediatore basi di dati fortemente eterogenee e magari mutevoli nel tempo. Oggetti simili provenienti da una stessa sorgente possono avere strutture differenti rendendo quindi particolarmente la trattazione di dati semistrutturati.
D'altro canto però l'assenza di schemi concettuali vincola le possibili interrogazioni ad un insieme predefinito dall'operatore (per queste viene preventivamente memorizzato un piano di accesso), limitando in questo modo la libertà di richieste all'utente del sistema ed inoltre non permette, in caso di database di grandi dimensioni, la realizzazione di una ottimizzazio-

ne semantica. L'approccio *semantico* è invece caratterizzato dai seguenti aspetti:

- il mediatore dispone, per ogni sorgente, di uno schema concettuale;
- nello schema concettuale sono presenti oltre ai metadati anche informazioni semantiche che possono essere sfruttate sia nella fase di integrazione delle sorgenti, sia in quella di ottimizzazione delle interrogazioni;
- deve essere disponibile un modello comune per descrivere le informazioni da condividere (e dunque per descrivere anche i metadati);
- viene realizzata un'unificazione (parziale o totale) degli schemi concettuali per arrivare alla definizione di uno schema globale.

Lo *schema globale* sopra citato rappresenta una *vista integrata* delle sorgenti e tale vista può essere realizzata seguendo due approcci distinti, quello *materializzato* e quello *virtuale* [11].
La prima soluzione, adottata nei data warehouse [14], le informazioni vengono raccolte in un database centralizzato, quindi le interrogazioni possono essere eseguite senza dover accedere alle sorgenti interrogazioni. Sebbene i tempi di risposta siano decisamente contenuti, la necessità di mantenere l'allineamento tra vista globale e sorgenti impone la definizione e l'impiego di complesse procedure per l'aggiornamento dei dati.
La seconda strategia si basa invece su un modello di *decomposizione* delle query che, analizzando le richieste, porti all'individuazione delle sorgenti "*interessanti*" e alla generazioni di sotto-interrogazioni che possano essere eseguite localmente. Lo schema globale deve poi disporre di tutte le informazioni necessarie alla ricombinazione, o *fusione*, dei dati ricevuti, in modo da ottenere informazioni significative, cioè al contempo complete e corrette.

L'approccio adottato In base alla classificazione fatta possiamo dire che MOMIS segue un approccio *semantico* e *virtuale*. Partendo dagli schemi concettuali locali, con una metodologia *bottom up*, si arriva infatti a definire uno schema globale in grado di fornire un accesso integrato alle sorgenti. Tale schema è quindi arricchito di tutte quelle informazioni che permettono l'individuazione dei dati ed il loro reperimento direttamente dalle fonti di informazione.

Diverse sono le motivazioni che hanno spinto all'adozione di un approccio di questo tipo:

1. la presenza di una schema globale permette all'utente di formulare qualsiasi interrogazione che sia consistente con lo schema;
2. le informazioni semantiche in esso comprese nello schema globale possono contribuire ad una eventuale ottimizzazione delle interrogazioni;
3. l'adozione di una semantica *type as a set* per gli schemi permette di controllarne la consistenza, facendo riferimento alle loro descrizioni;
4. la vista virtuale rende il sistema estremamente flessibile, in grado cioè di sopportare frequenti cambiamenti sia nel numero e tipo di sorgenti, sia nei loro contenuti (non occorre prevedere onerose politiche di allineamento);

Parallelamente a questa impostazione si è deciso di adottare, sia per la rappresentazione degli schemi che per la formulazione delle interrogazioni, un unico modello dei dati basato sul paradigma ad oggetti. Questa scelta è stata fatta per diverse ragioni:

1. la necessitd̀i disporre di un linguaggio di interrogazione espressivo, in grado cioè di rappresentare i concetti di alto livello fondamentali per l'estrazione di conoscenza da insieme di dati;
2. la natura stessa degli schemi che utilizzano i modelli ad oggetti, attraverso l'uso delle primitive di generalizzazione e di aggregazione, permette la riorganizzazione delle conoscenze estensionali;
3. ampi sforzi sono sono già stati realizzati per lo sviluppo di standard rivolti agli oggetti: CORBA [15] per lo scambio di oggetti attraverso sistemi diversi; ODMG-93 [16] (e con esso i modelli ODM e ODL per la descrizione degli schemi, e OQL come linguaggio di interrogazione) per gli object-oriented database;
4. l'adozione di una semantica di *mondo aperto* permette il superamento delle problematiche legate all'uso di un convenzionale modello ad oggetti per la descrizione di dati semistrutturati: gli oggetti di una classe condividono una struttura minima comune (che è quindi la descrizione della classe stessa), ma possono avere ulteriori proprietà non esplicitamente comprese nella struttura della classe di appartenenza.
5. la possibilità di tradurre, in modo automatico, i modelli ad oggetti in logiche descrittive (ad esempio OLCD) permette l'introduzione di comportamenti intelligenti di supporto all'operatore sia nella fase di integrazione sia in quella di interrogazione.

Queste scelte sono poi state tradotte in un modello dei dati ed in un'architettura.

1.2.2 Il Modello dei dati

Come si è detto all'interno del sistema è stato adottato un modello comune dei dati (ODM_{I^3}) di alto livello in modo da facilitare la comunicazione tra i Wrapper ed il Mediatore.
La base di partenza per la definizione di questo modello è rappresentata dalle raccomandazioni relative alla proposta di standardizzazione per linguaggi di mediazione, risultato del lavoro svolto in ambito I^3. Tali raccomandazioni sottolineano la necessità per un mediatore di poter gestire sorgenti con modelli complessi, come quello ad oggetti, e sorgenti molto più semplici come file di strutture ed individuano come possibile soluzione l'impiego di un formalismo il più possibile completo e quindi in grado di rappresentare in modo adeguato tutte le possibili situazioni.

Per quanto riguarda il linguaggio di definizione degli schemi si è cercato di cogliere le indicazioni emerse in ambito I^3discostandosi, nel contempo, il meno possibile dal linguaggio ODL proposto dal gruppo di standardizzazione ODMG-93. Si è così definito il linguaggio ODL_{I^3} come estensione del linguaggio standard ODL in modo da supportare le necessità del nostro mediatore.

Le principali caratteristiche del linguaggio ODL_{I^3}sono:

- possibilità di rappresentare sorgenti strutturate (database relazionali, ad oggetti, e file system) e semistrutturate. Ciò significa che tutte le fonti di informazione, indipendentemente dal modello originario, e lo schema globale verranno descritti mediante il modello comune, facendo quindi riferimento al concetto di classe ed aggregazione (sarà poi compito dei Wrapper provvedere alla traduzione in termini del modello originale);

- dichiarazione di regole di integrità (*if then rule*), definite sia sugli schemi locali (e magari da questi ricevute), che riferite allo schema globale, e quindi inserite dal progettista del mediatore;

- dichiarazione di regole di mediazione, o *mapping rule*, utilizzate per specificare l'accoppiamento tra i concetti globali e i concetti locali originali;

- utilizzo della *semantica di mondo aperto*, che permette alle classi descritte di cambiare formato (magari aggiungendo attributi agli oggetti) senza necessariamente cambiarne la descrizione (prerogativa, questa, indispensabile per la gestione di sorgenti semistrutturate).

- traduzione automatica e trasparente all'utente delle descrizioni nella logica descrittiva **OLCD**, con conseguente possibilitá di utilizzare comportamenti intelligenti nei controlli di consistenza e nell'ottimizzazione semantica delle interrogazioni;

- introduzione dell'operatore di *unione* (`union`), che permette l'espressione di strutture dati in alternativa nella definizione di una classe;

- introduzione del costruttore *optional* (*), specificato dopo il nome di un attributo per indicare la sua natura opzionale;

- dichiarazione di relazioni terminologiche, che permettono di specificare relazioni di sinonimia (SYN), ipernimia (BT), iponomia (NT) e relazione associativa (RT) tra due tipi.

Utilizzando questo linguaggio, il wrapper compie la traduzione delle classi da integrare e le fornisce al mediatore: da sottolineare che le descrizioni ricevute rappresentano tutte e sole le classi che una determinata sorgente vuole mettere a disposizione del sistema, e quindi interrogabili. Non è quindi detto che lo schema locale ricevuto dal mediatore rappresenti l'intera sorgente, bensì ne descrive il sottoinsieme di informazioni visibili da un utente del mediatore, esterno quindi alla sorgente stessa.

Anche nella definizione del linguaggio di interrogazione (OQL_{I^3}) si è deciso di adottare la sintassi OQL senza discostarsi dallo standard.
Questo linguaggio è estremamente versatile ed espressivo perciò se da un lato sarà necessario uno sforzo maggiore nello sviluppo di moduli per l'interpretazione e gestione delle interrogazioni (implementando le funzionalità proprie di un ODBMS), dall'altro si avrà la possibilità di sfruttare al meglio le informazioni rappresentate nello schema globale. Le inevitabili complicazioni a livello implementativo sono, pertanto, ampiamente giustificate da una maggiore versatilità e facilità d'uso, per l'utente (si impiega infatti un linguaggio standard e non un formalismo ad-hoc). Le principali caratteristiche di questo linguaggio sono:

- OQL è basato sul modello ad oggetti definito da ODMG.

- OQL utilizza una sintassi simile a quella definita per SQL 92. Rispetto a questa presenta delle estensioni finalizzate alla gestione degli aspetti object-oriented, in particolare gli oggetti complessi, l'identità degli oggetti, le espressioni di percorso, il polimorfismo, l'invocazione delle operazioni e il late binding.

- OQL fornisce delle primitive ad alto livello per manipolare insiemi di oggetti, ma non è strettamente legato a questo costrutto. Fornisce infatti anche le primitive per gestire, con la stessa efficienza, altri tipi strutturati come array, liste e strutture.

- OQL non fornisce in modo esplicito operatori per l'aggiornamento della base di dati, lasciando questo compito a operazioni opportune che fanno

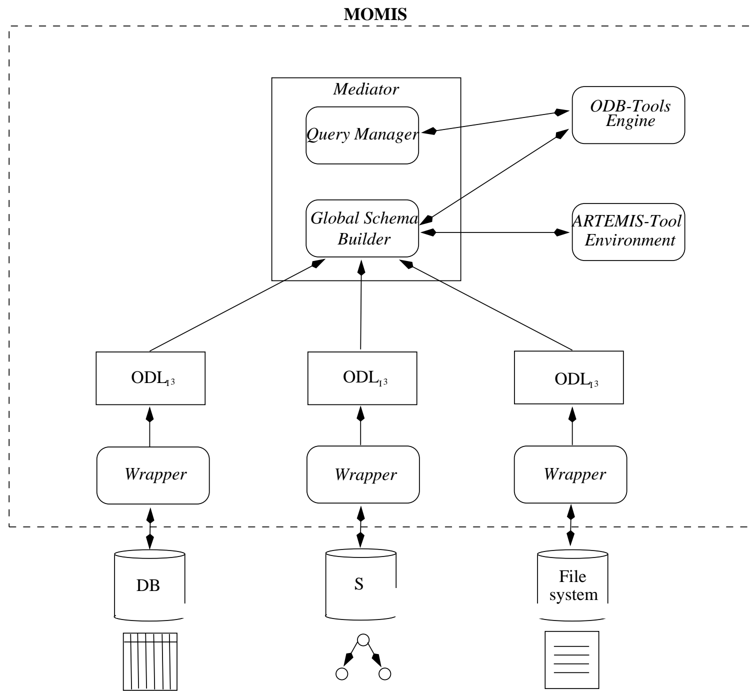

Figura 1.3: Architettura del sistema MOMIS

parte delle caratteristiche di ogni oggetto che popola il database. In questo modo si rispetta la semantica propria degli ODBMS che, per definizione, vengono gestiti attraverso i metodi definiti sugli oggetti.

- OQL permette di accedere agli oggetti in maniera dichiarativa. Questa caratteristica rende più immediata la formulazione dell'interrogazione da parte dell'utente e più semplice ottimizzare le interrogazioni.

1.2.3 L'architettura di MOMIS

In MOMIS, che ricordo adotta l'architettura di riferimento I^3 [6], si possono distinguere quattro componenti principali (come si puó vedere da Fig. 1.3):

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 ODL$_{I^3}$ (descritto in Sezione 1.2.2);

- in fase di query processing, traducono la query ricevuta dal mediatore (espressa quindi nel linguaggio comune di interrogazione OQL_{I^3}, che è 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, in grado di fornire una rappresentazione omogenea delle informazioni ed un accesso integrato alle sorgenti. Il Mediatore è composto da tre 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. Questa fase di integrazione, realizzata in modo semiautomatico con l'interazione del progettista del sistema, fa uso degli ODB-Tools (sfruttati dal modulo software $\textbf{SIM}_1$) e delle tecniche di clustering;
   - Extensional Hierarchy Builder (EHB): questo modulo è stato progettato per individuare e elaborare le relazioni estensionali tra classi locali. I risultati prodotti consentono un notevole miglioramento nell'efficienza e nella qualità delle risposte prodotte dal Query Manager.
   - 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 [17, 18] che compie la validazione di schemi e l'ottimizzazione di query [19, 20, 21]. Le caratteristiche di questo strumento verranno brevemente presentate nella sezione successiva.

4. *ARTEMIS-Tool Environment*, un tool che compie analisi e clustering di schemi [22].

Mediante questa struttura e queste funzionalità MOMIS è quindi in grado di 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.

Pur non essendo centrale per il questo sistema, è opportuno descrivere il processo che rende disponibile al Mediatore gli schemi sorgenti in linguaggio ODL_{I^3}. Per quanto riguarda i database strutturati convenzionali, la descrizione dello schema é sempre disponibile e puó quindi essere tradotto automaticamente in linguaggio ODL_{I^3} da uno specifico wrapper. Diverso é il discorso delle sorgenti semistrutturate, dove, a priori, non é definito alcuno schema. Per questa ragione risulta necessaria una procedura di estrazione dello schema per le sorgenti semistrutturate (e.g., nella forma di dataguides [23]). In MOMIS introduciamo perció il concetto di *object pattern* per rappresentare a livello intensionale la struttura degli oggetti di una sorgente semistrutturata [24].

1.3 ODB-Tools Engine

Uno degli aspetti più innovativi di MOMIS è rappresentato dall'impiego di logiche descrittive e tecniche di intelligenza artificiale sia in fase di costruzione della vista globale sia nell'ottimizzazione semantica delle interrogazioni.
Questi comportamenti intelligenti sono stati introdotti utilizzando ODB-Tools che è uno strumento software, sviluppato presso il Dipartimento di Ingegneria dell'Università di Modena [19, 21], per la validazione di schemi e l'ottimizzazione semantica di interrogazioni per le Basi di Dati Orientate agli Oggetti (OODB). Gli algoritmi operanti in ODB-Tools sono basati su tecniche di inferenza che sfruttano il calcolo della *sussunzione* e la nozione di *espansione semantica* di interrogazioni per la trasformazione delle query al fine di ottenere tempi di risposta inferiori. Il primo concetto è stato introdotto nell'area dell'Intelligenza Artificiale, più precisamente nell'ambito delle Logiche Descrittive, il secondo nell'ambito delle Basi di Dati. Questi concetti sono stati studiati e formalizzati in **OLCD** (*Object Languages with Complements allowing Descriptive cycles*), una logica descrittiva per basi di dati. Come interfaccia verso l'utente esterno è stata scelta la proposta ODMG-93 [25], utilizzando il linguaggio ODL (Object Definition Language) per la definizione degli schemi ed il linguaggio OQL (Object Query Language) per la scrittura di query. Entrambi i linguaggi sono stati estesi per la piena compatibilità al formalismo **OLCD**(**O**bject **L**ogics for **C**onstraint **D**escription).

1.3.1 La Logica Descrittiva OLCD

Le Logiche Descrittive (DLs) [26, 27] costituiscono una restrizione delle $1^{st}OPL$ [28]: esse consentono di esprimere i *concept*[1] sotto forma di formule logiche, usando solamente predicati unari e binari, contenenti solo una variabile

[1] Un concept è una struttura del tutto simile ad una classe

(corrispondente alle istanze del concept). Un grande vantaggio offerto dalle DLs, per le applicazioni di tipo DBMS, è costituito dalla capacità 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. Questa capacità deriva dal fatto che, tanto le DLs quanto i CODM, si riferiscono esclusivamente agli aspetti *strutturali*: tipi, valori complessi, oggetti con identità, classi, ereditarietà. Unendo quindi le caratteristiche di similarità coi CODM e le tecniche di inferenza tipiche delle $1^{st}OPL$ si raggiunge l'ambizioso obiettivo di dotare i sistemi di basi di dati di componenti intelligenti per il supporto alle attività di design, di ottimizzazione e, come sarà illustrato nei prossimi capitoli, di integrazione di informazioni poste in sorgenti eterogenee.

Il formalismo **OLCD** deriva dal precedente **ODL**, proposto in [29], che estende l'espressività dei linguaggi sviluppati nell'area delle DLs. La caratteristica principale di **OLCD** 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 considerata anche la possibilità di utilizzare dei sottoinsiemi di questi (come potrebbero essere, ad esempio, intervalli di interi). A partire da questi tipi di base si possono definire i *tipi valore*, attraverso gli usuali costruttori di tipo definiti nei *CODMs*, quali *tuple*, *insiemi*, *liste* e *tipi classe*, che denotano insiemi di oggetti con una identità ed un valore associato. Ai tipi può 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*). L'ereditarietà, sia semplice che multipla, è espressa direttamente nella descrizione di una classe tramite l'operatore di *intersezione*.

OLCD 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], **OLCD** è stato esteso per permettere la formulazione dichiarativa di un insieme rilevante di vincoli di integrità definiti sulla base di dati. Attraverso questo logica descrittiva, è possibile descrivere, oltre alle classi, anche le *regole di integrità*: permettono la formulazione dichiarativa di un insieme rilevante di vincoli di integrità sotto forma di regole *if-then* i cui antecedenti e conseguenti sono espressioni di tipo **OLCD**. In tale modo, è possibile descrivere correlazioni tra proprietà strutturali della stessa classe, o condizioni sufficienti per il popolamento di sotto-classi di una classe data. In altre parole le rule costituiscono uno strumento dichiarativo per descrivere gli oggetti che popolano il sistema.

Per la rappresentazione dei vincoli di integrità è stata introdotta una sintassi intui-

tiva coerente con la proposta ODMG-93. In particolare si sono sfruttati il costrutto `for all`, il costrutto `exists`, gli operatori booleani e i predicati di confronto utilizzati nell'OQL.

Una regola di integrità è dichiarata attraverso la seguente sintassi:

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

Le `condizioni`, antecedente e conseguente, hanno la medesima forma e sono costituite da una lista di espressioni booleane in `and` tra loro; all'interno di una condizione, attributi e oggetti sono identificati mediante la *dot notation*. Nello schema di riferimento `Cardiology_Department`, ad esempio, possiamo introdurre le seguenti regole di integrità:

```
rule rule_1 forall X in Patient: X.therapy = "Kemiotherapy"
   then X.room >= 100 ;

rule rule_3 forall X in Patient:
                    exists S in X.exam: S.outcome = "Heart risk"
   then X.therapy = "Intensive" ;
```

La `rule_1` stabilisce che un paziente sottoposto ad un trattamento kemioterapico deve essere ricoverato nelle stanze dalla centesima in poi. La `rule_3` asserisce che i pazienti che hanno avuto *almeno* un esame con esito "$Heart\ risk$" devono essere sottoposti ad una terapia intensiva.

1.3.2 Le regole OLCD e l'espansione semantica di un tipo

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, sia 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*, cioé 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 (l'interrogazione 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 alla iniziale. In questo modo l'interrogazione 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 fisico.

1.3.3 Validazione e Sussunzione

Uno dei problemi principali che il progettista di una base di dati deve affrontare è quello della consistenza delle classi introdotte nello schema. Infatti, molti modelli e linguaggi di definizione dei dati sono sufficientemente espressivi da permettere la rappresentazione di classi inconsistenti, cioè classi che non potranno contenere alcun oggetto della base di dati. Tale eventualità sussiste anche in **OLCD**: ad esempio, la possibilità di esprimere intervalli di interi permette la dichiarazione di classi con attributi omonimi vincolati a intervalli disgiunti. Il prototipo rivela durante la fase di validazione dello schema come inconsistente una eventuale congiunzione di tali classi. Il concetto di *sussunzione* esprime invece la relazione esistente tra due classi di oggetti quando l'appartenenza di un oggetto alla seconda comporta necessariamente l'appartenenza alla prima. La relazione di sussunzione può essere calcolata automaticamente tramite il confronto sintattico tra le descrizioni delle classi; l'algoritmo di calcolo è stato presentato in [35]. Poiché accanto

alle relazioni di ereditarietà definite esplicitamente dal progettista possono esistere altre relazioni di specializzazione implicite, queste possono essere esplicitate dal calcolo della relazione di sussunzione presenti nell'intero schema: il prototipo, dopo aver verificato la consistenza di ciascuna classe, determina tali relazioni di specializzazione implicite fornendo un valido strumento inferenziale per l'utente progettista della base dei dati.

1.3.4 Ottimizzazione semantica delle interrogazioni

L'ottimizzazione semantica di una interrogazione utilizza il processo di *espansione semantica*, attraverso il quale viene generata una interrogazione equivalente che incorpora ogni possibile restrizione non presente nell'interrogazione originale e tuttavia *logicamente implicata* dallo schema globale del database (classi + tipi + regole d'integrità). L'algoritmo di calcolo, di complessità polinomiale, è stato proposto in [36] e rappresenta il motore del modulo di ottimizzazione del prototipo.

Il linguaggio di interrogazione utilizzato è compatibile con OQL (Object Query Language), proposto in ODMG-93, sia per l'input delle query sia per l'output restituito dopo l'ottimizzazione. Una interrogazione espressa in OQL potrà beneficiare del processo di ottimizzazione semantica se può essere tradotta, in modo equivalente, in una espressione di tipo di **OLCD**: vista la ricchezza di tale formalismo, un insieme rilevante di interrogazioni può essere ottimizzato. In particolare, si riescono a trattare interrogazioni riferite ad una singola classe con condizioni espresse sulla gerarchia di aggregazione. Tuttavia, poiché il linguaggio di interrogazione OQL è più espressivo del formalismo **OLCD**, introduciamo, seguendo l'approccio proposto in [37], una separazione ideale dell'interrogazione in una parte *clean*, che può essere rappresentata come tipo in **OLCD**, e una parte *dirty*, che va oltre l'espressività del sistema di tipi; l'ottimizzazione semantica sarà effettuata solo sulla parte *clean*.

Allo scopo di evidenziare le trasformazioni effettuate dal processo di ottimizzazione semantica, l'interrogazione viene riportata in uscita differenziandone graficamente i vari fattori sulla base della seguente classificazione:

- fattori specificati dall'utente e non modificati (`stampati`)
- fattori modificati o introdotti dall'ottimizzatore (<u>sottolineati</u>)
- fattori dirty, specificati dall'utente ma non trattati dall'ottimizzatore (*corsivati*)

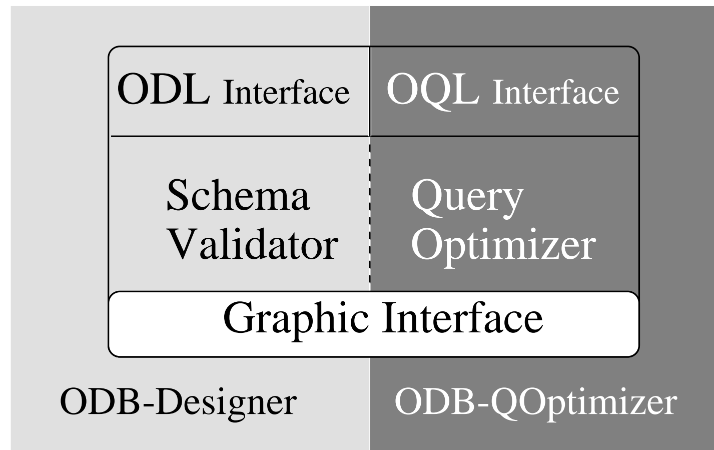

Figura 1.4: Architettura di ODB-Tool

1.3.5 Architettura di ODB-Tools

ODB-Tool, sviluppato al Dipartimento di Scienze dell'Ingegneria dell'Università di Modena, è un prototipo software per la validazione di schemi e l'ottimizzazione di interrogazioni in ambiente OODB. L'architettura, mostrata in figura 1.4, presenta i vari moduli integrati che definiscono un ambiente *user-friendly* basato sul linguaggio standard ODMG-93. L'utente inserisce gli schemi in linguaggio ODL e le query in OQL ottenendo come risultato la validazione dello schema, l'ottimizzazione dell'interrogazione (in OQL) e la rappresentazione grafica della gerarchia di ereditarietà e di aggregazione dello schema(queste funzionalità sono visibili ed utilizzabili attraverso il sito web *http://www.sparc20.dsi.unimo.it*).

Vediamo in dettaglio la descrizione di ciascun modulo:

- **ODL Interface**
  È il modulo di input degli schemi. Accetta la sintassi ODL e trasforma le classi in descrizioni native del formalismo **OLCD**.

- **OQL Interface**
  È il modulo di input e output delle interrogazioni. Utilizza il linguaggio OQL sia per l'input che per l'output della query ottimizzata, sia viene trasformata in descrizioni del formalismo **OLCD**. I predicati booleani in output sono differenziati a seconda del proprio significato:

  – i fattori introdotti o modificati dall'ottimizzazione sono mostrati in colore rosso

  – i fattori non modificati vengono mostrati in colore grigio

  – i fattori ignorati vengono mostrati in colore nero

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

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

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

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

I moduli di interfaccia, validazione ed ottimizzazione sono stati realizzati in linguaggio C, utilizzando il compilatore gcc 2.7.2, i generatori flex 2.5 e bison 1.24, mentre il modulo di rappresentazione grafica è stato realizzato in JAVA, utilizzando il compilatore JDK 1.1. La piattaforma utilizzata è una SUN SPARCSTATION 20, con sistema operativo Solaris 2.5.

Capitolo 2

Integrazione intensionale di schemi

Il principale scopo che ci si è preposti con MOMIS è la realizzazione di un sistema di mediazione versatile ed efficiente, capace di assistere l'utente nel reperimento di informazioni su di un insieme estremamente diversificato di sorgenti.
Tale obiettivo è stato conseguito agendo in due direzioni. Da un lato sono stati progettati strumenti in grado di assistere il progettista nella complessa fase di integrazione degli schemi e dall'altro si è realizzato un Query Manager che, posta un'interrogazione sulla vista globale, automatizza il processo di reperimento ed integrazione delle informazioni.
Per garantire che la risposta fornita sia al contempo corretta, completa e minima, entrambe questa fasi sfruttano un insieme di conoscenze relative alle sovrapposizioni delle estensioni delle classi, alle relazioni tra le intensioni e alla semantica degli schemi. Tale conoscenza è in parte fornita dal progettista ed in parte ricavata dalle informazioni implicitamente rappresentate negli schemi locali, tuttavia occorre sottolineare che 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 tanto meno 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.

Vediamoli più in dettaglio.

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 [38, 39], 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 sia 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). A tale scopo, come si vedrà nel seguito, si potrebbe utilizzare sistemi lessicali, quali, ad esempio, WordNet [40, 41].

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 [7] 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 cioé 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.

2.1 Processo di Integrazione

Schemi locali parzialmente sovrapposti possono rappresentare uno stesso concetto adottando strutture diverse, pertanto il presupposto fondamentale per la realizzazione di un Mediatore consiste nella capacità di individuare e risolvere questi conflitti, che possono essere definiti "$intensionali$".
Il modulo GSB impiega logiche deduttive, proprie dell'intelligenza artificiale, per rendere semi-automatica la fase di integrazione delle sorgenti informative (sia strutturate sia semistrutturate). Basandosi sulle descrizioni degli schemi locali in linguaggio ODL$_{I^3}$ vengono quindi utilizzate tecniche di Description Logics e clustering per arrivare alla definizione di uno schema globale completamente privo di dati ma contenente tutte le informazioni necessarie al loro reperimento. Le fasi del processo di integrazione sono:

1. *Estrazione di relazioni terminologiche*, grazie al supporto di ODB-Tools. Durante questo passo viene costruito un Thesaurus Comune di *relazioni terminologiche*. Le relazioni terminologiche esprimono la conoscenza di inter-schema su sorgenti diverse e corrispondono alle asserzioni intensionali utilizzate in [42]. Le relazioni terminologiche sono derivate in modo semi-automatico a partire dalle descrizioni degli schemi in ODL$_{I^3}$, attraverso l'analisi strutturale e di contesto delle classi coinvolte, utilizzando ODB-Tools e le tecniche di Description Logics. L'estrazione delle relazioni terminologiche sará discussa in Sezione 2.2.

2. *Affinity-based clustering 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 affinitá tra le classi ODL$_{I^3}$ in modo di identificare le informazioni da essere integrate a livello globale. A tal fine, ARTEMIS calcola i coefficienti che misurano il livello di affinitá delle classi ODL$_{I^3}$ basandosi sia sui nomi sia sugli attributi. Le classi ODL$_{I^3}$ con maggiore affinitá vengono raggruppate utilizzando le tecniche di clustering [43].

3. *Costruzione dello schema globale di mediatore*, con il supporto di ODB-Tools i cluster di classi ODL$_{I^3}$ affini sono analizzate per costruire lo schema globale del Mediatore. Per ciascun cluster viene definita una classe globale

Sorgente University (S_1)

```
Research_Staff(first_name,last_name,relation,email,
               dept_code,section_code)
School_Member(first_name,last_name,faculty,year)
Department(dept_name,dept_code,budget,dept_area)
Section(section_name,section_code,length,room_code)
Room(room_code,seats_number,notes)
```

Sorgente Computer_Science (S_2)

```
CS_Person(name)
Professor:CS_Person(title,belongs_to:Division,rank)
Student:CS_Person(year,takes:set⟨Course⟩,rank)
Division(description,address:Location,fund,sector,employee_nr)
Location(city,street,number,county)
Course(course_name,taught_by:Professor)
```

Sorgente Tax_Position (S_3)

```
University_Student(name,student_code,faculty_name,tax_fee)
```

Figura 2.1: Esempio di riferimento

ODL$_{I^3}$ che rappresenta tutte le classi che sono riferite al cluster, e che é caratterizzata dall'unione dei loro attributi. L'insieme delle classi globali definite costituisce lo schema globale di Mediatore che deve essere usato per porre le query alle sorgenti locali integrate. In questa fase **OLCD** e ODB-Tools sono utilizzati per realizzare una generazione semi-automatica delle classi globali.

Nelle prossime sezioni verranno approfondite le fasi sopra descritte, utilizzando il seguente esempio per meglio illustrare tutti i passaggi.

Esempio di riferimento L'esempio si riferisce ad una realtà universitaria: le sorgenti da integrare sono tre. La prima sorgente, `University` (S_1), è un database di tipo relazionale, che contiene informazioni sullo staff e sugli studenti di una determinata università. È composta da cinque tabelle: `Research_Staff`, `School_Member`, `Department`, `Section` e `Room`. Per ogni professore (presente nella tabella `Research_Staff`), sono memorizzate informazioni sul suo dipartimento (attraverso la foreign key `dept_code`), sul suo indirizzo di posta elettronica (`email`), e sul corso da lui tenuto (`section_code`). Per il corso, viene memorizzata pure l'aula (`Room`) dove questo si svolge, mentre del dipartimento sono descritti, oltre al nome (`dept_name`) ed al codice (`dept_code`),

il budget (budget) che ha a disposizione e l'area (dept_area) a cui appartiene, sia essa Scientifica, Economica, ... Per gli studenti presenti nella tabella School_Member sono invece mantenuti il nome (nella coppia first_name e last_name), la facoltà di appartenenza (faculty) e l'anno di corso (year).
La sorgente Computer_Science (S_2) contiene invece informazioni sulle persone afferenti a questa facoltà, ed è un database ad oggetti. Sono presenti sei classi: CS_Person, Professor, Student, Division, Location e Course. I dati mantenuti sono comunque abbastanza simili a quelli della sorgente S_1: per quanto riguarda i professori, sono memorizzati il titolo (title), e la divisione di appartenenza (belongs_to), che a sua volta fa parte di un dipartimento (e ne può quindi essere considerata una specializzazione); per gli studenti sono memorizzati i corsi seguiti (takes) e l'anno di corso (year). Il corso ha poi un attributo complesso che lo lega al professore che ne è titolare (taught_by), mentre per la divisione si tiene l'indirizzo (address), i fondi (fund) e il numero di impiegati (employee_nr).
È presente inoltre una terza sorgente, Tax_Position (S_3), facente capo alla segreteria studenti, che mantiene i dati relativi alle tasse da pagare (tax_fee). In questo caso (S_3), non si tratta di un database ma di un file system, che contiene quindi semplici tracciati record.

2.2 Estrazione di Relazioni Terminologiche

Lo scopo di questa fase è la costruzione di un Thesaurus di *relazioni terminologiche* che rappresenti la conoscenza a disposizione sulle classi da integrare (ovvero sui nomi delle classi, sugli attributi) e che sarà la base per il calcolo di affinità tra le classi stesse. Definiamo quindi un modello di rappresentazione delle classi. Sia $S = \{S_1, S_2, \ldots, S_N\}$ un insieme di schemi di N sorgenti eterogenee che devono essere integrate. Come richiesto dall'ODL$_{I^3}$, ogni schema sorgente S_i è composto da un insieme di *classi*: una classe $c_{ji} \in S_i$ è caratterizzata da un nome e da un insieme di attributi, $c_{ji} = \langle n_{c_{ji}}, A(c_{ji}) \rangle$. A sua volta, ogni attributo $a_h \in A(c_{ji})$, con $h = 1, \ldots, n$, è definito da una coppia $a_h = \langle n_h, d_h \rangle$, dove n_h è il nome e d_h è il dominio associato ad a_h. Si è inoltre ipotizzato che, per identificare in modo univoco all'interno del mediatore un nome (sia esso di attributo, sia esso di classe), sia rispettivamente necessaria la coppia *nome_sorgente*,*nome classe* e *nome_sorgente*,*nome_attributo*. Si parlerà inoltre genericamente di *termine* t_i indicando con esso un nome di classe o di attributo.

Le relazioni che si possono definire all'interno del Thesaurus sono le seguenti:

- SYN (synonym-of): definita tra due termini t_i e t_j, con $t_i \neq t_j$, che sono considerati sinonimi, ovvero che possono essere interscambiati nelle sorgenti,

identificando lo stesso concetto del mondo reale. Un esempio di relazione SYN nel nostro esempio è ⟨`SU.Section` SYN `SCS.Course`⟩. SYN é simmetrica, cioé, t_i SYN $t_j \Rightarrow t_j$ SYN t_i.

- BT (broader-term): definita tra due termini t_i e t_j tali che t_i ha un significato più ampio, più generale di t_j. Un caso di BT, nel nostro esempio, può essere ⟨`SU.Research_Staff` BT `SCS.Professor`⟩.

- NT (narrower-term): concettualmente è la stessa relazione espressa con una BT, intesa dall'altro punto di vista, dunque t_i BT $t_j \rightarrow t_j$ NT t_i. Lo stesso esempio potrebbe infatti essere ⟨`SCS.Professor` NT `SU.Research_Staff`⟩.

- RT (related-term): definita tra due termini t_i e t_j che sono generalmente usati nello stesso contesto, tra i quali esiste comunque un legame generico. Per esempio, possiamo avere la seguente ⟨`SCS.Student` RT `SCS.Course`⟩. La relazione é simmetrica.

La scoperta di relazioni terminologiche presenti all'interno degli schemi è un processo semi-automatico, caratterizzato dalla interazione tra il progettista del sistema e gli ODB-Tools. Lo sforzo fatto in questa fase è stato diretto a limitare il più possibile l'intervento dell'operatore, al fine di aumentare la porzione di definizione del Thesaurus realizzabile in modo realmente automatico. L'intero processo che porta, partendo dalle descrizioni degli schemi in ODL$_{I^3}$, alla definizione del un Thesaurus comune si articola in quattro passi.

1. **Estrazione automatica di relazioni dagli schemi sorgenti.** Sfruttando le informazioni semantiche presenti negli schemi strutturati (sia basati sui modelli ad oggetti, sia relazionali) può essere identificato in modo automatico un insieme di relazioni terminologiche. In particolare durante questa preliminare fase di analisi, si puó automaticamente estrarre:

   (a) **schemi ad oggetti**:
      - relazioni BT e NT derivate dalle gerarchie di generalizzazione;
      - relazioni RT derivate dalle gerarchie di aggregazione;

   (b) **schemi relazionali**:
      - relazioni BT e NT derivate dalle gerarchie di generalizzazione (foreign key definita su primary key in entrambe le relazioni;
      - relazioni RT derivate dalle foreign key.

Esempio 1 Considerando l'esempio di riferimento, le relazioni terminologiche automaticamente estratte sono le seguenti:

⟨SCS.Professor NT SCS.CS_Person⟩
⟨SCS.Student NT SCS.CS_Person⟩
⟨SCS.Professor RT SCS.Division⟩
⟨SCS.Student RT SCS.Course⟩
⟨SCS.Division RT SCS.Location⟩
⟨SCS.Course RT SCS.Professor⟩
⟨SU.Research_Staff RT SU.Department⟩
⟨SU.Research_Staff RT SU.Section⟩
⟨SU.Section RT SU.Room⟩

2. **Revisione/Integrazione delle relazioni.** Interagendo con il modulo, il progettista deve inserire tutte le relazioni terminologiche che non sono state estratte nel passo precedente, ma che devono comunque essere presenti per pervenire ad una esatta integrazione delle sorgenti. In particolare, possono essere inserite relazioni che coinvolgono classi appartenenti a schemi diversi, come pure relazioni che si riferiscono a nomi di attributi. Da sottolineare che, durante questa fase, tutte le relazioni inserite dal progettista hanno carattere esclusivamente *terminologico*, escludendo quindi qualunque considerazione sulle estensioni. È quindi una fase che, nei prossimi sviluppi del progetto, potrebbe facilmente essere coadiuvata dall'uso di un dizionario che evidenzi eventuali termini sinonimi, o correlati tra loro. Un approccio potrebbe essere quello di rivolgersi a sistemi lessicali (vedere, per esempio WordNet [40, 41]), che prevedono relazioni terminologiche tra termini a priori.

   In generale, relazioni terminologiche esplicite possono correlare classi ODL_{I^3} le cui descrizioni presentano conflitti semantici rispetto alle relazioni di subsumption/generalizzazione ed equivalenza, poiché queste ultime specificano relazione di supertipo/sottotipo e di uguaglianza tra tipi delle classi coinvolte.

   Per esempio, supponiamo che una relazione di SYN sia definita tra due classi aventi la stessa struttura (per esempio SU.Section e SCS.Course). Per rendere compatibile questa relazione terminologica con una relazione di equivalenza per ODB-Tools é necessario uniformare la descrizione di entrambe le classi. Il risultato della modifica per le classi esaminate diventa:

```
interface Section
( ... )
```

```
{ attribute string  section_name;
  attribute integer section_code;
  attribute integer length;
  attribute integer room_code;
  attribute string  course_name
  attribute Professor taught_by };

interface Course
( ... )
{ attribute string  course_name
  attribute Professor taught_by;
  attribute string  section_name;
  attribute integer section_code;
  attribute integer length;
  attribute integer room_code };
```

Lo stesso tipo di problematica si evidenzia per la relazione di BT tra `SU.Department` e `SCS.Division`. Tradurre questa relazione terminologica nella corrispondente relazione di subsumption implica la modifica delle descrizioni nel seguente modo:

```
interface Department
(...)
{ attribute string  dept_name;
  attribute integer budget;
  attribute integer dept_code;
  attribute string  dept_area };

interface Division
( ...  )
{ attribute string   dept_name;
  attribute integer  budget;
  attribute integer  dept_code;
  attribute string   dept_area;
  attribute string   description;
  attribute Location address;
  attribute integer  found;
  attribute integer  employee_nr;
  attribute string   sector };
```

Infine quando viene asserita una nuova relazione di RT verrà inserito, nella classe alla destra della relazione, un nuovo attributo di aggregazione che

mappa sulla prima classe.

Queste trasformazioni conducono alla generazione automatica di uno ”schema virtuale” contenente, per ciascuno schema locale, le descrizioni modificate delle classi; tale schema viene utilizzato da ODB-Tools per eseguire ulteriori inferenze di relazioni per arricchire il Thesaurus.

Esempio 1 Nel nostro esempio, supponiamo che il progettista inserisca le seguenti relazioni terminologiche relative sia a classi che ad attributi:

⟨SU.Research_Staff BT SCS.Professor⟩
⟨SU.School_Member BT SCS.Student⟩
⟨STP.University_Student BT SCS.Student⟩
⟨SU.Department BT SCS.Division⟩
⟨SU.Section SYN SCS.Course⟩
⟨name BT first_name⟩
⟨name BT last_name⟩
⟨dept_code BT belongs_to⟩
⟨dept_name SYN description⟩
⟨section_name SYN course_name⟩
⟨faculty SYN faculty_name⟩
⟨fund SYN budget⟩
⟨dept_area SYN sector⟩

3. **Validazione delle relazioni.** In questa fase, ODB-Tools viene utilizzato per validare le relazioni terminologiche del Thesaurus definite tra due attributi. La validazione è basata sul controllo di compatibilità dei domini associati agli attributi. In questo modo le relazioni terminologiche vengono distinte in *valide* e *invalide*. In particolare, dati i due attributi $a_t = \langle n_t, d_t \rangle$ e $a_q = \langle n_q, d_q \rangle$ coinvolti in una relazione, verranno eseguiti i seguenti controlli:

   - $\langle n_t$ SYN $n_q \rangle$: la relazione è considerata valida se d_t e d_q sono equivalenti, oppure uno è la specializzzazione dell'altro;
   - $\langle n_t$ BT $n_q \rangle$: la relazione è considerata valida se d_t contiene od è equivalente a d_q;
   - $\langle n_t$ NT $n_q \rangle$: la relazione è considerata valida se d_t è contenuto in, od è equivalente a d_q;

   Nel caso in cui il dominio dell'attributo contiene il connettore di unione, una relazione viene considerata valida quando almeno un dominio risulta compatibile. Questa fase di validazione è compiuta utilizzando ODB-Tools per il calcolo della relazione di subsumption e equivalence.

Esempio 2 Con riferimento al Thesaurus definito nell'esempio 1, riportiamo l'output della fase di validazione: per ciascuna relazione il flag di controllo [1] indica una relazione valida, mentre il flag [0] una invalida.

⟨name BT first_name⟩	[1]
⟨name BT last_name⟩	[1]
⟨dept_code BT belongs_to⟩	[0]
⟨dept_name SYN description⟩	[1]
⟨section_name SYN course_name⟩	[1]
⟨faculty SYN faculty_name⟩	[1]
⟨fund SYN budget⟩	[1]
⟨dept_area SYN sector⟩	[1]

4. **Inferenza di nuove relazioni.** In questa fase vengono inferite nuove relazioni terminologiche utilizzando lo "schema virtuale" definito nella fase di revisione/integrazione e calcolando le nuove relazioni di subsumption e aggregazione. Mediante ODB-Toolsqueste nuove relazioni semantiche vengono tradotte nelle corrispondenti relazioni terminologiche arricchendo il Thesaurus che ora contiene sia le relazioni esplicite sia quelle inferite. Il risultato ottenuto viene chiamato Common Thesaurus.

   Esempio 3 Rispetto al nostro esempio, le relazioni terminologiche inferite sono:

   ⟨SCS.CS_Person BT SU.Research_Staff⟩
   ⟨SCS.CS_Person BT SU.School_Member⟩
   ⟨SU.Section RT SCS.Professor⟩
   ⟨SCS.Professor RT SU.Department⟩
   ⟨SCS.Professor RT SU.Section⟩
   ⟨SCS.Course RT SU.Room⟩
   ⟨SCS.Student RT SU.Section⟩
   ⟨SCS.CS_Person BT STP.University_Student⟩

   In Figura 2.2 sono invece riportate tutte le classi di tutte le sorgenti, così come risultano riorganizzate alla fine della fase di generazione del Thesaurus comune, facendo quindi riferimento alle classi modificate. È quindi una rappresentazione grafica delle relazioni che sussistono tra queste: le linee tratteggiate mettono in evidenza le relazioni inferite, le linee unite rappresentano invece quelle esplicite; le linee dotate di frecce rappresentano inoltre relazioni di generalizzazione, mentre quelle in cui le frecce sono assenti denotano le relazioni di aggregazione.

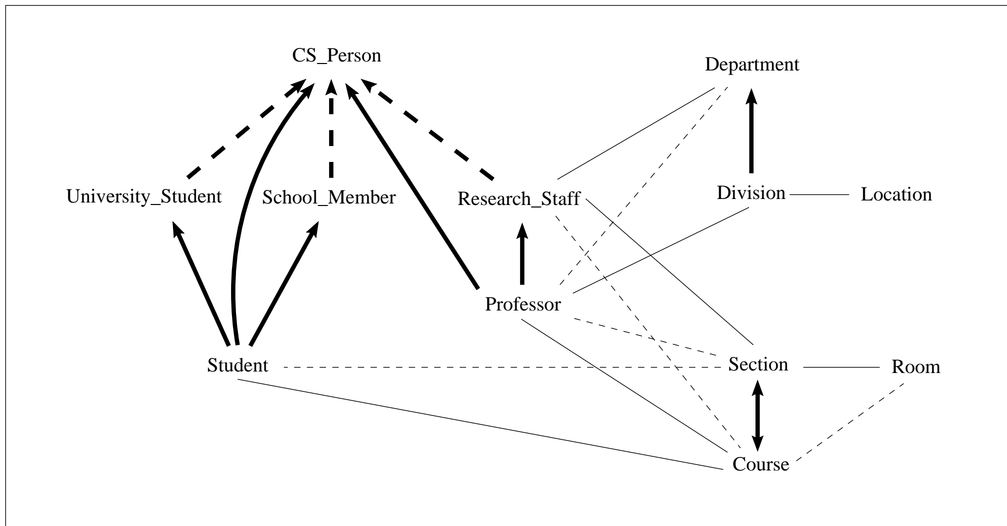

Figura 2.2: Thesaurus comune per le sorgenti $\mathbf{S_1}$, $\mathbf{S_2}$, e $\mathbf{S_3}$.

2.3 Analisi di Affinità delle classi ODL$_{I^3}$

Per realizzare l'integrazione degli schemi ODL$_{I^3}$ delle differenti sorgenti in uno schema globale, abbiamo bisogno di tecniche per l'identificazione delle classi che descrivono le stesse informazioni (o informazioni semanticamente equivalenti), e che sono localizzate all'interno di sorgenti diverse. A questo scopo, le classi ODL$_{I^3}$ sono analizzate e raffrontate attraverso il concetto di *affinità*, che ci permette di determinare il livello di *similarità* tra classi.

Questa attività è compiuta con il supporto dell'ambiente di ARTEMIS. ARTEMIS è stato costruito per l'integrazione semi-automatica di database eterogenei di tipo strutturato [44]. Nell'ambito del progetto MOMIS, le potenzialità di ARTEMIS sono state estese per permettere l'analisi di affinità di descrizioni espresse in linguaggio ODL$_{I^3}$ e gestire i dati semistrutturati.

Le classi ODL$_{I^3}$ sono analizzate e valutate attraverso un *affinity coefficient* che permette di determinare il livello di similarità tra classi contenute in sorgenti diverse. In particolare, per le classi vengono analizzate le relazioni che esistono tra i loro nomi (attraverso il *Name Affinity Coefficient*) e tra i loro attributi (per mezzo dello *Structural affinity Coefficient*), per arrivare ad un valore globale denominato *Global Affinity Coefficient*.

La valutazione dei coefficienti di affinità si basa sulle relazioni terminologiche memorizzate nel Thesaurus. A questo scopo, il Thesaurus viene organizzato in una struttura simile alle Associative Networks [45], 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. 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 $\sigma_{syn} \geq \sigma_{bt/nt} \geq \sigma_{rt}$). Nel nostro esempio, e nelle sperimentazioni precedentemente realizzate presso l'Università di Milano, si è adottato $\sigma_{syn} = 1$, $\sigma_{bt} = \sigma_{nt} = 0.8$ e $\sigma_{rt} = 0.5$. Occorre, a questo punto, osservare che la correttezza del processo di affinità dipende sia dall'affidabilità delle relazioni terminologiche, sia dai parametri (i.e., strengths, weights, thresholds) che intervengono nel calcolo dei coefficienti. Come si può facilmente intuire, il processo di affinità contiene intrinsecamente una parte di soggettività, dovuta al fatto che la conoscenza specifica e l'esperienza del progettista giocano un ruolo fondamentale nella costruzione del Common Thesaurus e nella scelta dei parametri. Per rendere la fase di affinità più obiettiva, in MOMIS abbiamo introdotto funzionalità di tipo interattivo. In particolare, la costruzione e la validazione del Common Thesaurus in ODB-Tools è un processo interattivo, ed il progettista può aggiungere terminological relationships addizionali tipiche del dominio applicativo in esame. ODB-Tools valida tutte le relazioni contenute nel Thesaurus e ne inferisce di nuove, mantenendo il Common Thesaurus consistente e corretto. Il processo di valutazione dell'affinità in ARTEMIS è pure interattivo e basato sui pesi (modificabili), per permettere al designer di modificare in modo appropriato i parametri, (ad esempio in funzione del contesto applicativo è possibile selezionare diverse modalità di valutazione della *Structural Affinity*), e validare le scelte compiute durante tutti gli step del processo. L'approccio di affinità utilizzato in ARTEMIS è stato sperimentato su insiemi diversi di schemi concettuali di database per selezionare i valori di default più appropriati, che sono quelli risultati più soddisfacenti nella maggior parte dei casi.

Per una discussione più approfondita delle sperimentazioni compiute su schemi forniti dalla Pubblica Amministrazione Italiana, si può fare riferimento a [46, 44]. Considerazioni di tipo più generali sull'utilizzo e sulla definizione delle tecniche semi-automatiche basate sui pesi per l'analisi di schemi concettuali si possono trovare in [47].

2.4 Generazione dei Cluster di classi ODL$_{I^3}$

Per l'identificazione degli insiemi di classi ODL$_{I^3}$ aventi elevata affinità negli schemi considerati sono utilizzate tecniche di clustering, attraverso le quali le classi sono automaticamente classificate in gruppi caratterizzati da differenti

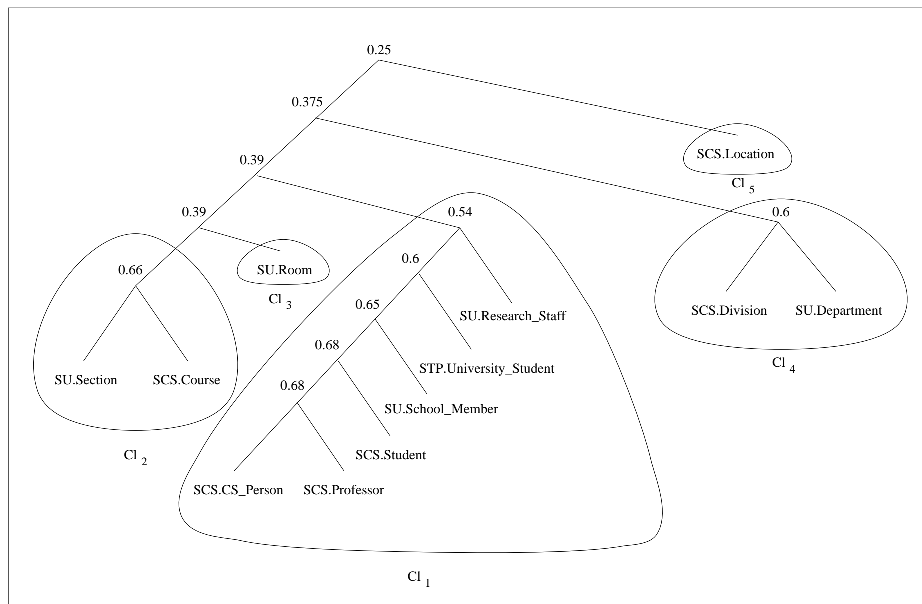

Figura 2.3: Albero di affinità

livelli di affinità, formando un albero [43].
La procedura di clustering adottata [1] opera in modo iterativo andando a creare insiemi di classi, *cluster* appunto, di dimensioni via via crescenti. Ad ogni passo viene quindi costruito un nuovo cluster unendo quelli ottenuti al passo precedente ed aventi il valore massimo di affinità. Questo processo porta alla creazione di un *albero di affinità*, caratterizzato dall'avere come foglie le classi locali, come radice un l'insieme di tutti i *cluster*, ed i cui nodi sono proprio i *cluster* individuati (con valore di affinità calante spostandosi verso la radice).

Un esempio di come le classi vengano organizzate in un *Albero di Affinità* è riportato in Figura 2.3 in cui l'algoritmo di *Clustering* è stato applicato al nostro esempio di riferimento.

Dopo aver costruito l'albero di affinitá, il problema è quello di selezionare i cluster più appropriati per la definizione delle classi nello schema globale. La procedura di selezione dei cluster, in ARTEMIS, viene mantenuta interattiva attraverso la modifica del valore di soglia. Il progettista specifica il valore della soglia T ed i cluster caratterizzati da un valore di $GA()$ superiore o uguale a T sono selezionati e proposti. Per alti valori di T si ottengono cluster piccoli e molto omogenei tra loro. Diminuendo il valore di T, i cluster ottenuti contengono più classi e sono più eterogenei. Nel tool il valore di default di T viene posto pari a 0.5. Tale valore può essere modificato dinamicamente, sulla base della specifica

applicazione in esame.

2.5 Costruzione dello Schema Globale di mediatore

In questa sezione viene presentato il processo che porta alla definizione dello *Schema Globale* del Mediatore a partire dai cluster precedentemente determinati, ovvero della visione dei dati che sarà presentata all'utente in fase di Query Processing.

La prima fase di questo processo viene realizzata automaticamente e genera, per ogni cluster, una *classe_globale*$_i$ rappresentativa di tutte le classi che fanno parte del cluster (costituisce una visione unificata di queste classi locali). Sia Cl_i un cluster determinato nella fase precedente: ad esso viene associata la *classe_globale*$_i$, alle quale corrisponderà quindi un insieme di attributi globali. La fase di determinazione degli attributi è realizzabile in modo automatico, basandosi sulle relazioni tra attributi memorizzate nel Thesaurus (vedi Sezione 2.2) e seguendo i seguenti criteri:

- ad ogni *classe_globale*$_i$ è associata l'unione degli attributi di tutte le classi appartenenti al cluster Cl_i dal quale è stata generata;
- all'interno dell'unione degli attributi sono identificati tutti gli insiemi di termini definiti sinonimi, e ne viene riportato solo uno tra essi (rimuovendo quindi tutti gli altri);
- all'interno dell'unione degli attributi sono identificati tutti gli insiemi di termini legati da relazioni di specializzazione (tra i quali erano quindi state definite relazioni di BT e NT) e vengono riorganizzati all'interno di gerarchie: per ognuna di queste gerarchie è mantenuto solamente il termine più generale (che quindi ne sta a capo e ne può essere considerato il rappresentante) mentre sono rimossi tutti gli altri.

Per esempio, per il cluster $\texttt{Cl}_1$ di Fig. 2.3 si ottiene il seguente insieme di attributi globali:

```
A = (name, rank, title, dept_code, year, takes, relation,
     email, student_code, tax_fee, section_code, faculty)
```

Oltre a questa unione "$ragionata$" degli attributi è necessaria un'ulteriore fase di raffinamento che aumenti l'espressività dello schema globale, portando alla generazione, per ogni classe globale, di una *Mapping Table* cioè di una struttura dati contenente tutte le informazioni necessarie per il passaggio dalla rappresentazione

globale agli schemi locali. Questa fase vede l'intervento interattivo del progettista che è chiamato ad esplicitare tutte quelle informazioni necessarie, da un alto, all'utente finale per poter utilizzare in modo efficace la vista globale, e, dall'altro, al *Query Manager* che deve effettuare in modo automatico la trasformazione delle interrogazioni.
Come è facile intuire l'elemento fondamentale di tutto il processo di raffinamento è la presenza di un'interfaccia grafica che mostri al designer tutte le informazioni ed alternative disponibili, assistendolo poi nella scelta della rappresentazione più adatta. Informazioni fondamentali a tale scopo sono ad esempio le strutture degli schemi locali, ma anche descrizioni relative al ruolo e significato degli attributi e dizionari dei termini usati.
Le decisioni che devono essere prese sono relative a:

1. $global_class_i$ name
   il tool propone un insieme di nomi candidati per la classe globale, utilizzando le terminological relationships definite nel Common Thesaurus per le classi contenute in $\mathtt{Cl_i}$. Basandosi su questi candidati, il designer può decidere il nome più appropriato per $global_class_i$. Nel caso in cui i nomi candidati non siano significativi, è comunque possibile assegnare nomi diversi. Nel nostro esempio possiamo decidere di assegnare il nome di `University_Person` alla classe globale definita in corrispondenza del cluster $\mathtt{Cl_1}$.

   Perciò, rispetto a $\mathtt{Cl_1}$ otteniamo la classe globale:

   University_Person = (`University_Person`,{ `name, rank, title, dept_code, year, takes, relation, email, student_code, tax_fee, section_code, faculty`})

2. *mappings* tra gli attributi globali di $global_class_i$ ed i corrispondenti attributi delle classi in $\mathtt{Cl_i}$

   In particolare, se un attributo globale è ottenuto da più di un attributo di una stessa classe in $\mathtt{Cl_i}$, il designer deve specificare il tipo di *correspondence* che deve essere definita nell'attributo globale. In MOMIS vengono fornite le seguenti opzioni:

   - *and* correspondence
     specifica che un attributo globale corrisponde alla concatenazione degli attributi della classe $c_h \in \mathtt{Cl_i}$.
     Per esempio, l'attributo globale `name` di `University_Person` corrisponde alla concatenazione degli attributi `first_name` e

last_name della classe SU.University_Worker in $\texttt{Cl}_1$. Specificando l'*and* correspondence per l'attributo name, il progettista afferma che entrambi i valori degli attributi first_name e last_name debbono essere considerati quando è coinvolta la classe SU.University_Worker.

- *union* correspondence

  specifica che l'attributo globale corrisponde ad almeno uno degli attributi specificati della classe $c_h \in \texttt{Cl}_\texttt{i}$;

3. *default values* che vengono assegnati agli attributi globali in corrispondenza della classe $c_h \in \texttt{Cl}_\texttt{i}$. Un valore di default può indicare che l'attributo non è presente tra i campi della classe c_h (in questo case si parla di valore "$null$"), oppure che assume un valore costante (indicato appunto dal valore di default).

4. *new attributes* per la classe globale. Un nuovo attributo può essere aggiunto per inserire informazioni legate alla semantica degli schemi. Ad esempio in University_Person si potrebbe inserire un attributo che riporti il nome della classe di provenienza, in modo da poter fare interrogazioni rivolte ad uno specifico schema locale.

Come scritto precedentemente, la *union* correspondance risulta utile quando un attributo globale corrisponde a due o più attributi di una classe di una sorgente, in base al valore di un attributo terzo chiamato *tag attribute*. Per esempio, supponendo di avere definito un cluster globale per la classe automobile e che in una classe di una sorgente viene memorizzato il prezzo delle auto nelle due valute di Lire Italiane e US Dollari e si abbia country come attributo tag. Il progettista può definire una *union* correspondance tra gli attributi Italian_price e US_price introducendo una rule che specifichi il mapping locale basato sul valore dell'attributo country. Utilizzando la sintassi delle mapping rule, la dichiarazione è la seguente:

```
...
 attribute integer price
    mapping rule (S.car.Italian_price union
                  S.car.US_price on Rule1),
                 ...
...
 rule Rule1 { case of S.car.country:
              ``Italy'' : S.car.Italian_price;
              ``US''    : S.car.US_price; }
```

Per quanto riguarda gli attributi di default ed aggiuntivi occorre dire il loro impiego permette di esplicitare meta-informazioni che diventano fondamentali nel garantire la necessaria espressività alla rappresentazione unificata.
Le classi `SCS.Student` e `SCS.Professor` dell'esempio di riferimento sono caratterizzate dall'avere un attributo `rank` che ne specifica il ruolo all'interno dello schema locale, distinguendo appunto tra studenti e docenti. Questo attributo non corrisponde a nessuno di quelli presenti nelle altre sorgenti, ma esprime conoscenza espressa implicitamente negli schemi, (le informazioni relative alle persone sono raccolte in classi diverse in funzione del loro ruolo). Pertanto, nella vista globale, anzichè indicarlo come campo "*null*", è decisamente più funzionale attribuirgli un valore di default che rispecchi il nome, e quindi il contenuto, della classe locale. In modo analogo l'impiego di attributi aggiuntivi può sopperire all'appiattimento che deriva dall'unificazione in un unica classe globale di classi locali distinte. Se ad esempio nello schema relativo a `University_Person` non fosse presente l'attributo `faculty` sarebbe comunque fondamentale aggiungere un campo con valore di default che indicasse il nome della facoltà di provenienza. In questo modo sarebbe, infatti, possibile interrogare la vista globale distinguendo i soggetti in funzione della facoltà a cui sono iscritti o nella quale lavorano.
La rappresentazione delle meta-informazioni espresse negli schemi locali è quindi un aspetto estremamente importante nella generazione della vista globale e il sistema deve fornire adeguate funzionalità di supporto sia per la fase di integrazione, sia per la fase di interrogazione. Durante il processo di integrazione occorre poter ridurre al minimo la perdita di conoscenza ed in fase interrogazione è necessario poterla sfruttare in modo efficace (conoscendo il significato ed i valori assunti da un attributo di default l'utente può estrarre informazioni più accurate e formulare query più selettive, quindi più efficienti)

Un esempio di definizione di un classe globale in linguaggio ODL$_{I^3}$ è mostrato in Fig. 2.4 per la classe `Hospital_Patient`. Come si può vedere da questa figura, per ciascun attributo sono definite regole di mapping, in modo da specificare sia informazioni sulla corrispondenza tra attributo globale ed il corrispondente nelle classi locali associate, sia su eventuali valori di default o di null (mancanza della corrispondenza). Per esempio, per l'attributo globale `name`, la regola di mapping specifica gli attributi che debbono essere considerati in ciascuna classe locale del cluster $\mathtt{Cl}_1$. Nel caso specifico, viene definita una *and* correspondence per la classe `SU.Research_Staff`. Una diversa mapping rule è definita per l'attributo globale `rank` per specificare il valore che deve essere ad esso associato per le istanze di `SU.Research_Staff` e `SU.School_Member`.

Come si può vedere, nella definizione di un attributo globale non viene specificato alcun dominio: questo significa che MOMIS accetta i domini definiti nelle sorgenti locali e, in risposta ad una query, sono visualizzati i dati nei formati

```
interface University_Person
(extent Research_Staffers, School_Members, CS_Person
        Professors, Students, University_Students
 key    name)
{ attribute string name
    mapping_rule (University.Research_Staff.first_name and
                 University.Research_Staff.last_name)
                 (University.School_Member.first_name and
                 University.School_Member.last_name),
                 Computer_Science.CS_Person.name
                 Computer_Science.Professor.name
                 Computer_Science.Student.name
                 Tax_Position.University_Student.name;
 attribute string rank
    mapping_rule University.Research_Staff = 'Professor',
                 University.School_Member = 'Student',
    ... }
```

Figura 2.4: Esempio di classe globale in ODL_{I^3}

specifici delle sorgenti locali.
In Figura 2.5 viene mostrato come la descrizione il linguaggio ODL_{I^3} della classe globale `Hospital_Patient` sia poi tradotta in una *Mapping Table* persistente.

University_Person	**name**	**rank**	**dept**	**faculty**	...
Research_Staff	first_name *and* last_name	'Professor'	dept_code	null	...
School_Member	first_name *and* last_name	'Student'	null	faculty	...
CS_Person	name	null	null	'Computer_Science'	...
Professor	name	rank	belongs_to	'Computer_Science'	...
Student	name	rank	null	'Computer_Science'	...
University_Student	name	'Student'	null	faculty_name	...

Workplace	**name**	**area**	**employee_nr**	**budget**	...
Department	dept_name	dept_area	null	budget	...
Division	description	sector	employee_nr	fund	...

Figura 2.5: Mapping table di University_Person e Workplace

Da quanto detto in questo paragrafo è evidente che l'insieme di *Mapping Table*, rappresentano il perno di tutto il sistema costituendo il risultato del processo di integrazione ed il punto di partenza per la fase di query processing. Uno degli obiettivi principali di questa tesi consiste appunto nella definizione di una struttura dati per la rappresentazione di queste tabelle e nello sviluppo di una libreria "*aperta*" contenente le principali funzioni di mapping. L'intenzione è

quella di sviluppare uno strumento flessibile che possa essere esteso e personalizzato dal progettista in funzione delle proprie necessità.

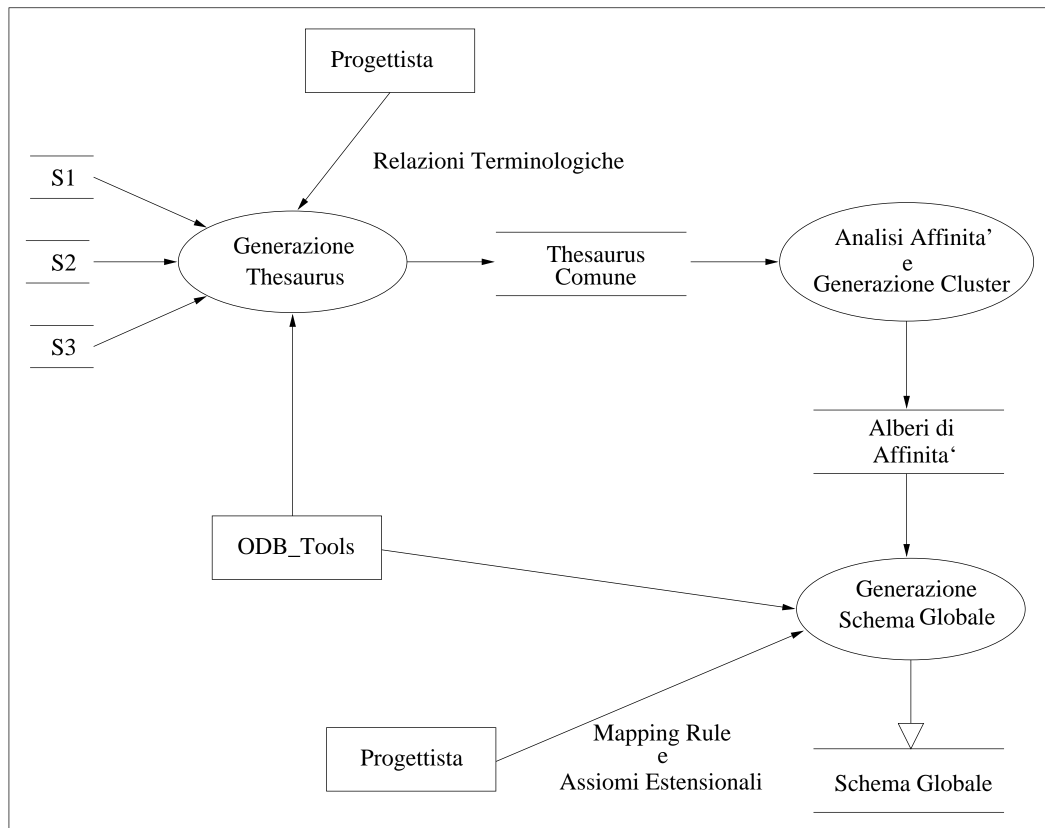

Figura 2.6: Fasi dell' Integrazione Intensionale

In Figura 2.6 sono rappresentate le fasi del processo di integrazione estensionale evidenziando l'intervento del progettista da un lato e l'impiego di ODB-Tools dall'altro.

Capitolo 3

Integrazione estensionale di schemi

La gestione delle interrogazioni in un contesto distribuito ed eterogeneo richiede la capacità di reperire il giusto insieme di dati dalle sorgenti, ma anche l'abilità nel combinarli in modo corretto. Perché un'interrogazione possa essere eseguita in un contesto eterogeneo, è necessario poter risolvere i conflitti intensionali e semantici ma occorre anche sapere come la conoscenza rappresentata nella vista globale sia effettivamente mappata sui dati presenti nelle sorgenti.
In questo capitolo verranno illustrare le problematiche connesse alla gestione della conoscenza estensionale, mostrando come questo tipo di informazioni possano essere utilizzate per arricchire lo schema globale gestito dal Mediatore..

3.1 Le relazioni estensionali

Le attività descritte nel capitolo 2 sono volte alla risoluzione dei conflitti intensionali, cioè quelle incompatibilità derivanti dall'avere porzioni di schemi sovrapposte (descrivono gli stessi aspetti del dominio applicativo), ma con strutture differenti. Ciò che è stato fatto consiste quindi nel fornire una rappresentazione unificata ed omogenea, dei medesimi concetti descritti in sorgenti differenti.

L'integrazione degli schemi non è però l'unico aspetto che occorre gestire per ottenere un'effettiva integrazione di sorgenti eterogenee, infatti, come descritto in [2, 48, 49], è necessario risolvere anche i conflitti derivanti dalle sovrapposizioni delle estensioni, cioè dalla presenza, in sorgenti diverse, di informazioni relative alla stessa entità del "*mondo reale*".
Per comprendere le problematiche connesse alla presenza di *sovrapposizioni estensionali* occorre chiarire la distinzione tra *Oggetti* di un database ed *Entità* di un certo contesto applicativo.
Studiando un determinato contesto possono essere infatti individuate entità caratterizzate da un certo insieme di comportamenti e proprietà, esse esprimono

concetti ben definiti del "$mondo\ reale$" ai quali però possono essere associate molte rappresentazioni alternative. In particolare Database indipendenti forniranno modellazioni differenti, non solo descrivendo con strutture diverse le stesse proprietà ma anche andando a cogliere, in funzione delle loro finalità ed obiettivi, aspetti differenti della stessa entità. Pertanto un'entità rappresenta un concetto astratto che prescinde da una particolare rappresentazione, mentre un oggetto è uno strumento di modellazione che, utilizzando una particolare struttura, ne cattura determinati aspetti.
È evidente, a questo punto, che sorgenti autonome possono contenere oggetti corrispondenti alla stessa entità ed ognuno di questi, in parte, replicherà proprietà già presenti in altri oggetti ma potrà anche fornire un proprio contributo descrivendone di nuovi. Pertanto l'obiettivo dell'integrazione deve essere non solo il reperimento dei singoli oggetti ma piuttosto la ricomposizione dell'entità a cui sono associati.
Perché ciò sia possibile è necessario comprendere come le informazioni provenienti dalle varie sorgenti debbano essere combinate e quindi occorre impiegare sia le relazioni tra le estensioni sia quelle sulle intensioni. Le prime permettono di individuare e ricostruire le istanze della classe "$astratta$" di entità e le seconde specificano quali sono le proprietà effettivamente note di tali istanze.

Esempio 4 Supponiamo di voler gestire un dominio applicativo in cui è presente il concetto di "$persona$" (P) ed immaginiamo di avere tre classi locali (C_1, C_2 e C_3) contenenti informazioni relative a persone. Immaginiamo poi che la classe di entità "$persona$" sia caratterizzata dalle proprietà $a_1, \ldots, a_8$ e che gli schemi delle classi locali siano parzialmente sovrapposti.

int(P) = $\{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8\}$

int(C_1) = $\{a_1, a_2, a_3, a_4\}$

int(C_2) = $\{a_3, a_4, a_5, a_6\}$

int(C_3) = $\{a_7, a_8\}$

Per arrivare ad un'integrazione corretta può non essere sufficiente fare una semplice unione delle estensioni delle classi, in quanto dati relativi alla stessa entità potrebbero essere presenti in più classi. L'unione, in questo caso, comporterebbe un errore nella rappresentazione, infatti ogni oggetto nelle classi locali verrebbe considerato come un'istanza distinta della classe globale, Figura 3.1 (a), determinando una duplicazione di informazioni (la stessa persona sarebbe presente più volte), e l'incompletezza della risposta (le persone duplicate hanno un

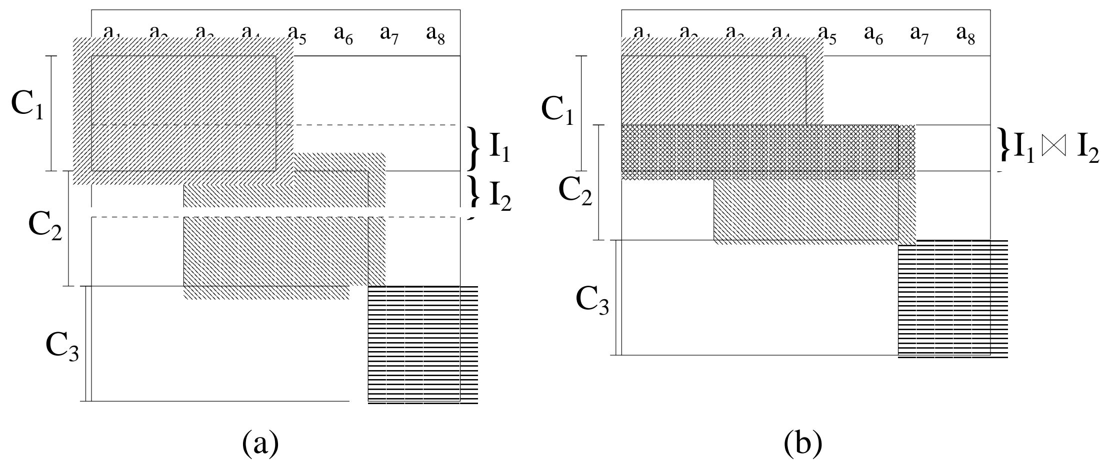

Figura 3.1: Estensione della classe di entità

insieme incompleto di proprietà).
Questi inconvenienti possono essere risolti soltanto gestendo in modo corretto le relazioni tra le estensioni. Se ad esempio le estensioni di C_1 e C_2 risultano essere sovrapposte significa che un sottoinsieme delle loro istanze, pur avendo attributi diversi ed appartenendo a sorgenti differenti, corrispondono alle stesse entità del mondo reale. In Figura 3.1 (b) viene appunto mostrato come gli oggetti presenti nei sottoinsiemi I_1 e I_2 ed appartenenti alla sovrapposizione tra C_1 e C_2, debbano essere "$fusi$" per fornire un'informazione corretta e completa[1].

Per realizzare un'effettiva integrazione delle informazioni occorre quindi risolvere i conflitti intensionali, in modo da recuperare insiemi di oggetti dalle singole sorgenti, ma è anche necessario fornire gli strumenti che consentano la "$fusione$" di tali oggetti per poter cos ì ricostruire le estensioni delle classi di entità del "$dominio\ applicativo$".

L'approccio seguito in MOMIS [2] si basa sulla teoria della *formal context analisys* che, come descritto in [50], è volta alla generazione di una gerarchia di ereditarietà in cui viene rappresentata la conoscenza disponibile, nell'insieme di schemi locali, su di un determinato aspetto della realtà. Gli elementi che caratterizzano questo approccio teorico sono:

- definizione di *assiomi estensionali*
  Tali assiomi descrivono le relazioni insiemistiche esistenti tra le estensioni

[1] Ovviamente il concetto di completezza è relativo alle informazioni contenute in uno stesso cluster

delle sorgenti, in particolare date due classi A e B sono individuabili; quattro tipi di situazioni[2]:

- *disgiunzione*: $\forall t : S_A^t \cap S_B^t = \emptyset$
- *equivalenza*: $\forall t : S_A^t = S_B^t$
- *inclusione*: $\forall t : S_A^t \subseteq S_B^t$
- *sovrapposizione*: $\forall t : S_A^t \cap S_B^t \neq \emptyset$

La presenza di un insieme completo e corretto di assiomi è un prerequisito fondamentale per il conseguimento di un'effettiva integrazione. D'altro canto la loro definizione è a carico del progettista e solo in parte può essere automatizzata pertanto diviene fondamentale disporre di strumenti che aiutino il designer nella fase di specifica.

- individuazione delle *base extension*
  Una "*base extension*" rappresenta un sottoinsieme di entità appartenenti ad uno stesso concetto del dominio applicativo. Presa quindi una classe di entità, un insieme di "*base extension*" ne rappresenta il partizionamento in modo che ogni istanza appartenga ad una ed una sola di esse e che tutte le istanze in una stessa "*base extension*" abbiano lo stesso insieme di proprietà [3].
  Le "*base extension*" sono quindi individuate dalle relazioni estensionali presenti tra classi locali e sono caratterizzate dall'avere come estensione l'insieme di entità formate dall'intersezione delle classi che la compongono e come intensione l'unione dei loro schemi.

Esempio 5 supponiamo che tra le classi `SCS.CS_Person`, `SU.University_Worker`, e `SU.Research_Staff` dell'esempio di riferimento, sussistano le relazioni estensionali mostrate in Figura 3.2.

Si ha che `SU.Research_Staff` è inclusa in `SU.University_Worker`, `SCS.CS_Person` ed `SU.University_Worker` sono parzialmente sovrapposte e la stessa condizione sussiste tra `SCS.CS_Person` e `SU.Research_Staff` .

[2] con S_A^t e S_B^t si sono indicati gli stati, all'istante t, rispettivamente delle classi A e B.

[3] al contrario di quanto accade per un normale database, una *classe di entità* sarà popolata da istanze caratterizzate dall'avere, in generale, solo un sottoinsieme delle proprietà definite nella classe stessa

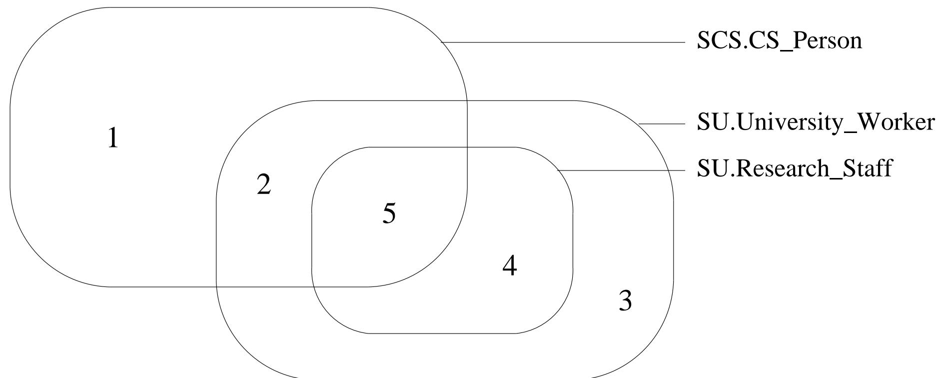

Figura 3.2:

Base Extension	1	2	3	4	5
SCS.CS_Person	1	1	0	0	1
SU.University_Worker	0	1	0	1	1
SU.Research_Staff	0	0	1	1	1

Tabella 3.1: Rappresentazione delle "*base extension*"

Le "*base extension*" presenti sono quindi cinque. In particolare possiamo dire che che le istanze in BE1 e BE3 sono caratterizzate dall'avere le sole proprietà, rispettivamente delle classi `SCS.CS_Person` e `SU.University_Worker`, mentre le altre godono delle proprietà presenti in tutte le classi di cui rappresentano l'intersezione.

- generazione del *concept lattice* o *gerarchia estensionale*
  Sfruttando la conoscenza intensionale, le "*base extension*" relative ad una stessa classe di entità vengono ora riorganizzate in un *concept lattice* che diviene il punto di partenza per ogni processo di interrogazione. L'idea è quella di associare ad ogni classe globale una gerarchia estensionale, composta da classi "*virtuali*". Queste classi sono caratterizzate dall'avere uno schema, ed un'estensione che è composta dall'insieme di base extension soddisfacenti lo schema dato.
  Questa gerarchia, fornita un'interrogazione, deve fornire tutte le informazioni necessarie alla "*costruzione*" delle entità aventi le proprietà richieste. Ciò significa poter individuare tutte le "*base extension*" aventi, almeno, l'insieme di attributi specificati nella query.

MOMIS realizza il processo di integrazione estensionale sfruttando i risul-

tati dell'integrazione intensionale e le funzionalità offerte da ODB-Tools. In particolare i passi che compongono tale processo sono:

3.1.1 Definizione degli assiomi estensionali

La prima fase del processo di integrazione estensionale deve prevedere un'accurata analisi degli schemi, in modo da individuare un insieme il più possibile completo di relazioni estensionali tra classi. Parte degli assiomi possono essere ricavati direttamente dalle definizioni stesse degli schemi (una relazione di specializzazione tra classi corrisponde ad un assioma di inclusione e lo stesso vale per un'associazione con vincolo di integrità referenziale), ma per le relazioni interschema la fonte principale rimane sicuramente il progettista.
Pur avendo una buona conoscenza del dominio applicativo e della semantica degli schemi che devono essere integrati non è semplice individuare tutte le relazioni esistenti tra classi, pertanto risulta fondamentale disporre di strumenti di supporto sia nella fase di definizione delle relazioni sia in quella di verifica delle stesse. In particolare durante la fase di definizione è importante fare ricorso a metodi che permettano di arrivare all'individuazione di un insieme corretto di assiomi mediante raffinamenti successivi, in questa direzione, ad esempio, una proposta decisamente interessante è quella illustrata in [48].
Occorre anche osservare che l'insieme di asserzioni a cui si è pervenuti può portare ad una parziale revisione dello schema globale prodotto dal modulo Global Schema Builder, infatti l'analisi estensionale fatta in MOMIS si basa su due presupposti:

- tra classi appartenenti ad uno stesso cluster e per le quali non è specificata nessuna relazione si assume che le loro estensioni siano sovrapposte.
- tra classi appartenenti a cluster diversi deve sussistere una relazione di disgiunzione estensionale.

La seconda ipotesi è stata introdotta in quanto le classi globali dovrebbero raccogliere tutte le informazioni relative ad uno stesso concetto del dominio applicativo, pertanto non possono essere presenti entità aventi proprietà partizionate su cluster distinti. Se uno o più assiomi violassero questa condizione allora sarebbe necessario o rimuoverli, perché non corretti, o modificare i cluster, ad esempio fondendoli in uno unico.

Rimane da dire che in MOMIS le relazioni estensionali vengono espresse come rule nel linguaggio ODL_{I^3}, senza bisogno di estendere ulteriormente la sintassi e senza creare ulteriori complicazioni all'utilizzatore. La ragione che consente di ricorrere alla sintassi delle rule risiede nel fatto che assiomi estensionali e rule

sono semanticamente equivalenti infatti queste ultime non sono che un modo per dire che un insieme di istanze di un certo concetto c_1, che godono di certe proprietà, appartengono ad un altro concetto c_2. Date due classi A e B gli assiomi sono quindi rappresentati nel seguente modo[4]:

1. relazione di *disgiunzione*:

   `rule RE1 forall x in (A and B) then x in bottom`

2. relazione di *inclusione*:

   `rule RE2 forall x in B then x in A`

3. relazione di *equivalenza*

   `rule RE3 forall x in A then x in B`

   `rule RE4 forall x in B then x in A`

Si osserva che le relazioni di sovrapposizione non devono essere rappresentate in modo esplicito infatti, come già detto, in MOMIS si assume che le estensioni di tutte le coppie di classi in uno stesso cluster e per le quali non sia specificata nulla, siano parzialmente sovrapposte.

Facendo riferimento al nostro schema d'esempio possiamo immaginare di definire il seguente insieme di assiomi estensionali:

```
rule RE1a forall x in SU.School_Member
            then x in STP.University_Student;

rule RE1b forall x in STP.University_Student
            then x in SU.School_Member;

rule RE2 forall x in SU.Research_Staff
           then x in SU.University_Worker;

rule RE3 forall x in SCS.Student
           then x in SU.School_Member;

rule RE4 forall x in SCS.Professor
           then x in SU.Research_Staff;

rule RE5 forall x in (SCS.Professor and SU.School_Member)
           then x in bottom;
```

[4]nella logica descrittiva **OLCD** un tipo o classe *bottom* rappresenta un "*concetto*" incongruente, cioè che non può essere in nessun caso popolato da dati o istanze

```
rule RE6 forall x in (SU.Research_Staff and STP.University_Student)
           then x in bottom;

rule RE7 forall x in (SU.Research_Staff and SCS.Student)
           then x in bottom;
```

nota : per meglio illustrare gli aspetti connessi alla gestione delle relazioni estensionali è stato modificato l'esempio di riferimento aggiungendo alla sorgente `University` la classe `University_Worker` avente il seguente schema:

```
University_Worker(first_name, last_name, dept_code, pay)
```

Questa classe è stata inserita nel cluster Cl_1.

3.1.2 Traduzione degli assiomi estensionali in proprietà intensionali

Gli assiomi relativi alle classi di uno stesso cluster sono impiegati per la costruzione della corrispondente gerarchia estensionale. Perché ciò sia possibile occorre prima integrare le informazioni a disposizione, arricchendo le descrizioni delle classi locali con le conoscenze estensionali, a tale scopo viene effettuata una trasformazione degli assiomi in relazioni di ereditarietà con il seguente approccio:

1. ogni asserzione di equivalenza fra due classi porta alla generazione di una classe con intensione corrispondente all'unione delle due intensioni, questa *classe equivalente* va a sostituire quelle iniziali.
   Ad esempio date le seguenti definizioni:

   ```
   interface School_Member              interface University_Student
   { attribute string  name;            { attribute string name;
     attribute string  faculty;           attribute integer studcode;
     attribute integer year; } ;          attribute string faculty;
                                          attribute integer tax; } ;

   rule RE1a forall x in SU.School_Member
               then x in STP.University_Student;

   rule RE1b forall x in STP.University_Student
               then x in SU.School_Member;
   ```

   si introduce la *classe equivalente* che sostituisce `STP.University_Student` e `SU.School_Member` , e la cui intensione è pari all'unione delle rispettive intensioni:

```
interface School_Member_University_Student
{ attribute string  name;
  attribute string  faculty;
  attribute integer studcode;
  attribute integer year;
  attribute integer tax; } ;
```

2. ogni asserzione di inclusione "$ridefinisce$" la classe inclusa introducendo l'ereditarietà dalla superclasse. Ad esempio:

```
interface Research_Staff
{ attribute string name
  attribute string relation;
  attribute string email;
  attribute integer deptcode;
  attribute integer sectioncode;
```

```
interface CS_Person
{ attribute string name; } ;

interface Professor : CS_Person
{ attribute string title;
  attribute Division belongsto;
  attribute string relation; } ;
```

```
rule RE4 forall x in SCS.Professor
           then x in SU.Research_Staff;
```

`SCS.Professor` è così ridefinita:

```
interface Professor : CS_Person, Research_Staff
{ attribute string title;
  attribute Division belongsto;
  attribute string relation; } ;
```

3. ogni asserzione di disgiunzione introduce un tipo *bottom* che eredita dalle classi disgiunte. Con riferimento al nostro esempio gli assiomi:

```
rule RE5 forall x in (SCS.Professor and
   School_Member_University_Student) then x in bottom;

rule RE6 forall x in (SU.Research_Staff and
   School_Member_University_Student) then x in bottom;
```

generano le nuove classi *bottom*:

```
view Bottom_P_SS : Professor , School_Member_University_Student
{} ;
view Bottom_RS_SS: Research_Staff ,
                   School_Member_University_Student
{};
```

4. ogni asserzione di sovrapposizione (sono implicite per ogni coppia di classi per cui non sia stata predicata esplicitamente nessun'altra relazione estensionale), porta alla definizione di una nuova classe *virtuale* che specializza entrambe le classi di partenza.
   Occorre osservare che oltre alle sovrapposizione tra le classi inizialmente presenti nel cluster occorrerà prendere in considerazione, e quindi trasformare, anche tutte le possibili sovrapposizioni presenti tra le nuove classi virtuali inserite dalla fase stessa di traduzione [5].

```
view Inter_CSP_UW : CS_Person , University_Worker
{} ;
```

La giustificazione per queste trasformazioni deriva dal fatto che definire una relazione tra le estensioni di due classi significa affermare l'esistenza di istanze in classi distinte e corrispondenti alla stessa entità del dominio applicativo. Tale entità può quindi essere immaginata come appartenente ad un classe virtuale avente come schema l'unione delle intensioni delle classi di partenza (in caso di disgiunzione basta adottare il formalismo illustrato per imporre che la nuova classe introdotta abbia estensione sempre vuota).

In questo modo viene creato uno schema virtuale che costituisce la base per le elaborazioni successive. È importante sottolineare che questo schema e soprattutto la gerarchia estensionale da esso ottenuta non andranno a sostituire l'insieme iniziale di classi ma si affiancheranno ad esso per arricchire la conoscenza disponibile sul cluster.

3.1.3 Verifica di congruenza e individuazione delle Base Extension

In questa fase viene impiegato il componente *Schema Validator* [21, 20] di ODB-Tools che, mediante il calcolo della subsumption, è in grado di riorganizzare lo schema virtuale precedentemente ottenuto, in una gerarchia di ereditarietà. Questa gerarchia ha un duplice scopo, infatti da un lato permette la scoperta di incongruenze nelle specifiche estensionali e dall'altro porta all'individuazione delle

[5] l'algoritmo che deve essere impiegato è quindi di tipo iterativo e terminerà solo quando non sarà più possibile aggiungere classi virtuali che non siano specializzazione di classi bottom

"*base extension*".
Gli assiomi inconsistenti sono evidenziati dalla presenza di una relazione di specializzazione tra classi e concetti bottom e devono essere risolti dal progettista il quale deve valutare come modificare le relazioni.

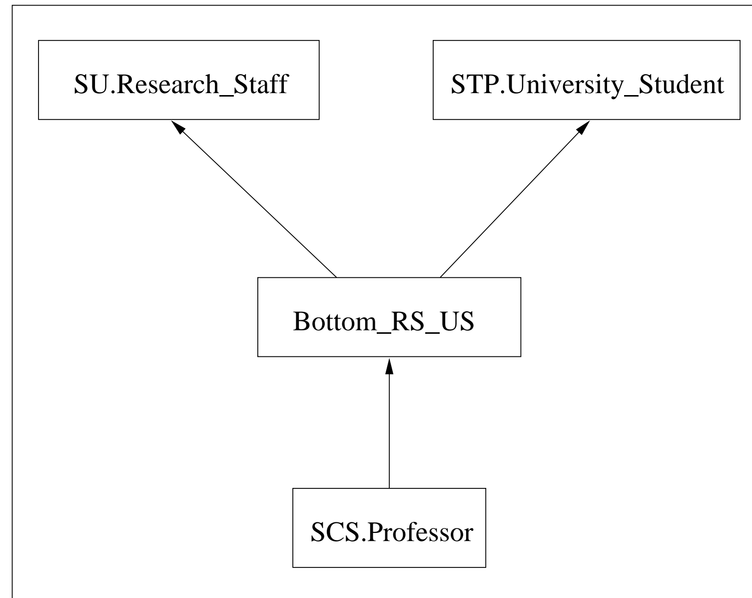

Figura 3.3: Esempio di gerarchia di ereditarietà

Esempio 6 Dalle relazioni introdotte sull'esempio di riferimento (`RE7, RE4`) si ricava che le classi `SU.Research_Staff` e `STP.University_Student` sono disgiunte e che `SCS.Professor` è inclusa in `SU.Research_Staff`. Supponendo che venga specificata anche la relazione:

```
rule RE8 forall x in Professor then x in University_Student
```

In questo caso ODB-Tools produrrebbe la tassonomia mostrata in Figura 3.3, in cui è evidente la presenza di un incongruenza, in quanto la classe `SCS.Professor` non potrebbe mai essere popolata. Per risolvere tale situazione il designer è chiamato decidere se eliminare la rule `RE8` o se togliere la condizione di disgiunzione tra `SU.Research_Staff` e `STP.University_Student`.

In Figura Figura 3.4 è riportata la gerarchia di ereditarietà relativa alla classe globale `University_Person` dell'esempio di riferimento. Questa gerarchia permette, terminata la fase di revisione e dopo aver eliminato i concetti bottom, di ottenere una descrizione completa delle base extension. Ad ogni classe

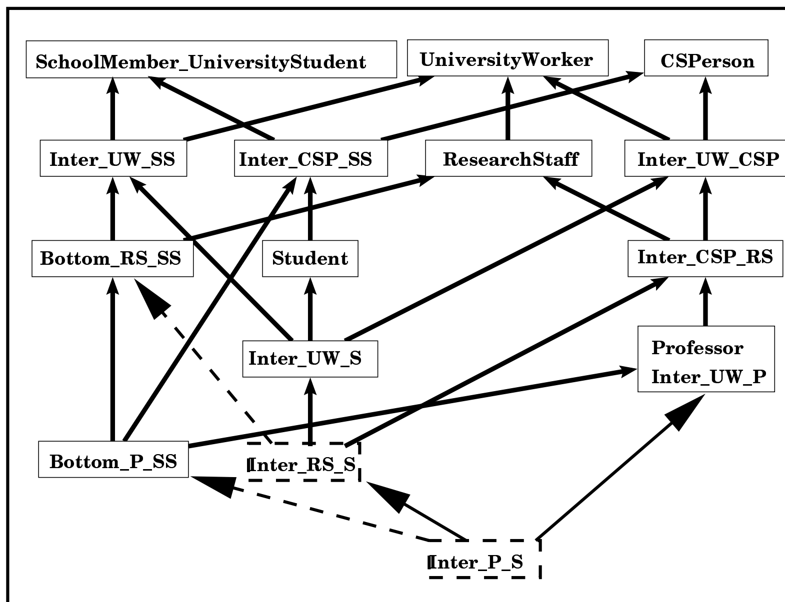

Figura 3.4: Esempio di Verifica di Congruenza

presente nello schema corrisponderà, infatti, una nuova "*base extension*" che è composta da tutte le classi locali di cui, in modo diretto o indiretto, è specializzazione. Ad esempio la classe `Inter_CSP_RS` di Figura 3.4 rappresenta la "*base extension*" generata dall'intersezione tra le classi `SU.Research_Staff` , `SU.University_Worker` e `SCS.CS_Person` . L'insieme completo delle "*base extension*" individuato dalla gerarchia precedentemente ricavata è riportato in Tabella 3.2.

L'intera struttura delle "*base extension*" può quindi essere ricavata a partire dall'insieme di classi che la compongono, in particolare:

- l'intensione è data dall'unione degli attributi **globali** descritti nelle classi locali dell'insieme;
- l'estensione è costituita dall'intersezione delle estensioni delle classi locali che la compongono;

Occorre dire che, mentre le informazioni relative all'intensione vengono solamente impiegate per la generazione della gerarchia estensionale, quelle relative alla composizione dell'estensione vanno ad arricchire la rappresentazione globale fornita dal Mediatore. Deve essere infatti costruita un struttura dati permanente, affiancata quindi alla mapping table, che indichi, per ogni base extension, come debbano essere combinate le classi che la compongono. In altri termini deve spe-

Base Extension	1	2	3	4	5	6	7	8	9	10	11
STP.University_Student		x					x	x	x	x	
SU.School_Member		x					x	x	x	x	
SCS.CS_Person	x	x		x		x		x	x		x
SCS.Professor	x										
SCS.Student		x						x			
SU.University_Worker	x	x	x		x	x	x				x
SU.Research_Staff	x				x						x

Tabella 3.2: Composizione delle "*base extension*" del Cluster University_Person

cificare quali operazioni di join devono essere eseguite, in quale ordine e quali attributi di collegamento si devono impiegare.

concetto di dominazione tra "*base extension*" :
Le "*base extension*" rappresentano insiemi disgiunti di entità, però per ricostruirle è necessario effettuare il join delle classi locali che le compongono e quindi ciò che viene prodotto durante questa operazione è in realtà un sovrainsieme della "*base extension*" cercata.

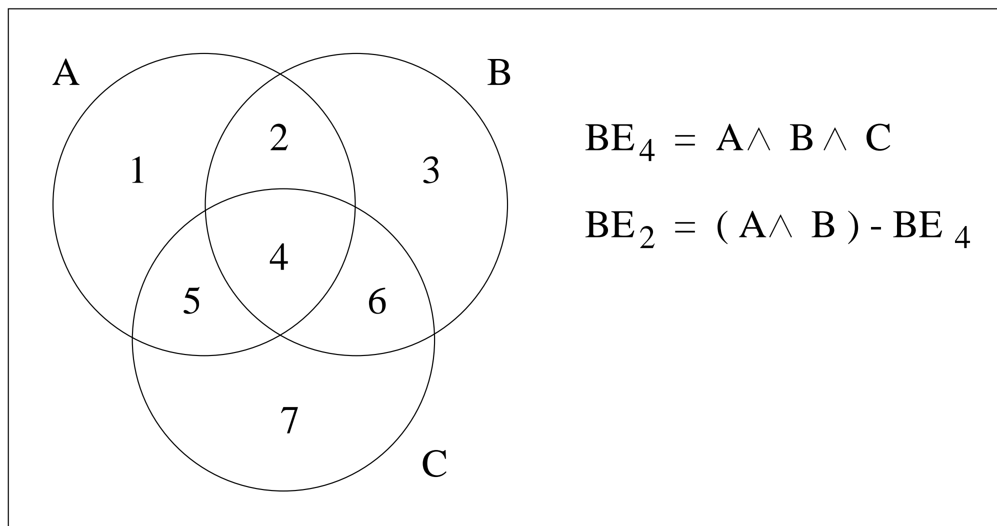

Figura 3.5: esempio di "*baseextensin*"

Osservando le relazioni estensionali tra le classi A, B e C di Figura 3.5, risulta evidente che l'intersezione, cioè il join, delle classi A e B produce oltre alle entità

della *Base Extension* 2, anche una rappresentazione parziale, non sono infatti recuperati tutti gli attributi, delle entità nella *Base Extension* 4.
Queste considerazioni portano ad introdurre un concetto di *dominazione* tra base extension:

Definizione 1 (Dominazione) *Date due* Base Extension BE_1, BE_2 *ed un insieme di attributi* $A = \{a_1, \ldots, a_n\}$, *allora* BE_1 domina BE_2 *rispetto ad A, se A è compreso nelle intensioni di entrambe e se le classi che compongono* BE_1 *sono un sottoinsieme di quelle che formano* BE_2.

Questo concetto diviene importante in fase di interrogazione infatti dovendo recuperare le entità che godono delle proprietà in A sarà sufficiente ricostruire soltanto la "*base extension*" dominante. Queste considerazioni verranno riprese nella sezione 4.2.

3.1.4 Generazione della gerarchia estensionale

Impiegando l'algoritmo illustrato in [2] ad ogni classe globale viene associata una gerarchia estensionale. Queste gerarchie sono costruite a partire dalle informazioni relative alle "*base extension*", infatti le classi che la compongono sono caratterizzate dall'avere:

- intensioni corrispondenti agli schemi delle "*base extension*" presenti nella classe globale. In questo modo si individua un insieme di intensioni (sovrapposte), che rappresentano le descrizioni di tutti i tipi di entità[6] raccolte nel cluster. Ciò significa che per ogni intensione esisterà almeno un tipo di entità, appartenente alla classe globale, caratterizzato da tutti e soli gli attributi dell'intensione stessa e che non sono previsti tipi di entità non associati ad almeno una delle intensioni;
- estensione data dall'unione di tutte le "*base extension*" che hanno almeno tutti gli attributi presenti nell'intensione della classe;

Queste gerarchie di classi "*virtuali*"devono dunque essere aggiunte, assieme alle descrizioni delle "*base extension*", allo schema globale fornito dal Mediatore, in modo da riorganizzare la conoscenza espressa nei cluster e migliorare il processo di interrogazione. Disponendo di tali informazioni, è infatti possibile determinare le classi locali coinvolte in un'interrogazione, facendo un'analisi

[6] si parla di tipi di entità e non di entità perché tutto ciò che riguarda la descrizione estensionale esprime condizioni di potenziale esistenza, quindi individuare un insieme di entità non significa che sia necessariamente popolato

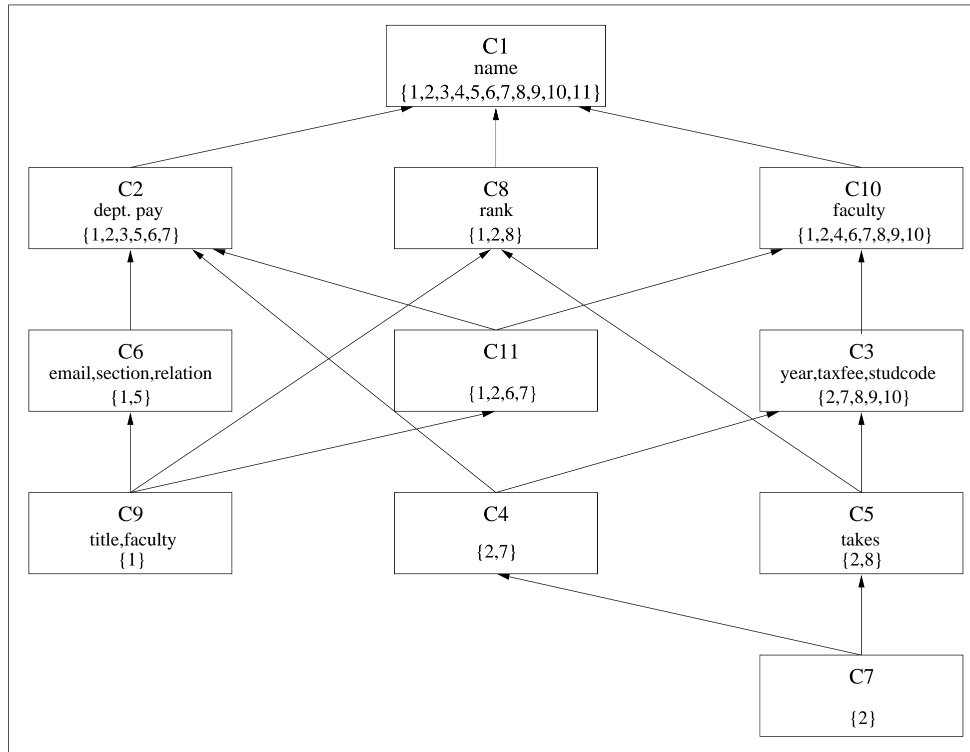

Figura 3.6: Rappresentazione della gerarchia estensionale

della query ed individuando la classe virtuale più generale che dispone di tutte le proprietà richieste.
In Figura 3.6 è riportata una rappresentazione della gerarchia estensionale associata alla classe globale `University_Person` dell'esempio di riferimento.

Perchè possano essere conseguiti i risultati descritti, non basta individuare le "*base extension*" ma occorre anche descrivere il modo per fondere le istanze delle classi che le compongono. In mancanza di queste informazioni, sarebbe infatti possibile riconoscere quando un certo insieme di istanze corrisponde, in realtà, alle stesse entità del dominio applicativo, ma non saremmo in grado di combinarle, perdendo così la possibilità di ricostruire un'informazione corretta e completa.
Questa ulteriore conoscenza non può essere generata in modo automatico, pertanto diventa chiara l'importanza del progettista, non solo nella fase di definizione degli assiomi estensionali ma durante tutto il processo di integrazione. In particolare si dovrà prevedere una sezione interattiva, che consenta al designer di individuare le operazioni necessarie alla generazione delle "*base extension*" a partire dalle classi locali che le compongono. In questa fase il progettista deve

quindi:

1. trovare, per ogni “*base extension*”, una pi˘ chiavi semantiche, cioè insiemi di attributi che individuino le entità del dominio in modo univoco[7];

2. determinare quali operazioni di join possono essere eseguite tra le classi locali di una stessa “*base extension*”. Ciò significa stabilire, per ogni coppia di classi locali, se è presente almeno una chiave semantica comune che permetta il join;

3. individuare almeno un insieme, o sequenza, di join che consenta la fusione di tutte le classi di una “*base extension*” (cioò significa che se non è possibile effettuare il join tra due classi devono comunque essere presenti altre classi di collegamento);

Queste operazioni sono dunque volte al completamento della struttura dati che descrive le “*base extension*” (vedi sezione 5.5.2).

[7]questa operazione richiede una profonda conoscenza della semantica sia del contesto applicativo, sia delle singole sorgenti

Capitolo 4

Il modulo Query Manager

Come già illustrato nella sezione 1.2.1, MOMIS gestisce una pluralità di sorgenti distribuite ed eterogenee adottando un approccio "$virtuale$", che quindi non prevede la replicazione dei dati nel Mediatore. Ciò significa che dovrà essere il Query Manager a gestire, per ogni interrogazione, la rappresentazione globale ottenuta mediante le fasi di integrazione intensionale ed estensionale al fine di materializzare presso l'utente le entità che popolano questa vista virtuale e che costituiscono la risposta cercata. Tale risposta prodotta dovrà rispondere ad esigenze di correttezza e completezza, in modo da recuperare tutte e sole le entità del dominio applicativo che godono delle proprietà richieste soddisfando le condizioni imposte, ma dovrà anche essere minima. Uno degli aspetti più innovativi del sistema MOMIS consiste appunto nell'impiego di componenti intelligenti che realizzano, sia sulla query globale sia su quelle locali, passi di ottimizzazione semantica capaci di ridurre il numero di sorgenti accedute ed il volume di dati ritornati.
Le fasi che caratterizzano il processo di gestione delle interrogazioni sono pertanto:

- *ottimizzazione semantica globale*: sfruttando le informazioni semantiche presenti a livello di schema globale, ed eventuali regole di integrità definite dal progettista, viene realizzata un'ottimizzazione semantica delle interrogazioni poste dall'utente;

- *individuazione delle sorgenti coinvolte*: analizzando la query vengono individuate le classi globali coinvolte e per ognuna di esse si determina a quali classi sorgenti si deve accedere;

- *generazione delle query locali*: sfruttando le regole di mapping tra rappresentazione globale e schemi locali viene prodotto un insieme di sottoquery direttamente eseguibile sulle sorgenti;

- *ottimizzazione semantica locale*: Una volta generate le subquery per ogni sorgente, si può pensare di sfruttare la presenza di vincoli di integrità sugli schemi delle sorgenti, unitamente alle capacità di ODB-Tools, per ridurre ulteriormente il costo di accesso ai dati. Questa opportunità può essere sfruttata purché siano rappresentate a livello di Mediatore le conoscenze semantiche relative agli schemi locali;

- *composizione della risposta*: i dati ritornati dalle sottoquery vengono ora combinati per generare la risposta all'interrogazione posta sullo schema globale

I primi passi di tale processo sono volti alla generazione di un piano di esecuzione che, per la singola query, specifichi quali dati devono essere reperiti dalle sorgenti e quali operazioni devono essere eseguite su di essi, mentre l'ultimo rappresenta la messa in esecuzione del suddetto piano per ottenere la risposta.

4.1 Ottimizzazione semantica globale

In questa fase MOMIS opera sulla query dell'utente sfruttando le tecniche di ottimizzazione semantica [51, 52] supportate dagli ODB-Tools (descritti in Sezione 1.3), al fine di ridurre il costo del piano di accesso che sarà generato.
Naturalmente, condizione necessaria per questa fase di ottimizzazione è la presenza di regole di integrità inter-sorgenti definite sullo schema globale. È pertanto il progettista che deve analizzare la semantica del dominio applicativo per individuare la presenza di regole di integrità applicabili a tutte le sorgenti. Queste regole possono essere condizioni effettivamente soddisfatte da tutte le estensioni delle classi locali (vincoli di questo tipo sono individuabili, ad esempio, in domini fortemente regolamentati come possono essere quelli relativi alla Pubblica Amministrazione), ma possono anche rappresentare restrizioni introdotte dal progettista per realizzare una particolare vista del contesto applicativo ottenuta appunto imponendo una determinata semantica alla vista integrata.

I vincoli inter-sorgenti sono definiti in termini di rule ODL$_{I^3}$ espresse sullo schema globale e vengono impiegati da ODB-Tools per riformulare la query iniziale producendone una semanticamente equivalente ma eseguibile in modo più efficiente. Ciò è ottenuto' ad esempio, eliminando predicati ridondanti (se questi contenessero join impliciti si eviterebbe la necessità di effettuare costose navigazioni), o aggiungendo nuove condizioni che possono portare ad una valutazione più efficiente della risposta (per la possibile presenza, nelle sorgenti, di indici sui predicati introdotti).
Supponiamo, per esempio, che nel nostro dominio universitario esista, a livello globale, una relazione che unisce i fondi di ricerca (rappresentati dall'attributo

globale budget) all'area (area) di appartenenza del dipartimento a cui questi fondi sono stati assegnati. Attraverso il linguaggio ODL_{I^3}, il progettista può quindi definire la seguente regola di integrità sulla classe globale Workplace:

rule R1 **for all** X **in** Workplace: (X.budget > 60000)
then X.area = 'Engineering'

Si consideri ora la seguente interrogazione: "Seleziona i nomi dei professori che lavorano in un dipartimento che ha un budget di ricerca maggiore di 60000 dollari", espressa dalla query Q7:

```
Q:  select  name
    from    University_Person
    where   rank = 'professor'
    and     works.budget > 80000
```

Il mediatore, sfruttando la regola R1, realizza l'espansione semantica della query (vedi Sezione 1.3.2) ed automaticamente ottiene la nuova query Q8:

```
Q':  select  name
     from    University_Person
     where   rank = 'professor'
     and     works.budget > 80000
     and     works.area = 'Engineering'
```

Come si vede, l'espansione semantica è stata realizzata al fine di aumentare il numero dei predicati presenti nella clausola *where*: questo processo, nonostante appesantisca la fase di query plan (ora deve essere riformulata una query più complessa rispetto all'originale), potrebbe alleggerire il lavoro di query processing delle singole sorgenti, nel caso in cui siano presenti indici secondari definiti sugli attributi aggiunti nella nuova query.
È opportuno osservare che perché questa fase di ottimizzazione possa effettivamente migliorare l'attività svolta da Query Manager è indispensabile fornire gli strumenti necessari a rendere la rappresentazione globale quanto più possibile espressiva. L'integrazione intensionale deve quindi essere seguita da una fase di raffinamento durante la quale, oltre all'individuazione delle regole di integrità globali, sia possibile arricchire lo schema evidenziando la presenza di eventuali gerarchie di aggregazione ed ereditarietà.

4.2 Individuazione delle sorgenti

Ricevendo una interrogazione posta su di una classe globale il Query Manager, innanzitutto, deve essere in grado di individuare l'insieme di classi locali contenenti i dati necessari alla generazione della risposta cercata. Questa fase deve essere svolta in modo da pervenire ad una risposta che sia il più possibile *corretta* e *completa*, dove per correttezza si intende la possibilità di reperire le sole istanze che godono delle proprietà richieste, soddisfando tutte le condizioni imposte, mentre la completezza indica la capacità di individuare tutte le sorgenti che possono contribuire alla risposta.
Mediante la mapping table si riesce solamente in parte a conseguire questi obiettivi, infatti data una query globale le informazioni di collegamento in essa rappresentate permettono di determinare, quali delle proprietà richieste nella query sono presenti in ogni classe locale appartenente al cluster, ma non consente la ricostruzione degli oggetti virtuali che rappresentano le entità descritte in termini globali. Impiegando soltanto questa struttura dati alcune delle politiche che possono essere adottate per la scelta delle sorgenti da interrogare sono ad esempio:

- eliminazione di tutte le classi locali che non prevedono nel loro schema almeno uno degli attributi globali richiesti (l'attributo corrisponde ad un valore "*null*" nella mapping table).

- eliminazione delle classi per le quali esiste almeno una proprietà non *verificabile*, cioè la query ha un predicato di selezione definito su di un attributo non presente nella sorgente.

- definizione di un livello di *credibilità* delle risposte che esprima la percentuale di condizioni che l'utente desidera siano verificate dai dati ricevuti.

Questi approcci portano alla valutazione di una risposta che può essere più o meno corretta (ciò dipende dalla politica adottata), ma che sicuramente è incompleta, infatti le sorgenti vengono considerate singolarmente perdendo la possibilità di individuare e ricostruire le entità proprie del contesto applicativo. Tali entità, come descritto in 3.1, sono caratterizzate dall'avere proprietà distribuite su più classi locali appartenenti a sorgenti differenti, pertanto la completezza della risposta può essere garantita solamente valutandone la correttezza in termini globali.
Considerando un certo insieme di oggetti locali corrispondenti alla stessa entità astratta, può infatti accadere che nessuno di essi goda di tutte le proprità richieste dall'interrogazione ma che opportunamente fusi generino un'informazione corretta.

Esempio 7 Consideriamo una semplice interrogazione che richieda i campi `title` e `relation` dalla classe `University_Person` .

```
select title, relation
from University_Person
```

Dall'analisi della mapping table si scopre che nessuna delle classi locali possiede entrambi gli attributi presenti nell'interrogazione, quindi facendo riferimento a singole istanze si potrebbe generare solamente una risposta parziale, ottenuta recuperando `title` da `SCS.Professor` e `relation` da `Research_Staff`.
Sfruttando però le informazioni estensionali si perviene ad una risposta più completa, infatti analizzando la gerarchia estensionale di Figura 3.6 si individua una classe virtuale, C_9, che presenta i campi richiesti e la cui estensione è rappresentata dalla *Base Extension* 1. Pertanto fondendo le classi `SCS.CS_Person` , `SCS.Professor` , `SU.University_Worker` e `Research_Staff`, che formano la "$base\ extension$" individuata, si è in grado di risolvere la query in modo corretto.

Per capire quali concetti del mondo reale contengono le informazioni richieste e quali sorgenti debbano essere interrogate per ricostruirli occorre allora gestire oltre alle mapping table anche la conoscenza estensionale rappresentata dalle "$base\ extension$" e dalla Gerarchia Estensionale.
L'impiego di tale conoscenza oltre a consentire il soddisfacimento dei requisiti di correttezza e completezza, introduce anche significativi elementi di ottimizzazione, permettendo l'eliminazione di inutili, duplicazioni.
Se ad esempio dovessimo accedere alle classi `SU.School_Member` e `STP.University_Student` per reperire gli attributi `name`, `rank` e `faculty` è possibile sfruttare la relazione di equivalenza tra le due classi e decidere di interrogarne soltanto una. La stessa situazione potrebbe verificarsi in presenza di relazioni di inclusione infatti se la superclasse possiede tutti gli attributi richiesti allora è del tutto inutile accedere anche alla classe inclusa. Si noti che le due classi possono appartenere a database differenti e quindi la specializzazione dell'interrogazione non potrebbe essere fatta da nessun DBMS locale.
In merito alla duplicazione occorre anche dire che se non fosse gestita in modo corretto, oltre a determinare un maggiore costo di esecuzione (un numero maggiore di dati che deve essere trasferito), porterebbe ad un errore nella risposta, infatti gli stessi dati, cioè relativi ad una stesa entità del contesto applicativo, potrebbero essere presentati all'utente più volte, figurando di fatto come associati ad entità differenti.
Sfruttando la conoscenza estensionale viene quindi completato il processo che porta all'individuazione delle sorgenti ed i passi che devono essere seguiti sono:

1. **analisi della query**
   in questa fase vengono esaminati i predicati di proiezione e selezione, al fine di stabilire quali sono gli attributi globali richiesti e, rispetto ai quali, deve essere valutata la correttezza della risposta. Ad esempio, nella query seguente:

   ```
   Q1: select name, year
       from University_Person
       where pay > 80000
   ```

   sono presenti gli attributi `name`, `year` e `pay`.

2. **individuazione delle "*base extension*"**
   analizzando gli schemi delle classi virtuali che compongono la gerarchia estensionale si scopre su quale tra esse deve essere posta la query. In particolare dovrà essere scelta come classe target la più generalizzata tra le classi che godono di tutte le proprietà richieste ed è composta dal maggior numero di "*base extension*", cioè quella che ha l'estensione massima. Rispetto alla query `Q1` e alla gerarchia mostrata in Figura 3.6, si scopre che la classe target è C_4 la cui estensione è costituita dalle *Base Extension* 2 e 7.

3. **individuazione delle classi locali e semplificazione del query plan**
   Le strutture dati associate alle "*base extension*" indicano quali classi locali devono essere combinate per materializzarne l'estensione, pertanto, a partire dalle base extension trovate al passo precedente, si è in grado di determinare le classi locali alle quali devono essere rivolte le richieste di dati. Noto l'insieme di "*base extension*" e le classi locali che le compongono è possibile, e necessario, realizzare ulteriori passi di semplificazione che, eliminando eventuali ridondanze, portino ad una riduzione del numero di subquery da generare e al conseguimento di una risposta corretta e completa. Durante questa fase occorre valutare quali "*base extension*" debbano essere effettivamente ricostruite e a quali classi locali si debba accedere. La semplificazione avviene quindi in due momenti distinti:

   - eliminazione di "*base extension*":
     Il concetto di dominazione introdotto in 3.1.3 viene a questo punto utilizzato per eliminare eventuali "*base extension*" ridondanti, infatti analizzando gli attributi che devono essere reperiti, si può scoprire che una "*base extension*" ne domina altre dunque queste devono essere scartate dal piano di accesso. Nell'esempio riportato (Tabella 3.2), si ha che la "*base extension*" 2 è dominata dalla 7 e quindi deve essere mantenuta soltanto quest'ultima. Occorre comunque osservare che

questo passo di semplificazione non elimina tutte le possibili ridondanze, infatti se vi sono due "*base extension*" che ne dominano una terza allora questa può essere scartata ma le restanti, una volta ricostruite, producono insiemi di entità parzialmente sovrapposti (la sovrapposizione è rappresentata proprio dall'estensione della "*base extension*" scartata). In una situazione di questo tipo non sarà sufficiente fare una semplice unione delle entità ma si dovrà effettuare un *outer join* tra le "*base extension*" in modo da presentare una sola volta la porzione comune.

- eliminazione di classi locali:
  Per ogni "*base extension*" rimasta è possibile ridurre il numero di classi locali alle quali occorre accedere per ottenerne la ricostruzione. Per capire quali classi locali possano essere tralasciate occorre considerare il tipo di relazioni estensionali esistenti ed anche il tipo di database. Presa una "*base extension*" può accadere infatti che due delle classi siano estensionalmente equivalenti pertanto, se nessuna delle due aggiunge informazioni rispetto all'altra (perché contiene attributi richiesti dalla query e non presenti nell'altra classe), allora è possibile effettuare una semplificazione al piano di accesso scartandone una.
  Considerando ancora l'esempio fatto ed osservando la sola "*base extension*" rimasta (7), dalla Tabella 3.2 si nota che essa è formata dalle classi locali `STP.University_Student` , `SU.School_Member` e `SU.University_Worker` . Le classi `SU.School_Member` e `STP.University_Student` sono estensionalmente equivalenti però la prima è in grado di fornire un'informazione più completa, contenendo due degli attributi richiesti (`name` e `year`). In queste condizioni è possibile decidere di scartare la classe `STP.University_Student` e dunque la query `Q1` sarebbe risolta interrogando le sole classi locali `SU.School_Member` e `SU.University_Worker`.
  Un'altra situazione in cui è possibile operare una semplificazione all'interno di una "*base extension*", si verifica quando due classi appartenenti allo stesso database sono una la specializzazione dell'altra. In queste condizioni, infatti, se la sorgente è object-oriented o se, comunque, la classe più specializzata contiene tutte le proprietà richieste, allora è corretto scartare dal piano di ricostruzione della "*base extension*" la superclasse, dal momento che non produrrebbe nessuna conoscenza aggiuntiva.

Occorre aggiungere che nel processo di semplificazione è necessario considerare, oltre agli attributi presenti nell'interrogazione, anche quelli

necessari alla ricostruzione delle "*base extension*". In particolare, questi attributi sono rilevanti al fine di decidere se scartare o no una classe, infatti quando la chiave primaria che permette la ricombinazione dei componenti della "*base extension*" non è la stessa per tutte le classi, è possibile che il join diretto tra due di esse non sia effettuabile e sia necessario utilizzarne una terza di collegamento. È evidente che per fondere le prime due classi occorre recuperare anche la terza.
Immaginiamo, ad esempio, di avere una "*base extension*" formata dalle classi A, B, C, con i seguenti schemi:

int(A) = $\{a_1, a_2, a_3, a_4, a_5\}$

int(B) = $\{b_1, b_2, b_3, b_4\}$

int(C) = $\{c_1, c_2, c_3, c_4, c_5, c_6\}$

Supponiamo ora di avere individuato due distinte chiavi semantiche K_1 e K_2, corrispondenti, rispettivamente, alle coppie di attributi: $\{a_1, a_2\}$, $\{c_1, c_2\}$ e $\{b_1, b_2\}$, $\{c_3, c_4\}$
Supponiamo ora che le coppie di attributi $\{a_1, a_2\}$ e $\{c_1, c_2\}$ rappresentino la stessa chiave semantica, consentendo il join tra le classi A e B, mentre le coppie $\{b_1, b_2\}$ e $\{c_3, c_4\}$ corrispondono alla chiave che permette di combinare le classi B e C.
A questo punti è evidente che soltanto la terza classe dispone di tutte le informazioni necessarie a ricostruire la "*base extension*", infatti per fondere le classi A e B è necessario fare prima il join tra A e C, quindi fare il join, del risultato ottenuto, con B .

Da quanto illustrato emerge che il risultato di questa fase non deve essere un semplice insieme di classi locali ma piuttosto un piano di accesso che specifichi, le sorgenti coinvolte ma anche le operazioni che il Query Manager dovrà eseguire per la generazione della risposta. Questo piano deve quindi indicare quali sono le operazioni di join da effettuare per ricomporre le "*base extension*" e come queste ultime devono essere unite, specificando se fare delle semplici unioni piuttosto che degli outer-join.

4.2.1 Gestione di query complesse

Nella descrizione del processo che porta all'individuazione delle classi da interrogare si è fatta l'ipotesi che la query fosse posta su una singola classe globale e si è

implicitamente assunto che non prevedesse operatori complessi. Questa tipologia di interrogazioni costituisce la percentuale maggiore di query significative per un Mediatore, pertanto è quella più interessante in un sistema come MOMIS.
Tuttavia volendo mantenere una completa compatibilità con il linguaggio OQL standard, è necessario estendere il tipo di situazioni gestibili prevedendo una fase di analisi che porti alla determinazione degli elementi che compongono la query stessa.
Come già spiegato, in MOMIS, è stato adottato un approccio virtuale per cui lo schema globale è formato da classi che non contengono istanze ma un insieme di strutture dati necessarie per reperire le informazioni direttamente dalle sorgenti.
Una query, in generale, conterrà richieste che non possono essere soddisfatte localmente quindi, in modo analogo a quanto avviene ad un normale DBMS; infatti, la query prima deve essere analizzata per individuare le classi globali coinvolte e, per ognuna di esse, le proprietà che devono essere recuperate. Quindi deve essere tradotta ed inoltrata alle sorgenti.
Note le richieste rivolte alle singole classi globali e la query "*globale*" di partenza è quindi possibile produrre un piano di esecuzione che specifichi la sequenza di operazioni che il Query Manager dovrà eseguire in fase di generazione della risposta. La generazione di questo tipo di piano comporta due fasi:

fase1: determinazione delle Basic Query
: una Basic Query è una query che contiene tutte le richieste rivolte alle singole classe globale e tale da poter essere soddisfatta direttamente dalle sorgenti.

fase2: ricostruzione delle query globali
: si specifica come devono essere impiegati e manipolati i dati restituiti dalle Basic Query in modo da ottenere la risposta cercata. Queste operazioni comprendono ad esempio i join espliciti tra classi globali, la costruzione di collezioni (struct, array, list, ecc..) e gli ordinamenti;

Le *Basic Query* costituiscono quindi gli elementi di base del piano di esecuzione e rappresentano di fatto una restrizione del linguaggio OQL. In particolare rispetto ad una query standard si ha che:

- è possibile navigare attraverso aggregazioni e associazioni per ricostruire oggetti complessi ma non sono ammessi join espliciti tra classi;
- non è prevista la presenza di subquery;
- non vengono restituite strutture complesse come **list**, **array** o **struct**;
- non sono presenti operatori di ordinamento (**order by**) o di conversione come:

– **listtoset**: trasforma una lista di elementi in un set, quindi privo di oggetti duplicati. Ad esempio lo statement *listtoset(list(1,2,3,2))* restituisce il set composto dagli elementi 1,2,3;

– **element**: data una collezione di oggetti, ritorna l'elemento in esso contenuto a condizione che esso sia unico;

– **flatten**: trasforma un collezione di collezione in una collezione ad un solo livello. Ad esempio lo statement *flatten(list(list(1,2),list(1,2,3))* restituisce la lista *list(1,2,1,2,3)*;

Questi operatori devono essere implementati ad un livello superiore (come funzioni invocabili nel piano di esecuzione) in quanto vanno ad agire su insiemi di informazioni già integrate.
Di seguito viene riportato un esempio di come possa essere decomposta una query globale in modo da ottenere il corrispondente piano:

```
Q1: select a.name, a.account.balance, b.employee_code
    from Person as a,
         Employee as b
    where a.name = b.name
    and a.sex = ``male''
    and b.salary > 100.000
```

Questa query globale contiene un join tra classi che viene sostituito dall'insieme di query:

```
Q1.1: select name, account.balance
      from Person
      where sex = ``male''


Q1.2: select name, employee_code
      from Employee
      where salary > 100.000


Q1.3: select x.name, x.account.balance, y.employee_code
      from Q1 as x,
           Q2 as y
      where x.name = y.name
```

In particolare Q1.1 e Q1.2 sono le basic query rivolte alle classi globali e per le quali deve essere eseguito il processo di individuazione delle classi sorgenti, ottenuto dalla fase uno, mentre Q1.3 rappresenta l'operazione di join che deve essere fatta dal Mediatore mediante l'invocazione della corrispondente funzione, ottenuta dalla fase due.

Le basic query possono essere descritte mediante una grammatica restrizione di quella relativa al linguaggio OQL standard, la cui descrizione in forma BNF è riportata in appendice E.

4.3 Generazione delle query locali

La fase di individuazione delle sorgenti, descritta in precedenza, deve essere applicata ad ogni basic query presente nel piano di esecuzione , individuando, per ognuna di esse un insieme di classi locali. Il Query Manager, disponendo delle descrizioni delle sorgenti e della struttura della basic query, pertanto ha tutte le informazioni necessarie per la generazione di quelle che possiamo definire *Local Query*, cioè le query che da inviare alle sorgenti per il reperimento dei dati.
Questa fase vede l'impiego della mapping table la quale contiene al suo interno tutte le informazioni necessarie alla risoluzione dei conflitti intensionali permettendo quindi la trasformazione della basic query, che ricordiamo è comunque posta sullo schema globale, in query eseguibili sulle sorgenti. Occorre osservare che il Query Manager deve "*riscrivere*" queste interrogazioni in funzione dei singoli schemi locali ma non deve preoccuparsi di quali linguaggi o formalismi essi impieghino. Ciò è reso possibile dall'impiego di un modello comune dei dati grazie al quale il Mediatore comunica con i Wrapper posti sulle sorgenti, saranno infatti questi moduli locali a tradurre l'interrogazione formulata in linguaggi OQL [1] nella sintassi opportuna.
Per meglio comprendere il significato della fase di trasformazione consideriamo un esempio:

```
select name, title, email, dept.name
from University_Person
where dept.employee > 200
```

analizzando la gerarchia estensionale e la tabella delle "*base extension*" si scopre che nella ricostruzione della risposta è coinvolta la sola "*base extension*" formata dalle classi locali `SCS.CS_Person` , `SCS.Professor` e

[1] ovviamente si intende la forma ristretta descritta precedentemente

SU.University_Worker e SU.Research_Staff . Semplificando ulteriormente si trova che basta generare soltanto due query locali[2]:

```
Q4: select a.first_name, a.last_name, a.email, b.dept_name
    from School_Member as a
         Department as b
    where a.dept_code = b.dept_code


Q5: select a.name, a.title
    from Professor as a
    where a.employee_nr > 200
```

È importante osservare che la fase di trasformazione delle query può portare ad ulteriori semplificazioni, infatti utilizzando gli attributi di default presenti nella mapping table e valutando i predicati di selezione delle interrogazioni si è in grado di ridurre il numero di query da inviare alle sorgenti. Se ad esempio fosse formulata una richiesta relativa a tutti i nomi delle persone della facoltà di "$Biology$" allora prima di iniziare la trasformazione in query locali sarebbe possibile eliminare dal piano tutte le "$base\ extension$" che coinvolgono le classi SCS.CS_Person , SCS.Professor e CSPS per le quali l'attributo faculty è settato al valore di default "CS", cioè "$Computer_Science$".
Un altro miglioramento nel piano di accesso è ottenibile quando deve essere ricostruita una sola "$base\ extension$" e tutte le classi locali sono nella stessa sorgente. In queste condizioni infatti è opportuno raggruppare le query locali in una unica ed effettuare il join localmente. Riprendendo l'esempio impiegato nella sezione precedente si nota che la query globale:

```
select name, year
from University_Person
where pay > 80000
```

porterebbe alla generazione delle due query locali:

```
Q6: select a.first_name, a.last_name
    from University_Worker as a
    where a.pay > 80000
```

[2]Nell'esempio si è assunto che la chiave semantica per ricostruire la "$base\ extension$" fosse costituita dall'insieme di attributi che formano il nome

```
Q7: select b.first_name, b.last_name, b.year
    from School_Member as b
```

Entrambe le interrogazioni sono rivolte alla sorgente `University`, pertanto potrebbero essere convenientemente riunite nell'unica query:

```
Q8:  select a.first_name, a.last_name, b.year
     from University_Worker as a
          School_Member as b
     where a.first_name = b.first_name
     and a.last_name = b.last_name
     and a.pay > 80000
```

4.4 Ottimizzazione semantica locale

La presenza di regole di integrità definite sulle singole sorgenti può essere sfruttata, mediante l'impiego di ODB-Tools, per effettuare un'ulteriore passo di ottimizzazione semantica volta a produrre query locali meno costose. L'ottimizzazione semantica favorisce l'esecuzione della query in un tradizionale database perché:

- aumenta la possibilità di utilizzare degli indici:

  aggiungendo un predicato implicato da una rule si può introdurre nella query un attributo che potrebbe essere indicizzato;

- consente di eliminare o modificare dei join impliciti:

  una rule consente di riconoscere se due condizioni sono ridondanti: in questo caso, se quella implicata comporta l' esecuzione di un join, viene eliminata;

- permette di evitare o ridurre l' accesso a dati inutili:

  questo caso è generato da una query che possa essere trasformata in una equivalente su di una sottoclasse:

- può determinare l' accesso a dati senza necessità di valutazione di predicati:

  questa possibilità è realizzata qualora si riconosca che una query sussuma una intera classe dello schema;

Ad esempio supponendo di disporre della rule:

```
reule RTP1: forall x in University_Student : x.faculty = ``Economisc''
            then x.tax_fee > 12000
```

porta all'espansione della query QTP1 in QTP2:

```
QTP1: select name
      from University_Student
      where faculty = ``Economics''
      and year = ``1971''


QTP2: select name
      from University_Student
      where faculty = ``Economics''
      and year = ``1971''
      and tax_fee > 12000
```

L'aggiunta di questo predicato può essere convenientemente impiegato dalla sorgente purché sia presente un indice sull'attributo `tax_fee`. L'espansione ottenuta porta ad un aumento del costo di analisi dell'interrogazione, ma risultati sperimentali [53] hanno dimostrato che il costo complessivo di esecuzione della query ottimizzata decresce rapidamente all'aumentare del numero di istanze nel database e, mediamente, all'aumentare del numero di query fatte.

Rimane da dire che si è scelto di collocare questa fase di ottimizzazione a livello di mediatore poiché è improbabile che tutte le sorgenti siano in grado di realizzarla ed è sembrato troppo pesante includere queste funzionalità nei wrapper.

4.5 Composizione della risposta

Le attività svolte dal Query Manager sino a questo punto sono volte alla preparazione della query all'esecuzione, cioè alla definizione del piano che permette la materializzazione delle informazioni presso il Mediatore . L'ultimo passo deve quindi seguire le indicazioni presenti in questo piano per accedere alle sorgenti e presentare all'utente una risposta integrata.
In questa fase, per prima cosa, il Query Manager dovrà inviare alle sorgenti le query locali presenti nei piani di associati ad ogni classe globale. Queste query restituiranno collezioni di dati che devono essere fuse in modo da ricostruire le

"*base extension*" e quindi, tramite queste, l'intera vista sulla singola classe globale. Le operazioni di unione, join e outer-join che devono essere effettuate ed il loro ordine sono indicate nei piani di accesso presenti nell'insieme di informazioni relative alle basic query.
Tutto questo corrisponde alla valutazione delle basic query presenti nel piano di esecuzione , pertanto ora il Query Manager dispone della porzione di schema globale, o meglio della porzione della sua estensione, necessaria a produrre la risposta cercata. Ciò che deve essere fatto consiste infatti nell'applicazione su tale insieme di dati delle elaborazioni indicate nel piano di esecuzione stesso, cioè devono essere eseguiti tutti i join tra classi globali ed aggregazioni che non potevano essere effettuate localmente e che sono necessarie per arrivare ad ottenere la risposta corrispondente alla query globale inizialmente formulata.

Capitolo 5

Progetto e Realizzazione del Query Manager

In questa tesi sono state analizzate le fasi e le informazioni necessarie per svolgere l'attività di query processing in un sistema basato su mediatore. Tale analisi ha permesso l'individuazione del percorso che deve seguire una generica query per essere risolta, portando alla progettazione del componente Query Manager del sistema MOMIS.
La progettazione del Query Manager è stata eseguita con l'obiettivo di realizzare un modulo software componibile ed estendibile, in modo da poter affrontare un sottoinsieme delle problematiche individuate e consentire una facile realizzazione di estensioni future. A tale scopo è stata effettuata una modellazione di tipo object-oriented, adottando poi il linguaggio Java nella fase implementativa.

Nella prima parte di questo capitolo vengono illustrate le problematiche affrontate e le soluzioni adottate, mentre nella seconda sono descritti i quattro package[1]che raccolgono le le classi Java implementate, e che sono:

1. *package queryman*
   contiene le classi che descrivono gli elementi del piano di esecuzione delle query globali. Queste classi sono dunque necessarie alla rappresentazione ed esecuzione dell'interrogazione.

2. *package globalschema*
   raccoglie le classi che descrivono lo schema globale, in particolare è stata sviluppata la struttura dati che rappresenta la mapping table e consente la trasformazione delle interrogazioni.

[1]i package permettono di organizzare le classi in modo da raggruppare tutte le funzionalità di un certo sotto-sistema, sono quindi simili alle librerie C e C++

3. *package oql*
   le classi di questo package costituiscono gli elementi della struttura dati usata per rappresentare la query OQL. Questa struttura è dunque il supporto necessario a tutte le elaborazioni che devono essere svolte sulle query.

4. *package utility*
   le classi in questo modulo svolgono funzionalità di utilità generale nell'ambito del Query Manager.

5.1 L'ambiente di sviluppo

Il Query Manager di MOMIS è stato sviluppato utilizzando il linguaggio di programmazione Java, abbandonando, pertanto, l'ambiente "*C*" impiegato nelle fasi iniziali del progetto MOMIS e per la realizzazione del modulo ODB-Tools. La decisione di cambiare il linguaggio di programmazione ha comportato un maggiore sforzo nella fase iniziale in quanto, da un lato, è stato necessario acquisire nuove conoscenze e, dall'altro, si è dovuto rinunciare alla possibilità di utilizzare codice e strutture dati realizzati in altri moduli già implementati e testati. Tuttavia, i vantaggi derivanti dall'adozione di un linguaggio object-oriented e Java in particolare hanno ampiamente giustificato la scelta fatta.
Le motivazioni che hanno portato ad adottare Java sono state principalmente le seguenti:

- il passaggio da linguaggi C e C++ a Java non è eccessivamente faticoso dal momento che la sintassi, almeno per gli aspetti principali, è sostanzialmente la stessa;

- introduce, rispetto C e C++, un'efficiente gestione degli errori basata su eccezioni che segnalano il problema e offrono la possibilità al processo in esecuzione di gestirlo in modo opportuno;

- è Java stesso a farsi carico della gestione della memoria dinamica[2], della sincronizzazione dei processi, delle caratteristiche della piattaforma hardware, consentendo lo sviluppatore di concentrarsi sulla fase di progettazione e modellazione dell'applicazione;

- è un linguaggio completamente orientato agli oggetti per cui un'applicazione è costituita da una collezione di classi ed istanze. Tutti i dati e comportamenti devono essere parte di una classe, favorendo, dunque, la modularità e componibilità del software sviluppato;

[2]la garbage collection viene effettuata dalla piattaforma Java evitando al progettista software di dover preoccuparsi della gestione dell'allocazione e de-allocazione di aree di memoria

- mediante il modulo *Javadoc*, è possibile creare una documentazione completa ed estremamente efficiente. Essa viene presentata mediante documenti *HTML* con collegamenti ipertestuali, capaci, quindi, di percorrere in modo naturale le gerarchie di aggregazione e specializzazione che raccolgono il software realizzato;

- la piattaforma Java mette a disposizione pacchetti per la gestione della grafica, i quali consentono lo sviluppo di interfacce estremamente efficaci e funzionali. Grazie a tali strumenti viene notevolmente ridotto lo sforzo necessario per la creazione di ambienti di supporto sia al progettista del Mediatore sia all'utente finale;

MOMIS è un progetto di ricerca a lungo termine pertanto caratterizzato dal ricevere il contributo di numerose persone, tra ricercatori e tesisti, a dall'essere soggetto a frequenti estensioni e revisioni. In un contesto di questo tipo è sembrato fondamentale adottare un ambiente di sviluppo tale da offrire flessibilità e componibilità e, al contempo, in grado di stimolare la produzione di una documentazione dettagliata e di pratico impiego.
In base a tali considerazioni Java è sembrata la scelta più opportuna, infatti il paradigma ad oggetti ed il concetto di ereditarietà permettono il trasferimento delle funzioni direttamente sugli oggetti in modo da attribuire loro un determinato comportamento. Si è così in grado di estendere il sistema semplicemente aggiungendo nuovi elementi caratterizzati da un loro specifico comportamento. Ovviamente, per ottenere questo livello di modularità, occorre effettuare un processo di generalizzazione che porti all'individuazione di un'interfaccia comune a tutti gli oggetti coinvolti nello stesso tipo di operazione, solo in questo modo, infatti, possono essere impiegati i risultati forniti da un'interfaccia senza doversi preoccupare di chi li ha generati.
Il componente Javadoc di Java permette poi la produzione di una documentazione estremamente efficace che può essere consultata rapidamente per individuare e capire le classi ed interfacce sviluppate e dunque ne semplifica sia l'impiego sia eventuali estensioni.

5.2 Query Manager

Le fasi di integrazione intensionale ed estensionale, sono volte alla generazione dello schema globale che dovrà poi essere impiegato dal Query Manager nella fase di gestione delle interrogazioni.

In Figura 5.1 viene appunto descritto, con uno schema funzionale di alto livello, l'insieme di attività che caratterizzano il sistema MOMIS. In particolare

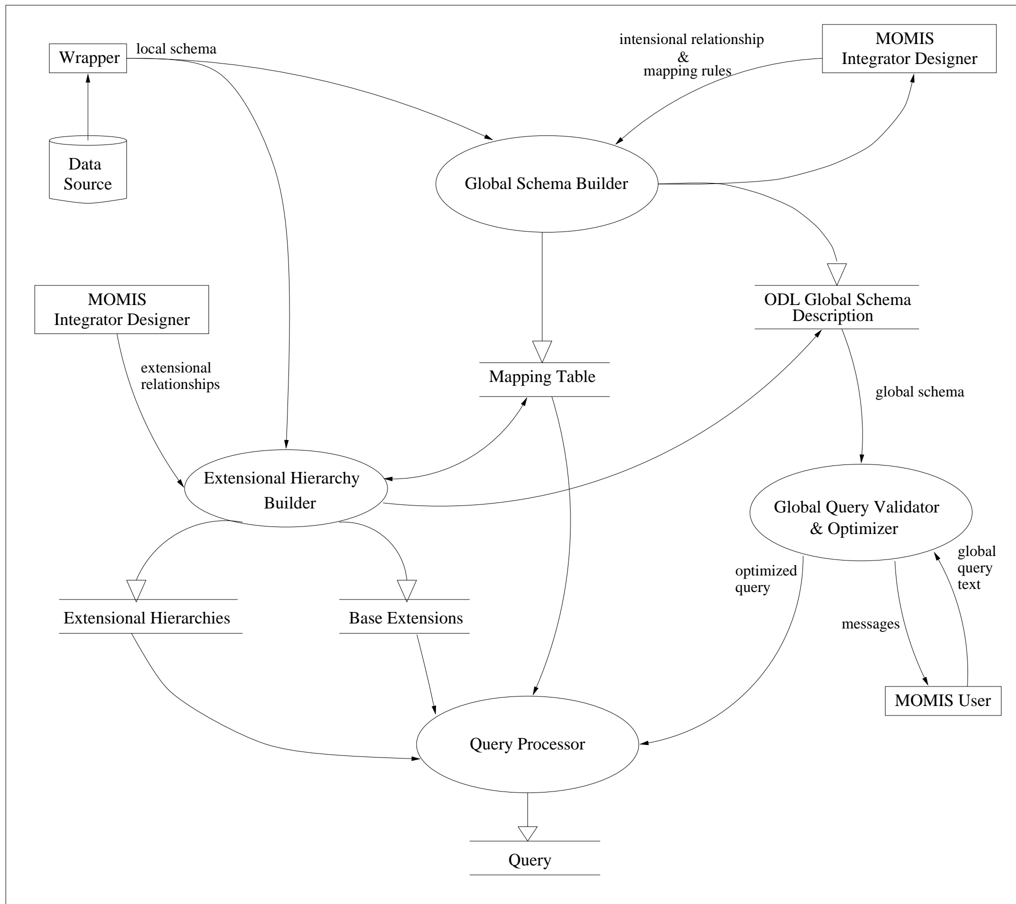

Figura 5.1: Schema funzionale del Mediatore MOMIS

durante la fase di integrazione intensionale il modulo **Global Schema Builder** riceve gli schemi delle sorgenti (espressi in linguaggio ODL$_{I^3}$), ed apre una sezione interattiva che, mediante l'intervento dell'utente e l'impiego di ODB-Tools, porta alla generazione della **Mapping Table** [54]. Un altro prodotto di questa fase è costituito dalla rappresentazione, sempre in sintassi ODL$_{I^3}$, dello schema globale integrato, questo schema potrà essere usato dall'utente come supporto durante la formulazione delle interrogazioni e, comunque, rappresenta il punto di partenza per la fase di ottimizzazione globale effettuata mediante il modulo **Query Optimizer** di ODB-Tools.
Il modulo **Extensional Hierarchy Builder** ha, invece, come obiettivo la creazione della **Extensional Hierarchy** e delle **Base Extension**. Questa fase viene svolta interagendo con l'utente che è chiamato ad individuare le relazioni intra,

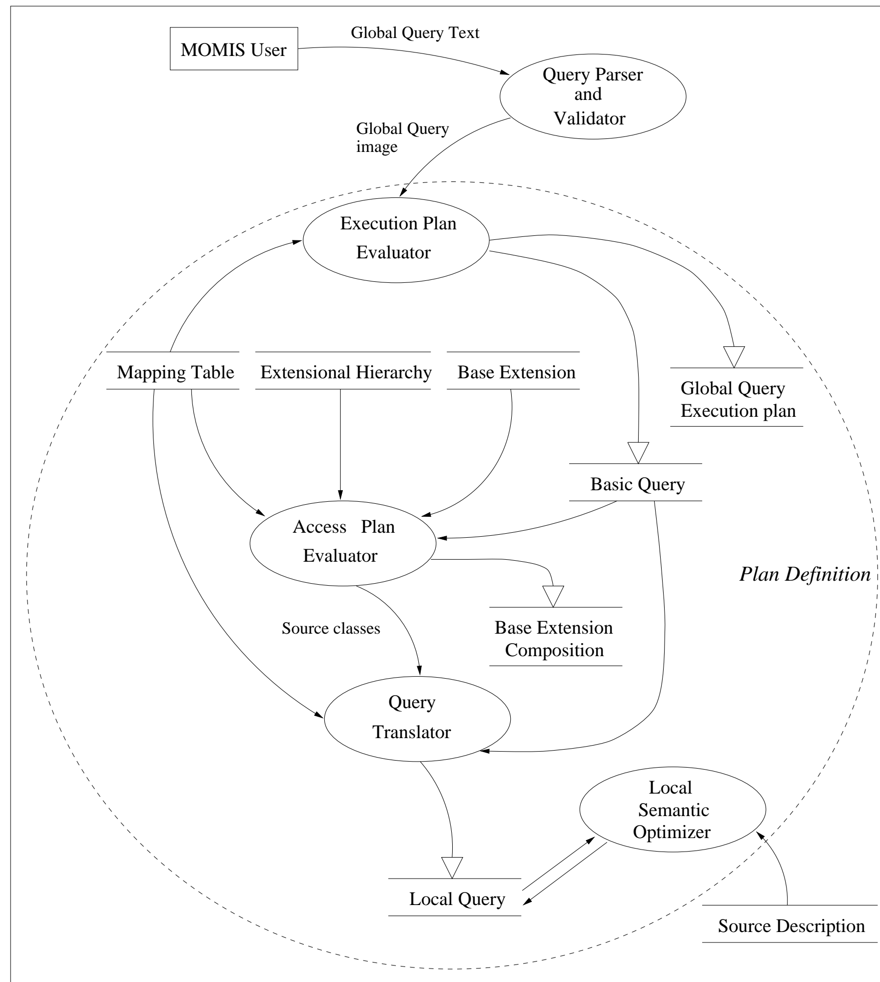

Figura 5.2: Definizione del piano di esecuzione di una Query

e soprattutto, inter-schema[3]. Come si può osservare, è prevista anche una fase di revisione dello schema globale, in quanto, durante la generazione della gerarchia estensionale, può essere necessario modificare la composizione dei cluster e quindi devono essere riviste le mapping table come anche la rappresentazione dello schema globale.

L'attività di query processing deve comunque utilizzare le strutture dati prodotte nella fase di integrazione, per risolvere ed eseguire le interrogazioni poste dall'utente. Le funzionalità di base che devono essere offerte sono due:

[3]questa attività si basa quindi, oltre che sulla descrizione delle sorgenti anche sul risultato del processo di clusterizzazione infatti deve essere noto il partizionamento delle classi locali nella mapping table.

Fase1: *Definizione del Query Plan*

Prima di porre la query in esecuzione occorre "$risolverla$", cioè individuare il piano ad essa associato. Come descritto nel capitolo 3, l'interrogazione posta sullo schema globale, o *Global Query*, non può essere inviata ai Wrapper e quindi agli schemi locali così come è, infatti:

- è posta su una vista integrata quindi, in generale, conterrà elementi e richiederà dati non presenti in tutte le sorgenti;
- è espressa in termini globali che possono essere differenti sia nella sintassi che nella semantica rispetto a quelli usati localmente;

Il primo punto richiede, come illustrato in figura 5.2, la definizione di un piano di esecuzione che, a partire dalla query globale e dallo schema globale, porti all'individuazione delle informazioni che possono essere recuperate dalle singole sorgenti. A questo punto è possibile attivare il processo di definizione del piano di accesso che, impiegando le mapping table e la conoscenza estensionale, porta all'individuazione delle classi sorgenti che devono essere effettivamente interrogate. Rimane ora da risolvere anche il secondo aspetto elencato ed effettuare la trasformazione delle interrogazioni locali, in modo che esse possano essere eseguite sulle singole sorgenti.

Fase2: *Query Execution*

La generazione della risposta avviene grazie all'applicazione del piano associato alla query e generato durante la fase precedente. Tale piano viene applicato a ritroso, pertanto i Wrapper eseguono le query locali per recuperare i dati direttamente dalle sorgenti. Queste informazioni vengono ora combinate, in base alle indicazioni contenute nella struttura dati *Basic Query Composition*, al fine di ottenere le informazioni integrate che costituiscono il contenuto delle singole classi globali. Giunti a questo punto sono state risolte le Basic Query ed i loro risultati devono essere ulteriormente elaborati al fine di ottenere la risposta complessiva. Le operazioni che devono essere effettuate sono dunque descritte nel piano di esecuzione associato alla query globale.

Le funzionalità descritte sono volte esclusivamente alla gestione ed esecuzione delle interrogazioni, occorre però dire che un sistema di questo tipo deve anche fornire strumenti di supporto per la definizione stessa delle query. Nelle future estensioni del sistema è opportuno progettare e sviluppare tool grafici che aiutino l'utente, anche inesperto, nella fase di creazione della query, consentendo un accesso efficace alle informazioni gestite dal sistema.

In questa tesi quanto è stato fatto consiste sostanzialmente nel:

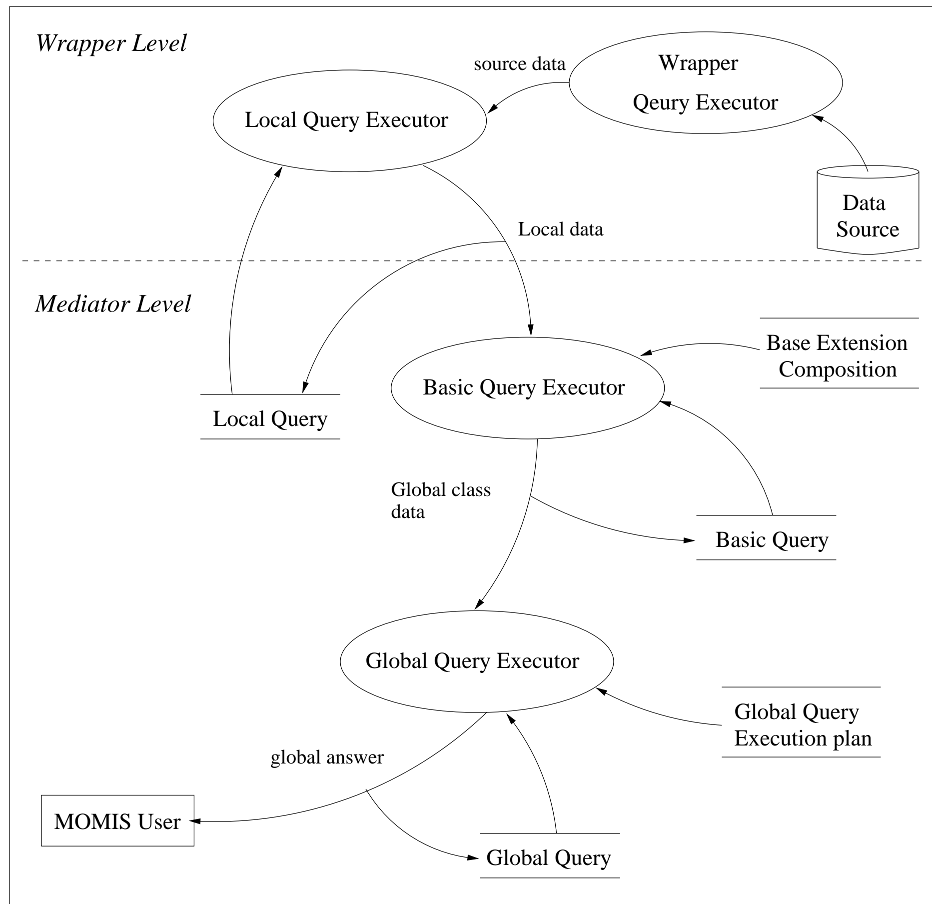

Figura 5.3: Esecuzione del piano

- progettare il modulo Query Manager, modellando le classi necessarie al suo funzionamento ;

- implementare i moduli "$Query\ Parser\ and\ Validator$" e "$Query\ Translator$", della Fase 1;

Pertanto si è delineato il percorso che deve seguire una generica query, implementando poi quelle attività che consentono la gestione di un caso particolare ma significativo , cioè delle interrogazioni globali che si presentano nella forma di basic query.
Prima di procedere con la descrizione del progetto realizzato è opportuno mostrare gli approcci seguiti nella realizzazione delle due funzionalità indicate.

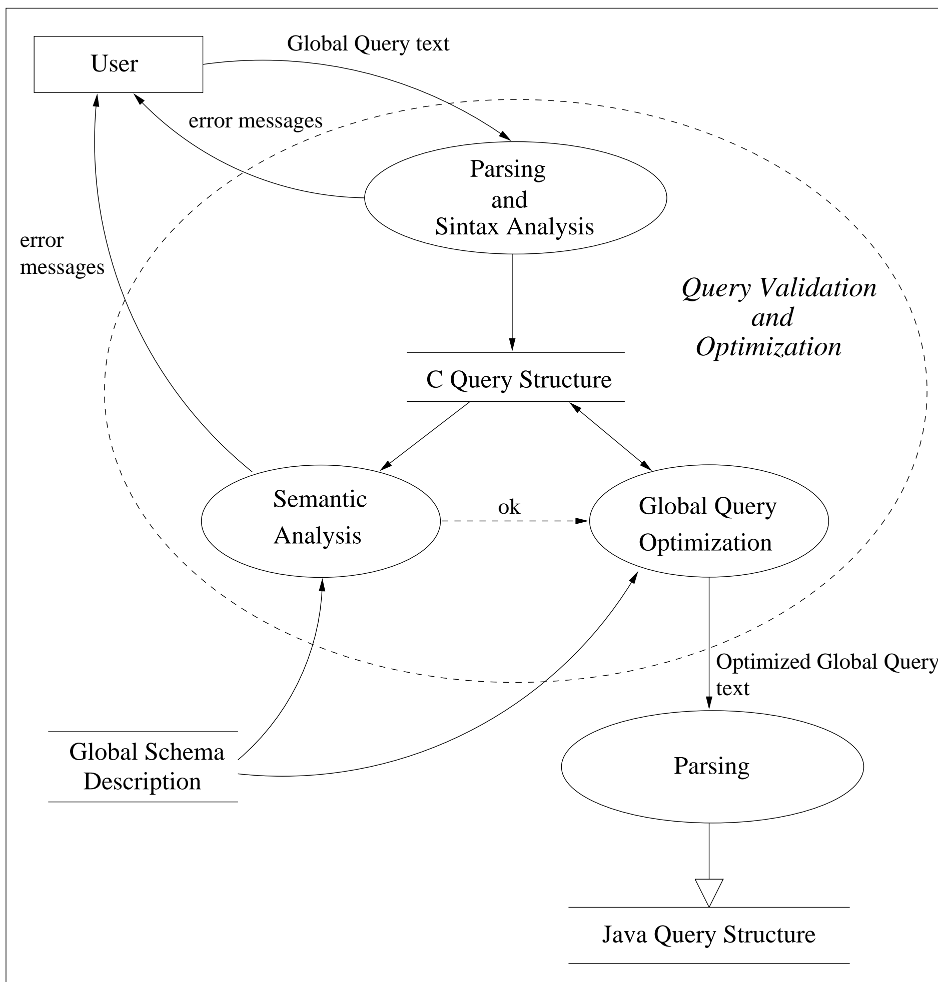

Figura 5.4: Schema di acquisizione ed ottimizzazione della query

5.2.1 Query Parser and Validator

La query, per poter essere elaborata, deve essere innanzitutto acquisita, validata ed ottimizzata.
Acquisire una query significa costruire in memoria centrale una struttura dati che ne rappresenti il contenuto. Questa immagine, che diviene la base per ogni altra elaborazione, è ottenuta mediante un modulo di parsing, cioè di riconoscimento grammaticale, il cui compito consiste nel verificare la correttezza sintattica di un'espressione, rispetto ad una determinata grammatica, producendo, eventualmente, un'immagine dell'espressione stessa in memoria centrale.
La struttura generata viene impiegata innanzitutto per effettuare il controllo semantico, nel quale si verifica, ad esempio, l'impiego delle espressioni (gli ope-

randi devono essere di un determinato tipo), e si accerta la correttezza della query rispetto allo schema (le classi e gli attributi indicati devono essere effettivamente esistenti). Terminata la validazione è possibile eseguire la fase di ottimizzazione semantica che, in modo automatico, porterà a modificare l'interrogazione e quindi la sua immagine in memoria.

In Figura 5.4 viene illustrato cime queste attività vengono svolte in MOMIS. Rispetto a quanto detto in precedenza si può notare un'inversione delle fasi di ottimizzazione e di parsing, ciò è conseguenza della necessità di sfruttare il software esistente unita alla decisione di adottare un nuovo linguaggio di programmazione. Per eseguire la validazione sintattica e semantica ed effettuare l'ottimizzazione della query, viene infatti impiegato il "$QueryOptimizer$" di ODB-Tools[4], il quale esegue una fase di parsing volta alla creazione della struttura dati "C" necessaria ad effettuare le elaborazioni richieste. A questo punto si dispone di una struttura che rappresenta la query ottimizzata ma che non può essere sfruttata dai moduli Java che invece implementano il Query Manager.
Tale struttura, pertanto, viene usata al fine di produrre un file di testo contenente l'interrogazione OQL (questo del resto è l'output che normalmente fornisce l'ottimizzatore semantico), che analizzato una seconda volta permette la creazione dell'immagine Java della query. Questa nuova fase di parsing viene eseguita dalla classe **OqlAnalizer** la quale impiega la gerarchia di classi raccolta nel package **oql** (sezione 5.3), per rappresentare la query. Così facendo ogni campo dell'interrogazione viene memorizzato mediante un corrispondente oggetto Java, specializzazione della classe **Oql_Query**.

Esempio 8 Considerando la query:

```
Q: select name, dept,budget
   from University_Person as up
   where pay > 50000
```

I campi che seguono la parola chiave *select* individuano gli attributi di proiezione. In particolare in Q si ha:

- un attributo semplice, `name`, al quale viene associato un oggetto di tipo **Oql_Identifier**;
- un attributo come `dept.budget`, che corrisponde ad un oggetto di tipo **Oql_PathExpr**;

[4]ODB-Tools è stato sviluppato in linguaggio C

La clausola *from* individua la classe, globale, target della query e l'eventuale iteratore[5](`up`). Le classi presenti in questo punto della query vengono memorizzate in oggetti di tipo **FromElement**.

La parola chiave *where* individua invece i predicati di selezione, ad esempio `pay > 50000` è un espressione relazionale rappresentato mediante la classe **Oql_Comparison**.

5.2.2 Query Translator

In questa fase, vengono esaminate le basic query presenti nel piano di esecuzione al fine di generare le interrogazioni che devono essere rivolte alle sorgenti.
L'adozione di un linguaggio comune di interrogazione tra Mediatore e Wrapper fa sì che il processo di traduzione venga svolto in modo indipendente dai linguaggi impiegati localmente, pertanto ciò che deve essere fatto consiste nel trasformare gli operatori presenti nella query in funzione dei costrutti usati localmente. Come illustrato nella sezione 2.5 le corrispondenze tra attributi globali e schemi locali sono riportati nella mapping table, ed alcune delle situazioni più significative che possono presentarsi sono:

- un attributo globale non ha una corrispondente proprietà locale o corrisponde ad un valore costante, in questo caso assegnato da un progettista;
- corrispondenza diretta tra un attributo globale ed uno locale;
- un attributo globale corrisponde a d un insieme di attributi locali;
- un attributo globale corrisponde localmente ad una foreign key;

Le regole di mapping illustrate, pur essendo relativamente semplici, introducono notevoli complicazioni al processo di trasformazione, infatti, non è sufficiente fare una semplice sostituzione di nomi ma occorre provvedere alla trasformazione dell'intero fattore booleano in cui l'attributo è inserito.
Per capire quanto detto è possibile mostrare alcune delle situazioni caratteristiche, ad esempio, utilizzando ancora lo schema di riferimento introdotto nella sezione 2.1, immaginiamo di dover tradurre, per la classe `SU.Research_Staff` della sorgente `University`, la seguente query:

[5] un iteratore è un nome unico all'interno della query stessa che permette di individuare in modo univoco una classe ed i suoi attributi. Gli iteratori sono necessari, ad esempio, per distinguere attributi con lo stesso nome appartenenti a classi differenti.

```
select e_mail
from University_Person
where name = "Tracy Miller"
```

la proprietà globale `name` viene mappata localmente nella coppia di attributi `first_name` e `last_name`, pertanto, il predicato di selezione dovrà essere sostituito da una coppia di condizioni poste in *and* ed ottenute decomponendo la stringa "$Tracy\ Miller$". Il risultato di tale traduzione è dunque:

```
select e_mail
from Research_Staff
where first_name = "Tracy"
and last_name = "Miller"
```

Un'altra situazione tipica è rappresentata dalla presenza di una navigazione implicita, la quale deve essere tradotta localmente in un insieme di join espliciti. In questo caso occorre modificare sia la clausola from, sia il predicato di selezione, in modo da introdurre la condizione di join. Ad esempio la query:

```
select dept.budget_found
from University_Person
```

per la classe `SU.Research_Staff` porterà alla generazione della seguente query locale:

```
select b.budget
from Research_Staff as a,
     Department as b
where a.dept_code = b.dept_code
```

Allo scopo di gestire queste situazioni, si è realizzato il processo di trasformazione dotando tutti gli elementi coinvolti delle funzionalità necessarie a generare la loro traduzione. In particolare:

- gli elementi della mapping table sono stati modellati come oggetti che implementano le regole di mapping specializzando il metodo *toQuery()* della classe **MappingElement**. Questo metodo restituisce un insieme di oggetti, specializzazione della classe **Oql_Query**, che devono essere impiegati per ottenere la rappresentazione locale;

- gli oggetti che rappresentano le espressioni della query implementano il metodo *translateQuery()* che, gestendo le informazioni relative all'espressione stessa e ai suoi operandi, restituisce gli oggetti che devono essere usati per generare la query locale.

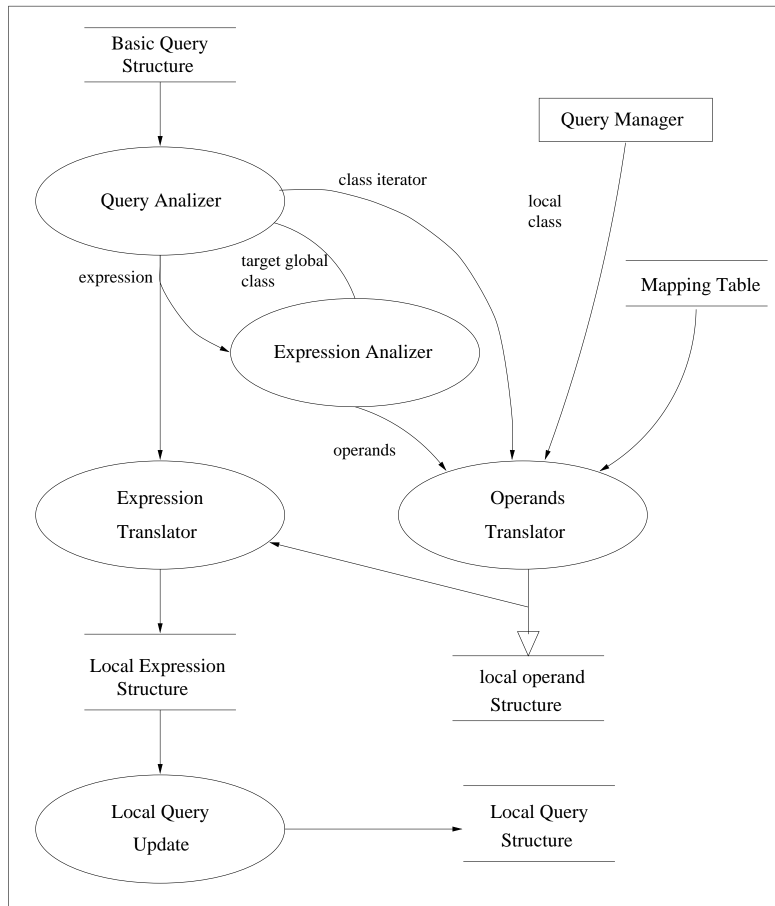

Figura 5.5: Trasformazione di una query

In questo modo, viene adottato un approccio modulare che, da un lato, permette una più semplice ed efficiente gestione del processo di trasformazione e, dall'altro, rende possibile introdurre nuove regole di mapping o modificare quelle esistenti semplicemente definendo nuove classi di oggetti che specializzino i metodi indicati.
Come illustrato in Figura 5.5, il meccanismo che è stato implementato è di tipo **top-down**, infatti i componenti complessi presenti nell'interrogazione (come possono essere espressioni relazionali o la query stessa), forniscono la loro traduzione agli operatori di livello superiore elaborando le informazioni ottenute dai propri componenti. Questo procedimento viene iterato sino ad arrivare ai singoli operandi, cioè attributi semplici o costanti, che otterranno le rappresentazioni locali direttamente dagli elementi della mapping table.
Perché tutto funzioni correttamente occorre, dunque, poter trasferire informazioni,

cioè strutture dati complesse, da un oggetto all’altro e a tale scopo è stata definita la classe **TransOutput** (sezione 5.5.4). Questa struttura è in grado di rappresentare tutte le informazioni ottenute dalla trasformazione di un operatore e viene dunque impiegata per generare gli operatori della query locale.

5.3 Il package “oql”

Il package *oql* raccoglie le classi introdotte per descrivere la struttura dati necessaria a rappresentare il contenuto delle query OQL. Esse sono dunque state definite per poter generare, in memoria centrale, un’immagine della query che costituisca il punto di partenza per qualsiasi tipo di elaborazione.

La grammatica del linguaggio OQL (riportata nella forma BNF in appendice D), prevede un insieme di espressioni che possono essere composte ricorsivamente in modo da ottenerne strutture più complesse sino ad arrivare alla definizione dell’interrogazione finale. La soluzione più adatta per rappresentare una situazione di questo tipo consiste, pertanto, nel definire una gerarchia di ereditarietà in cui la radice generalizza il concetto di espressione e le sotto-classi vengono associate ai singoli tipi di espressioni, in modo da descriverne il contenuto informativo. I vantaggi di questo approccio sono sostanzialmente due:

- la ricorsione implicita nella grammatica OQL viene implementata in modo del tutto naturale, sfruttando il meccanismo di ereditarietà di Java. In questo modo le classi definite possono essere composte per ottenere una qualsiasi query OQL;

- nella super-classe possono essere definite le interfacce di metodi che, ereditati e specializzati dalle varie espressioni, permettono l’implementazione delle funzionalità necessarie alla gestione delle query.

Le espressioni definite nella sintassi del linguaggio OQL sono state organizzate nella gerarchia di ereditarietà riportata in Figura 5.6. Le classi in essa presenti, oltre a prevedere le proprietà necessarie a descrivere la struttura dell’espressione, implementano il metodo **translateQuery()**. Questo metodo, come già spiegato nella sezione 5.2.2, viene impiegato nella fase di trasformazione delle query al fine di generare l’insieme di strutture dati che ne rappresentano la traduzione locale. L’interfaccia di **translateQuery()** prevede i seguenti parametri:

- la mapping table:
  rappresenta la classe globale su cui è posta la query[6]. Questo parametro descrive la classe globale e le regole di mapping, pertanto

[6]le query gestite durante la fase di trasformazione sono quelle di tipo basic

Figura 5.6: Gerarchia di classi per la rappresentazione delle query oql

contiene le informazioni necessarie a trasformare gli operandi presenti nell'espressione;

- la classe locale:
  indica la classe locale rispetto cui deve essere trasformato l'oggetto, individuando, così, una particolare riga nella mapping table.
  Mediante questo parametro è anche possibile recuperare, dalla mapping table, la descrizione della sorgente a cui appartiene la classe locale;

- la query locale:
  rappresenta la struttura dati contenente il risultato delle trasformazioni già effettuate sugli elementi della basic query. Questo parametro è di notevole importanza, infatti la query locale è una struttura complessa che può contenere molte informazioni necessarie al processo di trasformazione, in particolare nella gestione degli iteratori da associare alla classi locali coinvolte nella query locale.

Il metodo restituisce poi un oggetto istanza della classe **TransOutput**(paragrafo 5.5.4), in cui vengono riportati tutti gli elementi che devono essere inseriti nella query locale in fase di costruzione.

5.4 Il package "queryman"

All'interno di questo *package* sono state collocate tutte le classi necessarie alla gestione delle query, cioè alla generazione del piano e alla sua esecuzione. Queste classi sono organizzate nella struttura dati mostrata in Figura 5.7, in cui viene evidenziato come, la query globale creata dal **QueryManager**, sia formata da subquery secondo una struttura a tre livelli che ne rappresenta il piano di esecuzione. Andiamo ora a descrivere più in dettaglio le classi Java realizzate.

5.4.1 La classe "QueryManager"

Questa classe ha un'interfaccia minima costituita dall'attributo *schema* e dal metodo *createQuery()*:

- attributo *schema*
  questa proprietà è in realtà un riferimento ad un oggetto complesso di tipo **GlobalClass** che descrive l'intero schema globale, riportando di ogni classe globale il nome ma anche le altre strutture dati necessarie all'attività di query processing: la mapping table, la gerarchia estensionale e le "$base\ extension$" (vedi sezione 5.5 ;

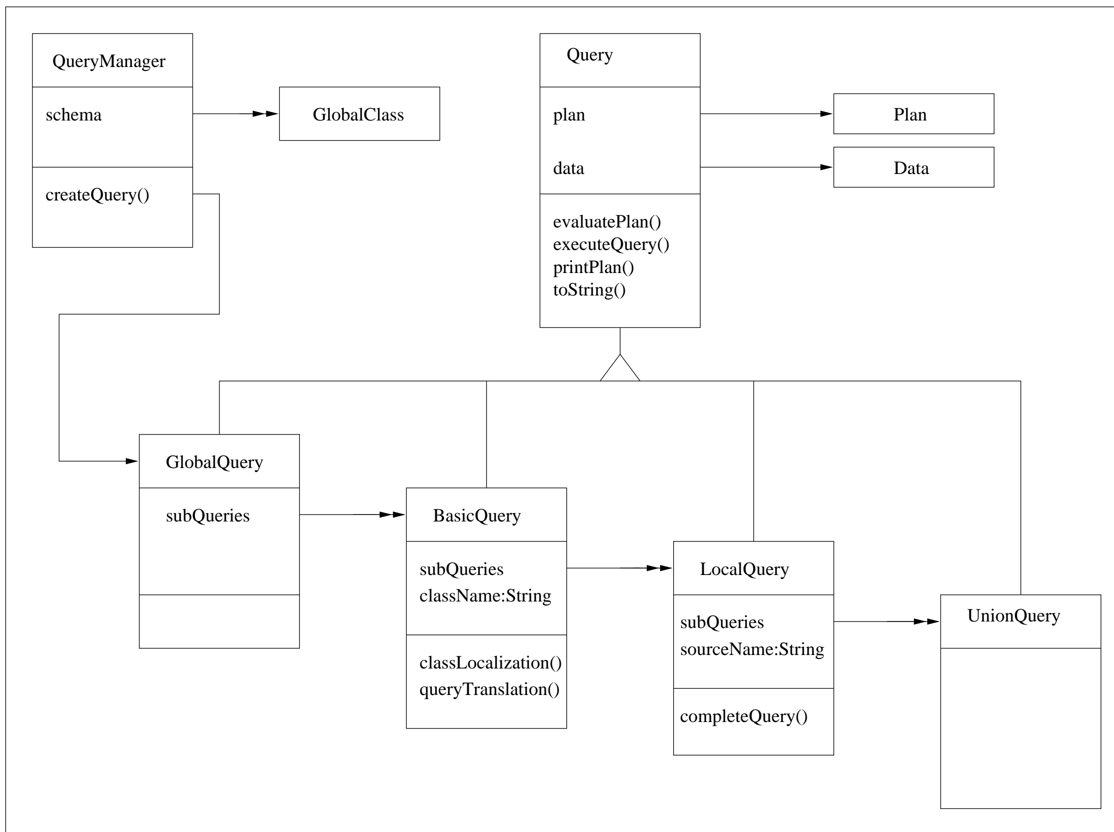

Figura 5.7: Modello ad oggetti del modulo queryman

- metodo *createQuery()*
  il metodo implementa le fasi di "*Query Parsing and Validation*" e di "*Plan Definition*", riportate in Figura 5.2, pertanto riceve il testo della query globale producendo in uscita un insieme di strutture dati che ne rappresentano il piano di esecuzione. Si ottiene, quindi, un *execution plan* composto da un insieme di basic query ed una sequenza di operazioni. Mentre le prime permettono la generazione delle viste sulle classi globali, le operazioni sfruttano questi dati per ottenere la valutazione della query nel suo complesso. Ad ogni basic query viene poi associato un piano di accesso che specifica come devono essere recuperati ed integrati i dati nelle sorgenti.
  Il piano di esecuzione e le basic query individuate vengono rispettivamente assegnate ai campi *plan* (ereditato dalla classe query), e *subQueries* della classe **Global Query**;

Occorre dire che l’obiettivo di questa tesi è stato quello di approfondire le problematiche proprie di un Mediatore e dunque si è deciso di gestire soltanto interrogazioni “*semplici*”, cioè già nella forma propria delle basic query. Ciò significa che la generazione del piano associato alla query globale è estremamente semplificata, siccome, basterà generare un sola **Basic Query**, corrispondente alla query globale iniziale, e definire un piano in cui è sufficiente invocare l’esecuzione della query prodotta.

Rimane forse da osservare che la complessità del sistema, la mole di dati gestita e l’eterogeneità di questi ultimi, può rendere complessa la formulazione di interrogazioni efficienti, pertanto, diviene un’esigenza primaria l’implementazione di una potente interfaccia grafica che guidi l’utente nella fase di definizione dell’interrogazione. I prossimi sviluppi del sistema MOMIS dovrebbero quindi essere volti alla definizione ed implementazione di tool grafici di supporto sia al progettista nella fase di integrazione, sia all’utente durante la formulazione delle interrogazioni.

5.4.2 La classe “Query”

Come illustrato in Figura 5.7 questa classe generalizza le query che compongono il piano di esecuzione. L’interfaccia da essa implementata prevede i seguenti campi e metodi:

- attributo *plan*
  è un campo complesso definito allo scopo di contenere le indicazioni che rappresentano il piano associato alla query. Questo oggetto può essere pensato come un vettore di stringhe in cui, ogni elemento, rappresenta l’invocazione di una funzione implementata dal Query Manager. Ad esempio nel piano associato ad una query globale potrebbe essere rappresentata una sequenza del tipo:

  ```
  Q1:execute(BasicQuery1);
  Q2:execute(BasicQuery2);
  join(Q1,Q2,Q1.code = Q2.code, nestedloop);
  project(Q1.name, Q2.salary);
  ```

  le prime due istruzioni generano la risposta delle basic query mentre le altre due effettuano il join delle informazioni ottenute, recuperando gli attributi globali richiesti, `name` e `salary`;

- attributo *data*
  anche questo è un campo complesso volto a contenere il risultato ritornato dall'esecuzione del piano e, quindi, delle sottoquery in esso presenti. Gli oggetti di tipo **Data** possono essere pensati come vettori di tuple in cui ogni elemento può essere un oggetto complesso;

- metodo *evaluatePlan()*
  facendo riferimento alle strutture dati che definiscono lo schema globale viene generato il contenuto dall'attributo *plan*, vengono cioè individuate le subquery che dovranno essere eseguite e sono definite le operazioni da effettuare sui dati prodotti;

- metodo *executeQuery()*
  eseguire una query significa mettere in atto il piano, *plan* ad essa associato. Questo metodo dovrà quindi effettuare il parsing dei comandi in esso presente ed invocare le corrispondenti funzionalità;

- metodo *toString()*
  fornisce una stringa contenente il testo della query rappresentata dall'oggetto;

- metodo *printPlan()*
  questo metodo serve per dare una rappresentazione del piano in modo che un utente "*esperto*" possa valutarle ed eventualmente decidere di modificarlo.

Una possibile estensione del sistema consiste nel dare all'utente la possibilità di intervenire sul piano generato per migliorarne l'esecuzione. Ovviamente, perché ciò possa essere fatto in modo efficace, occorre disporre di strumenti che, utilizzando gli schemi locali e globale ed eventuali informazioni statistiche sulle sorgenti, siano in grado di valutare la cardinalità della risposta e il costo di esecuzione.
Inoltre sarebbe opportuno prevedere dei metodi per il monitoraggio dello stato di esecuzione della query.

Le specializzazione della classe **Query** sono: **GlobalQuery**. **BasicQuery**, **LocalQuery** e **UnionQuery**. In particolare le prime tre estendono l'interfaccia aggiungendo l'attributo *subQueries*, cioè un vettore in cui vengono memorizzate le query di livello inferiore, cioè quelle individuate durante la generazione del piano.
Per le classi **BasicQuery** e **LocalQuery** è opportuno fare alcune considerazioni aggiuntive:

BasicQuery questa classe estende l’interfaccia presente in **Query** definendo un insieme di metodi volti alla costruzione del piano di accesso. Il metodo *evaluatePlan()* deve pertanto incaricarsi dell’individuazione delle sorgenti significative ai fini dell’interrogazione, provvedendo poi alla generazione delle query riferite ai singoli schemi e alla definizione delle operazioni che devono essere eseguite per ricomporre i risultati parziali.
Dopo aver determinato la classe globale, target della basic query, viene effettuata una fase di analisi volta ad individuare le classi locali che possono contribuire a generare la risposta. Questa operazione è eseguita dal metodo *classLocalization()* che, sfruttando le informazioni rappresentate nella gerarchia estensionale e nelle “*base extension*”, individua le sorgenti coinvolte e fornisce le indicazioni necessarie alla ricomposizione dei risultati delle query locali. Le operazioni da definire sono sostanzialmente le invocazioni dei metodi del Query Manager che implementano gli operatori di equi-join(necessarie per la ricostruzione delle “*base extension*”), gli outer-join (impiegati per combinare “*base extension*” sovrapposte) e le unione (usati per unire le entità di base extension disgiunte). In realtà non è ancora disponibile il modulo per la generazione della conoscenza estensionale, dunque in questa fase, ci si limita a fornire tutte le classi che compongono il cluster.

LocalQuery questa classe, specializzando il metodo *evaluatePlan()*, provvede alla definizione dalle query da inviare alle sorgenti. Come è già stato descritto questa operazione di trasformazione viene effettuata mediante un approccio **top-down** che sfrutta le interfacce definite per gli elementi della query e della mapping table. Il metodo *evaluatePlan()* si limita quindi ad invocare la funzione di trasformazione dell’oggetto di tipo **Oql_Query** contenente la struttura della basic query. Questo meccanismo è estremamente versatile in quanto non è necessario prevedere quale particolare struttura abbia la query o quali elementi essa possa contenere, in quanto saranno gli elementi stessi a provvedere alla generazione dell’elemento corrispondente alla loro traduzione.

Osservando lo schema ad oggetti di Figura 5.7 si nota che le local query non sono in effetti l’ultimo elemento del piano. Nel caso di sorgenti semistrutturate, è infatti possibile che, per ogni classe locale, vengano generate più query in modo da prevedere tutte le possibili rappresentazioni associate ad uno stesso oggetto. In queste condizioni sarà pertanto necessario prevedere la presenza anche di un campo *plan* in cui sia possibile specificare come queste query, che indichiamo come *Union Query*, debbano essere utilizzate per generare l’insieme di dati corrispondenti alla query locale. Questi aspetti sono stati studiati per consentire estensioni future ma non sono stati implementati in quanto la gestione di dati semistrutturati non rientra negli obiettivi di questa tesi.

Rimane da dire che le query prodotte da questa fase non sono ancora quelle che verranno inviate alle sorgenti, infatti occorre eseguire la fase di ottimizzazione semantica locale, nella quale, mediante l'ausilio di ODB-Tools e sfruttando le conoscenze semantiche descritte nei singoli schemi, vengono effettuate tutte quelle espansioni che possono migliorare l'esecuzione della query stessa sulle sorgenti.

5.4.3 Le classi "Plan" e "Data"

All'interno del package sono state inserite anche le classi per la rappresentazione dei piani di esecuzione e dei dati restituiti delle query. Sebbene queste classi siano caratterizzate da un'interfaccia molto semplice, composta da un unico attributo di tipo *Vector*, il loro inserimento è estremamente importante, in quanto semplifica la realizzazione di future estensioni. A tale scopo è infatti sufficiente specializzare queste classi e realizzare le interfacce necessarie, senza dover rivedere il software sviluppato in precedenza.
Ad esempio la classe **Plan** è stata specializzata mediante la classe **UQPlan** per consentire una corretta gestione delle query locali ed in particolare delle loro componenti, cioè le Union Query. Queste interrogazioni sono rivolte a classi presenti nelle sorgenti quindi restituiranno un risultato che è espresso in termini locali, ciò significa che non si avrà necessariamente una corrispondenza diretta tra attributi globali e locali. Considerando una singola union query le situazioni che si possono presentare sono:

- gli attributi globali corrispondono ad attributi locali aventi una semantica diversa, cioè con nomi e strutture differenti. Ad esempio l'attributo globale `name` della classe `University_Person` dello schema di riferimento viene tradotta, per la classe locale `SU.Research_Staff` nella coppia di attributi `first_name`, `last_name`;
- un attributo globale può non avere un corrispondente attributo nella query locale.

In base a queste considerazioni risulta evidente la necessità di disporre di una struttura dati che permetta la traduzione dei dati reperiti localmente nelle corrispondenti strutture globali. Solamente in questo modo, infatti è possibile dare ai risultati parziali una rappresentazione omogenea tale da consentire la loro integrazione. Durante la fase di ricomposizione della risposta occorre quindi effettuare un'operazione inversa rispetto a quella di trasformazione dell'interrogazione. Ad esempio, gli attributi `first_name`, `last_name`, precedentemente indicati, devono essere composti in modo da fornire un unico attributo `name` che possa essere confrontato con il corrispondente campo ottenuto dalle altre query locali.

Le informazioni necessarie alla fase di integrazione dei risultati sono contenute nei campi della classe *UBPlanElement* le cui istanze costituiscono gli elementi del piano associato alle union query. Questi attributi sono:

- *globalProperty*: descrive l'attributo globale;
- *localProperties*: descrive l'insieme di attributi locali corrispondenti al campo *globalProperty*;
- *mapping*: è un riferimento all'elemento della mapping table che ha portato alla trasformazione della proprietà globale in proprietà locali. Mediante questo riferimento è quindi possibile invocare il metodo che permette la composizione dei dati generando la rappresentazione globale.

5.5 Il package "globalschema"

In questo package sono raccolte le classi definite per la descrizione e gestione dello schema globale.
Lo schema integrato viene rappresentato nella classe **Global Class** come un insieme di classi globali ognuna delle quali è descritta dai seguenti attributi:

- *globalSchema*: indica il nome dello schema globale a cui appartiene la classe;
- *globalClassName*: indica il nome della classe globale rappresentata dall'oggetto;
- *mappingTable*: è un campo complesso che descrive la mapping table associata alla classe globale;
- *extHierarchy*: questo attributo è un riferimento all'oggetto complesso che descrive la gerarchia estensionale costruita sulla classe globale;
- *baseExtensions*: è un vettore i cui elementi sono oggetti complessi che descrivono le "$base\ extension$" individuate nella classe globale;

5.5.1 La classe "MappingTable"

La mapping table è stata introdotta allo scopo di descrivere le classi globali che compongono lo schema integrato, in essa devono quindi essere raccolte le seguenti informazioni:

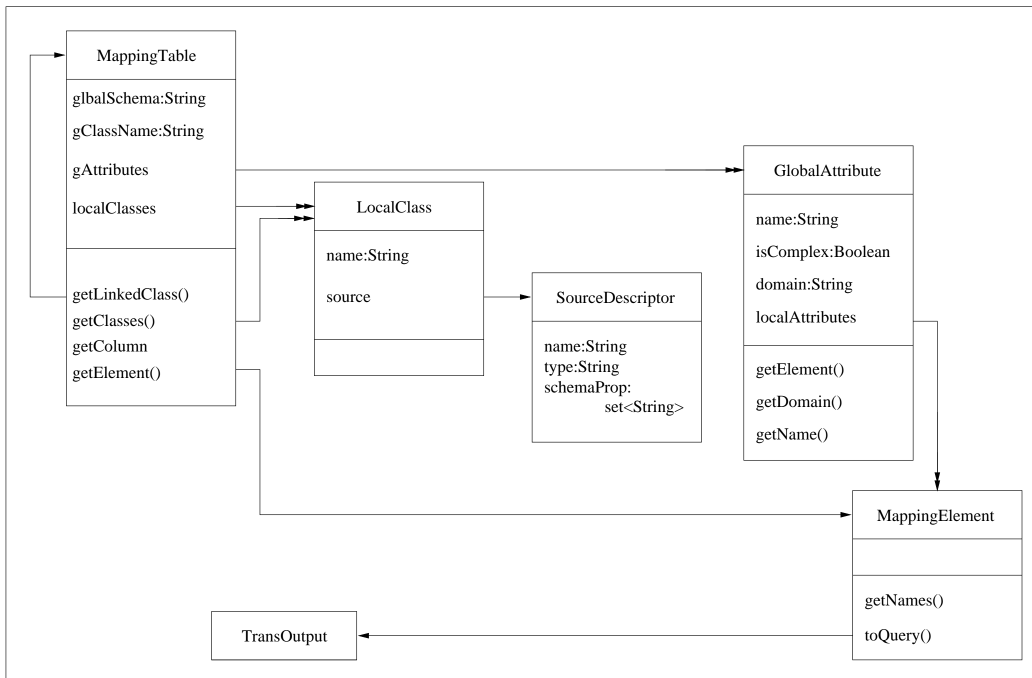

Figura 5.8: Modello ad oggetti della classe MappingTable

- una descrizione degli attributi globali presenti nell'intensione della classe globale. Per ogni attributo deve essere dunque specificato il nome ed eventualmente il tipo;
- per ogni attributo globale deve essere descritto il modo in cui questo è rappresentato nelle classi locali che formano il cluster;

La soluzione adottata per rappresentare questa struttura dati è mostrata in Figura 5.8.
Sebbene ad un primo esame possa sembrare non evidente, i suoi elementi sono organizzati in una struttura logica tabellare in cui:

- la prima colonna è formata da oggetti di tipo **LocalClass** che descrivono le classi locali appartenenti al cluster. Ognuno di questi oggetti ha anche un riferimento ad un'istanza della classe **SourceDescriptor** in cui sono raccolte le informazioni relative alla sorgente di appartenenza;
- nella prima riga vengono descritti gli attributi globali. Ogni attributo corrisponde ad un oggetto istanza della classe **GlobalAttribute** che, oltre al

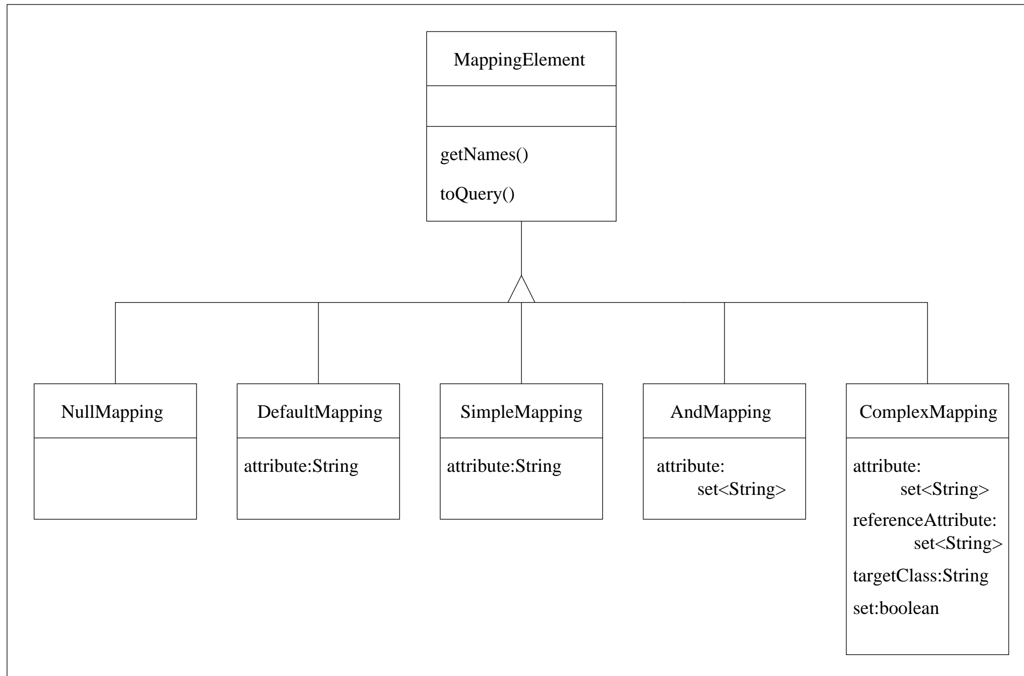

Figura 5.9: Modello ad oggetti della classe MappingElement

nome, presenta un riferimento alla colonna della mapping table in cui sono descritte le rappresentazioni locali dell’attributo stesso;

- gli elementi interni alla mapping table sono modellati mediante la classe **MappingElement**. Come illustrato in Figura 5.9, questa classe è la radice di una gerarchia di ereditarietà e le sue specializzazioni descrivono il modo in cui, allo stato attuale, un attributo globale può essere rappresentato negli schemi locali;

Ogni componente della struttura viene dunque modellato come un oggetto complesso dotato di un proprio comportamento ottenendo così un’elevata modularità e flessibilità.
Ad esempio, le classi **MappingTable** e **GlobalAttribute** implementano i metodi necessari alla navigazione attraverso le classi dello schema e al reperimento, da ogni mapping table, dei suoi componenti. Di particolare interesse è la classe **MappingElement** infatti, come è possibile osservare in Figura 5.9, è la radice di una gerarchia di ereditarieà le cui sotto-classi implementano il metodo *toQuery()* il quale appartiene all’interfaccia usata per la gestione delle rappresentazioni locali. Questo metodo viene usato nella fase di trasformazione delle interrogazioni

per tradurre gli attributi globali nei corrispondenti schemi locali. Esso riceve in ingresso i parametri:

- *context*: indica il contesto in cui è collocato (all'interno della query), l'attributo globale a cui corrisponde la regola di mapping. Ad esempio l'attributo può essere uno degli elementi che formano una path expression o un operando in un confronto;

- *condition*: indica una condizione che può essere usata per condizionare la rappresentazione restituita. Questo parametro è stato introdotto principalmente per gestire la *Union Correspondence* descritta nel paragrafo 5.9. In questo caso, infatti, è necessario poter selezionare una particolare rappresentazione tra più alternative possibili.

Il risultato dell'elaborazione viene poi restituito in un oggetto istanza della classe **TransOutput**. In realtà vengono usati solo due dei campi di questa struttura infatti nel campo *mappingField* di questa struttura devono essere inseriti tutti gli oggetti che descrivono la rappresentazione locale dell'attributo e nel campo *mappingElement* viene memorizzato un riferimento all'elemento stesso della mapping table.
Le varie specializzazioni di *MappingElement* devono ridefinire il metodo *toQuery()*, in modo da adattarne il comportamento alle proprie esigenze. Ovviamente, ciò deve essere fatto mantenendo la stessa interfaccia, ad esempio:

- le classi **NullMapping** e **DefaultMapping** restituiscono un oggetto di tipo **TransOutput**, contenente un solo elemento nel cui campo *mappingField* è inserito il valore di default. Questo valore è rappresentato mediante il corrispondente oggetto di tipo **Oql_Basic**, cioè per un valore "*null*" verrà usato un oggetto **Oql_Nil**, mentre per un valore costante il comportamento viene ulteriormente specializzato in modo da fornire un oggetto **Oql_Long**, **Oql_Float**,**Oql_OOString** in funzione delle esigenze particolari;

- la classe **SimpleMapping** restituisce, nel campo *mappingField*, un oggetto di tipo **Oql_Identifier** che rappresenta l'attributo locale (essendo un attributo semplice il parametro *refAttr* di questo oggetto deve corrispondere alla stringa vuota);

- la classe **AndMapping** ha un comportamento simile a quello di un attributo semplice ma, anziché restituire un solo oggetto, deve restituirne un insieme di oggetti;

- la classe **ComplexMapping** viene usata per rappresentare gli attributi che mappano su altre classi, pertanto verrà restituito un insieme di oggetti di tipo **Oql_Identifier** contenenti le informazioni necessarie a ricostruire il riferimento. Ognuno di essi deve quindi indicare il nome locale dell'attributo, la classe su cui mappa ed, eventualmente, il corrispondente attributo nella classe riferita[7].

Le regole di mapping implementate da questi elementi permettono la gestione dei conflitti più frequenti tra schemi, ma non esauriscono certo le situazioni che possono presentarsi e che devono essere risolte. L'efficacia del metodo impiegato per l'implementazione delle regole di mapping (sono comportamenti associati agli elementi stessi), consiste proprio nella possibilità di estendere i conflitti gestibili, semplicemente realizzando nuove classi che implementino il metodo *toQuery()* rispettando l'interfaccia descritta.
In questo modo la mapping table diviene una struttura componibile che può essere estesa ed adattata in funzione delle esigenze che si presentano nelle singole applicazioni.

5.5.2 La classe “BaseExtension”

Le “$base\ extension$” fanno parte della conoscenza estensionale che deve essere generata nella fase di integrazione degli schemi. Esse rappresentano insiemi disgiunti di entità appartenenti al dominio applicativo e sono individuate dall'intersezione di classi locali di uno stesso cluster.
In Figura 5.10 viene riportato lo schema ad oggetti di questa classe, in esso sono presenti i seguenti campi:

- *attributes*: è un vettore di stringhe in cui sono riportati i nomi degli attributi che compongono l'intensione della “$base\ extension$”;
- *localClasses*: indica l'insieme di classi locali presenti nella “$base\ extension$”. Questo attributo è quindi un vettore di oggetti di tipo **SourceClass**;
- *joinMap*: come già osservato, l'estensione della base extension è rappresentata dall'intersezione delle classi locali in essa presenti. Per generare tale estensione occorre quindi specificare come queste classi debbano essere combinate, cioè quali sono le operazioni di join da eseguire.

[7] se il riferimento viene rappresentato localmente mediante una foreign key allora gli attributi che permettono il collegamento (il join) delle due classi locali in relazione devono essere in numero uguale ma possono avere nomi diversi, pertanto per ogni attributo nella classe di partenza occorre specificare quale sia il corrispondente attributo nella classe di arrivo.

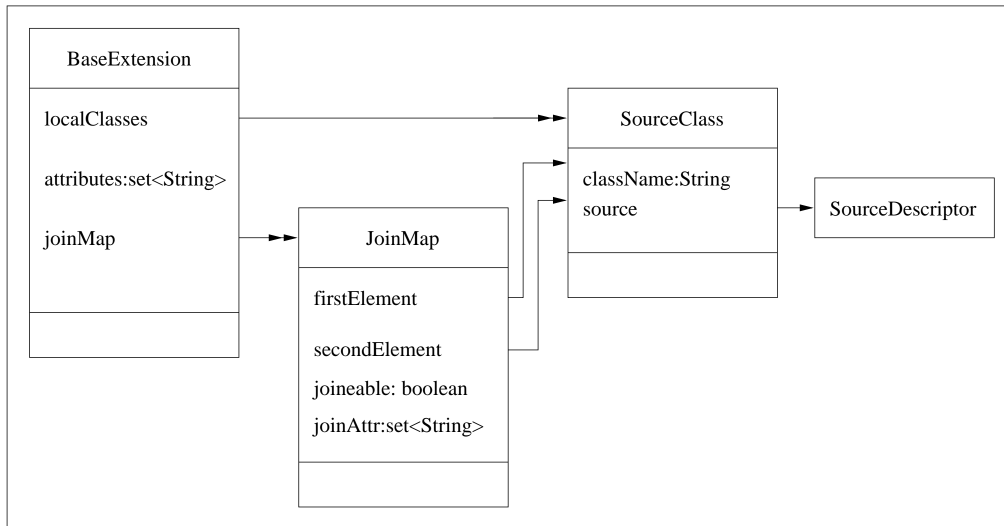

Figura 5.10: Modello ad oggetti della classe BaseExtension

L'attributo *joinMap* può essere quindi pensato come una matrice in cui, per ogni coppia di classi locali della "*base extension*", viene indicato se è possibile effettuare il join e, in caso affermativo, quali sono gli attributi da utilizzare in questa operazione. Si noti che la chiave semantica può non essere unica e non è detto che ogni classe abbia tutte le chiavi quindi può accadere che per combinare due classi sia necessario usarne una terza di collegamento.

Rimane da osservare che, per le classi locali, deve essere riportato anche un riferimento alla sorgente di appartenenza, infatti, come descritto nel paragrafo 4.2, questa informazione può essere utilizzata nella gestione delle interrogazioni per ottenere ulteriori ottimizzazioni nel piano di accesso.

5.5.3 La classe "ExtensionalHierarchy"

Come illustrato in Figura 5.11 questa classe ha un interfaccia composta da:

- attributo *virtualClasses*: è un vettore contenente le classi appartenenti alla gerarchia estensionale. Queste classi sono modellate mediante l'oggetto complesso **VirtualClass** che è caratterizzato dall'avere un'intensione ed un estensione: l'intensione è costituita da un insieme di nomi di attributi globali appartenenti al cluster associato alla gerarchia, mentre l'estensione è rappresentata da un insieme di base extension (quelle che posseggono tutte le proprietà presenti nell'intensione della classe virtuale);

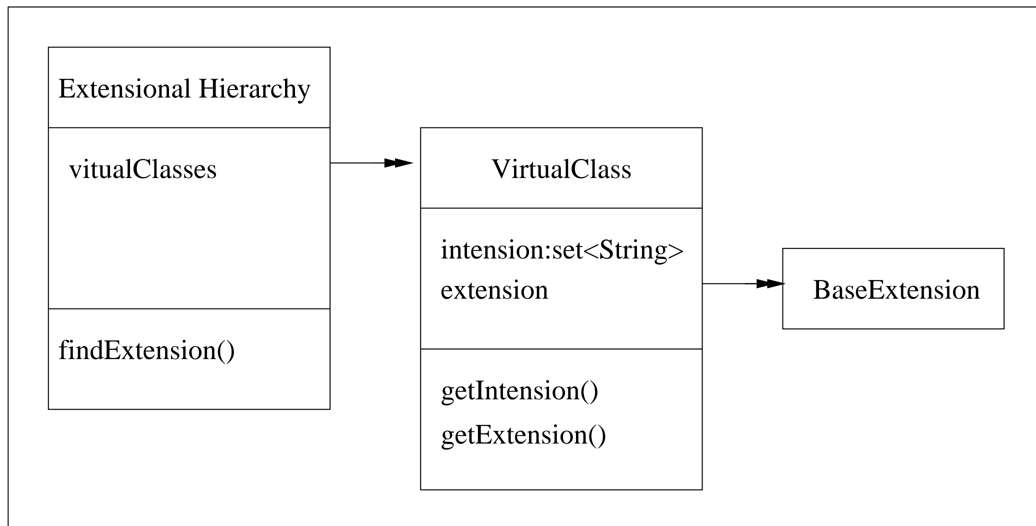

Figura 5.11: Modello ad oggetti della classe ExtensionalHierarchy

- metodo *findExtension()*: ricevendo in ingresso una lista di attributi globali deve restituire un insieme di "*base extension*".
  In questo metodo, per prima cosa, viene analizzata la gerarchia estensionale, allo scopo di individuare le classi virtuali che comprendono nella loro intensione l'insieme di attributi passato come parametro. A questo punto si può individuare l'insieme di "*base extension*" cercato, attraverso l'unione delle estensioni delle classi virtuali selezionate.

5.5.4 La classe "TransOutput"

La classe **TransOutput** è stata introdotta per poter implementare il meccanismo di trasformazione illustrato nel paragrafo 5.2.2, in essa possono essere infatti raccolte tutte le informazioni necessarie a rappresentare le traduzioni degli operatori presenti nelle query OQL.

Per comprendere la struttura di questa classe (Figura 5.12) occorre considerare due aspetti:

- in presenza di sorgenti semistrutturate, ad una stessa espressione può corrispondere un insieme di traduzioni differenti, infatti i dati semistrutturati non sono riconducibili ad uno schema preciso e ad uno stesso dato o concetto possono corrispondere oggetti di tipo diverso. Una possibile soluzione per rappresentare e, soprattutto, per recuperare oggetti con strutture differenti

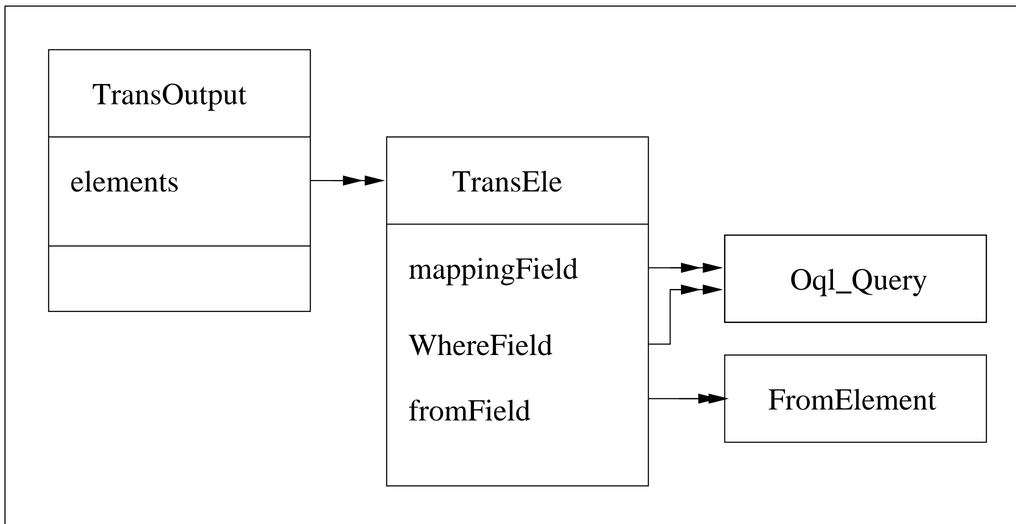

Figura 5.12: Schema della classe TransOutput

consiste appunto nel generare una diversa traduzione per ogni tipo tipo di oggetto individuato;

- ogni rappresentazione locale corrispondente ad una stessa espressione può richiedere oggetti di tipo diverso e, soprattutto, con differenti ruoli all'interno della query. Se ad esempio vogliamo tradurre una path expression per una sorgente relazionale, allora non basta inserire un semplice attributo nell'interrogazione locale ma occorre aggiungere anche una condizione di join. Ciò significa dover inserire una o più classi nella clausola from ed il corrispondente insieme di condizioni di equi-join nella clausola where.

Per rispondere a queste esigenze la classe **TransOutput** è stata modellata come un vettore di oggetti complessi.
Al fine di descrivere le singole traduzioni, cioè gli elementi del vettore, è stata introdotta la classe **TransEle** in cui è possibile rappresentare le sotto-espressioni della query locale suddividendoli in funzione della loro collocazione.

5.6 Il package "utility"

Il package *utility* raccoglie un insieme di classi che implementano funzionalità di carattere generale. Queste classi vengono cioè utilizzate nell'ambito del sistema MOMIS per realizzare la fase di query processing ma possono essere riutilizzate anche in contesti diversi.

5.6.1 La classe "parser"

Un parser è un analizzatore sintattico, cioè un modulo software che, basandosi su regole grammaticali opportunamente definite, è in grado di verificare la correttezza sintattica di una data espressione. La classe realizzata nel package *utility*, in particolare, implementa un parser per il linguaggio OQL, pertanto il suo obiettivo consiste nell'analizzare le query fornite in ingresso allo scopo di accertare la loro correttezza e produrre una struttura dati che ne rappresenti il contenuto.
Per realizzare questo modulo software è stato utilizzato "$BYACC$" [55], cioè un'utility di pubblico dominio, in grado di generare, a partire dalla descrizione LALR (Look Ahead Left Recursive) della grammatica, la classe Java che ne implementa il corrispondente parser.
La grammatica viene fornita a "$BYACC$" mediante un file in formato *YACC* in cui possono essere specificate anche azioni, le quali definiscono il comportamento da associare alla fase di parsing. Un file di questo tipo è caratterizzato da tre sezioni ordinate nel seguente modo:

1. *definizioni*: nella parte iniziale del file devono essere indicate le classi Java ed i package che da importare per consentire il funzionamento del parser. Inoltre, è questa la sezione in cui devono essere definiti i *token*, cioè i simboli che costituiscono l'alfabeto della grammatica;

2. *regole grammaticali*: rappresentano la descrizione LALR della grammatica, cioè l'insieme di regole di produzione che realizza il riconoscimento sintattico delle espressioni. Ogni statement presente in questa sezione è individuato da tre elementi:

    - un nome che identifica in modo univoco la regola di produzione;
    - la regola grammaticale, cioè un'espressione formata da token e nomi di regole;
    - un'azione, da invocare quando viene verificata la corrispondente regola;

3. *metodi*: in questa sezione il programmatore può aggiungere metodi che vengono associati alla classe **parser**. Questi metodi possono, dunque, essere invocati nelle azioni associate alle regole o, più in generale, per associare un determinato comportamento al parser stesso;

Particolarmente importante è la possibilità di associare un azione, cioè un comportamento, alle singole regole grammaticali, infatti, contestualmente al riconoscimento sintattico, è possibile generare una struttura dati che rappresenti il contenuto dell'espressione analizzata. In particolare. nella classe **parser**, ad ogni

regola viene associata la generazione di una particolare classe *oql* (vedi la sezione refpackage:oql), ottenendo, al termine del parsing, l'immagine della query impiegata dal Query Manager .
Nell'ultima sezione del file di input sono poi stati inseriti alcuni metodi aggiuntivi, in particolare:

- *yylex*: questo metodo implementa in linguaggio Java l'automa a stati finiti impiegato per il riconoscimento sintattico del testo da analizzare. Tale automa permette la scansione del espressione per riconoscerne i singoli elementi, cioè i simboli nell'alfabeto della grammatica ;

- *run*: invoca il parser per effettuare il controllo sintattico e generare la struttura dati Java;

Inoltre sono stati definiti metodo la gestione e segnalazione delle situazioni di errore.

Rimane da dire che, allo scopo di estendere le funzionalità offerte dal parser, nel package *utility* è stata inserita anche la classe **OqlAnalizer**. Questa nuova classe può essere specializzata inserendo metodi per la gestione della query, ad esempio è possibile prevedere una particolare gestione degli errori presenti nella query e riscontrati durante la fase di parsing, oppure, analizzando la struttura dati associata alla query, è possibile implementare il controllo semantico o altri tipi di analisi. Il costruttore della classe **OqlAnalizer** richiede come parametro di ingresso il testo di una query OQL quindi provvede ad invocare il parser gestendo poi la struttura dati e le altre informazioni da esso fornite.

5.6.2 Utility aggiuntive

La classe **Messages** è stata introdotta per consentire una migliore gestione dei messaggi di errore sia durante la fase di parsing, sia nel processo di generazione delle query locali.
Questa classe è rappresenta un vettore i cui elementi devono essere oggetti di tipo **ParEx**. Ogni elemento del vettore è quindi in grado di memorizzare un testo, in cui viene riportato il tipo di errore, ed un indicazione relativa alla posizione dell'errore (ad esempio se questo viene riscontrato durante il parsing di una query).

La classe **PathTokenizer** è stata introdotta, invece, per ottenere la serializzazione di nomi composti mediante la "*dot notation*". Il costruttore di questa classe accetta, quindi, in ingresso una stringa producendo un oggetto contenente, oltre alla stringa stessa, un vettore che ne contiene i singoli elementi

5.7 Il software

Il software prodotto in questa tesi è stato sviluppato utilizzando la versione **jdk1.2beta4** dell'interprete Java. Le classi implementate sono complessivamente 71, suddivise, in quattro package, nel seguente modo:

- package *oql*: 42 classi;
- package *queryman*: 10 classi;
- package *globalschema*: 14 classi;
- package *utility*: 5 classi.

Approssimativamente sono state prodotte 9800 linee di codice commentate. Occorre osservare che i commenti sono stati realizzati rispettando il formalismo richiesto da Javadoc, in modo da utilizzare questo componente per la generazione della corrispondente documentazione. L'impatto dei commenti introdotti, sulla dimensione del codice, è considerevole (approsimativamente 40%), ma li valore della documentazione prodotta giustifica sicuramente lo sforzo fatto. I vantaggi sono infatti:

- tutte le informazioni necessarie alla produzione della documentazione possono essere fornite durante la produzione del codice stesso. In questo modo aumentano le dimensioni del codice, ma si é in grado di fornire descrizioni più accurate e dettagliate;
- possono essere descritto in modo completo tutti gli elementi (attributi e metodi), dell'interfaccia di ogni classe implementata;
- il componente Javadoc produce in modo automatico una documentazione in formato HTML con collegamenti ipertestuali, che, in modo semplice e veloce, consentono di recuperare tutte le informazioni non solo relative alla classe in esame ma anche ai suoi componenti;

In Appendice C viene riportato il codice relativo ad una delle classi implementate (la classe **LocalQuery**), e la corrispondente documentazione ottenuta mediante Javadoc.

Il codice prodotto e la relativa documentazione possono essere trovati nel direttorio */export/home/progetti.comuni/tesi/zaccaria/sw* del server "$Sparc20$" del dipartimento di ingegneria, oppure all'indirizzo *http://sparc20.dsi.unimo.it/tesi/index.html*.

Capitolo 6

Confronto con altri lavori

Nonostante l'Integrazione Intelligente di Informazioni sia un campo di ricerca relativamente nuovo, esistono già in letteratura diversi sistemi che cercano di realizzare, in modo più o meno efficiente e flessibile, un modulo integratore (nei casi più riusciti) o un semplice modulo di ricerca di informazioni. A conclusione di questa tesi, si è ritenuto dunque utile esaminare alcuni di questi sistemi, alla ricerca delle diverse soluzioni adottate.
Per non appesantire questo capitolo, e per poter descrivere i sistemi presentati in modo sufficientemente esauriente, si è scelto di limitare a tre il numero di progetti presentati, cercando di selezionare i più significativi. Si rimanda comunque alla bibliografia per una analisi più approfondita sia di questi sistemi, sia di altri [56, 57, 58, 59].

6.1 TSIMMIS

TSIMMIS (**T**he **S**tanford- **I**BM **M**anager of **M**ultiple **I**nformation **S**ources) [60, 61, 62] è sicuramente uno dei progetti più interessanti in questo campo: sviluppato presso l'Università di Stanford in collaborazione con il Centro di ricerca IBM di Almaden, si pone come obiettivo lo sviluppo di strumenti che facilitino la rapida integrazione di sorgenti testuali eterogenee, includendo sia sorgenti di dati strutturati che **semistrutturati**. Questo obiettivo è raggiunto attraverso un'architettura comune a molti altri sistemi: i *wrapper* convertono i dati in un modello comune mentre i *mediator* combinano ed integrano i dati ricevuti dai wrapper. I wrapper inoltre forniscono un linguaggio di interrogazione comune per l'estrazione delle informazioni, mostrando un'interfaccia verso l'esterno uguale a quella dei mediatori: in questo modo, l'utente può porre le interrogazioni attraverso un unico linguaggio sia ai mediatori (ricevendo dati integrati da più sorgenti), sia direttamente ai wrapper (interrogando in questo modo un'unica fonte).

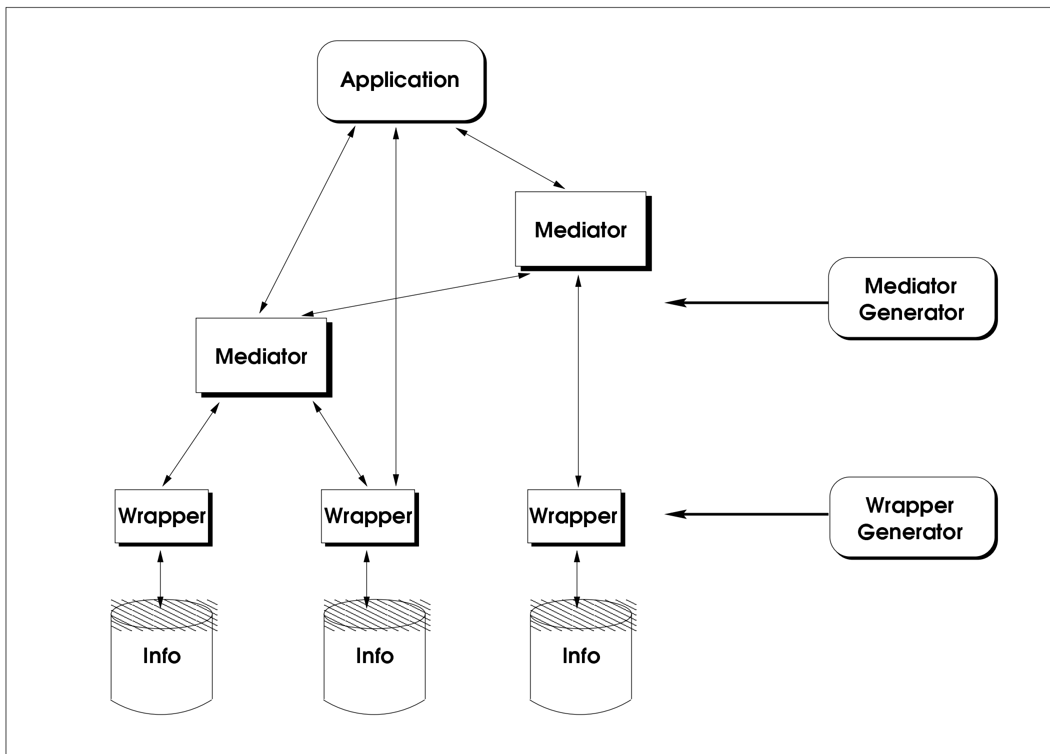

Figura 6.1: Architettura TSIMMIS

In Figura 6.1 è mostrata l'architettura: ad ogni sorgente corrisponde un wrapper (o *traduttore*) che converte nel modello comune i dati estratti dalla sorgente; sopra i wrapper stanno i mediatori. I wrapper convertono inoltre le query scritte utilizzando il modello comune in richieste comprensibili dalla particolare sorgente da loro servita.

La peculiarità di questo progetto si manifesta nel modello comune dei dati utilizzato per rappresentare le informazioni. **OEM** (Object Exchange Model) [63] è un modello a *etichette* basato sui concetti di identità di oggetto e annidamento particolarmente adatto a descrivere dati la cui struttura non è nota o è variabile nel tempo. In aggiunta a questo, sono disponibili ed utilizzati dal sistema due linguaggi di interrogazione (*OEM-QL*) e *MSL* (Mediator Specification Language), per ottenere i dati dalle fonti ed integrarli opportunamente.

Per facilitare il compito dell'amministratore del sistema, sono stati progettati due moduli (*translator generator* e *mediator generator*) che supportano le fasi di realizzazione rispettivamente dei wrapper e dei mediatori, fornendo un insieme di librerie di funzioni predefinite.

6.1.1 Il modello OEM

Si presenta brevemente il modello OEM, fondamentale per capire il tipo di approccio (*strutturale*) dell'intero progetto TSIMMIS. OEM fa parte dei cosiddetti *self describing model*, dove ad ogni informazione è associata una etichetta che ne descrive il significato. Esso comprende caratteristiche tipiche dell'approccio ad oggetti ma in modo molto semplificato: non fa uso di un forte sistema dei tipi (in pratica sono ammessi solamente i tipi base), non supporta direttamente né le classi, né i metodi, né l'ereditarietà, bensì solo l'identità e il nesting tra oggetti.

Un esempio di descrizione dell'oggetto **persona** è il seguente:

```
<ob1:  person,  set,          {sub1,sub2,sub3,sub4,sub5}>
       <sub1:   last_name,    str,  'Smith'>
       <sub2:   first_name,   str,  'John'>
       <sub3:   role,         str,  'faculty'>
       <sub4:   department,   str,  'cs'>
       <sub5:   telephone,    str,  '32435465'>
```

L'oggetto `person` considerato viene rappresentato da un'insieme di oggetti annidati: ogni oggetto è descritto mediante stringhe separate da virgole e comprese fra i simboli $<$ e $>$. La prima stringa (ad es. `ob1`) rappresenta l'identificatore, la successiva è un'etichetta che ne qualifica il genere, la terza e la quarta specificano il tipo ed il valore. Fondamentale, per la semantica dell'oggetto stesso, è l'etichetta, che ne descrive il ruolo all'interno del contesto in cui è utilizzato. Nell'esempio riportato, sono descritti in totale sei oggetti: un oggetto che si potrebbe definire di top-level (`person`) e cinque sotto-oggetti, che sono collegati a `person`, come mostra l'ultima stringa dell'oggetto `ob1`. È importante sottolineare che, usando l'OEM, non è necessario che oggetti che si descrivono tramite la stessa etichetta abbiano pure lo stesso schema: per esempio, potrebbe esistere un altro oggetto `person` con un diverso insieme di tipi di sotto-oggetti. In questo modo risulta molto semplificata l'integrazione di oggetti provenienti da schemi diversi e semistrutturati, infatti non devono essere predefinite le strutture che questi oggetti dovranno seguire, e saranno accettati tutti gli oggetti con una determinata etichetta, qualunque schema seguano.

6.1.2 Il linguaggio MSL

MSL è il linguaggio utilizzato per definire, in modo dichiarativo, i mediatori: attraverso l'uso di *rule*, si può specificare il punto di vista del mediatore, ed in questo modo definire manualmente lo "schema globale" . A run time, ricevuta una particolare richiesta, l'interprete del mediatore (MSI) raccoglie ed integra le

informazioni ricevute dalle sorgenti, in accordo con le specifiche definite nella vista. MSL è dunque un linguaggio dichiarativo per la definizione di *viste* in grado di interfacciarsi con OEM e quindi di supportare evoluzioni degli schemi, irregolarità strutturali degli oggetti nelle sorgenti e strutture sconosciute senza dover ogni volta ridefinire le viste. Ogni regola è costituita da una *testa* e da una *coda*, separate dal simbolo :-. La *coda* descrive dove andare a recuperare l'oggetto che si vuole ricevere, mentre l'intestazione definisce la struttura che questo oggetto dovrà avere, una volta estratto e ricostruito. Un esempio di regola MSL sarà presentato nella Sezione 6.1.4

6.1.3 Il generatore di Wrapper

TSIMMIS include un insieme di strumenti, chiamati *OEM Support Libraries*, per realizzare facilmente i wrapper, i mediatori e le interfacce utenti. Questi strumenti comprendono diverse procedure per lo scambio di oggetti OEM in un architettura Client/Server, dove un Client può essere indifferentemente un mediatore, una applicazione o un'interfaccia utente, mentre il server può essere sia un wrapper, sia un mediatore. In questo modo, si è cercato di semplificare il più possibile la realizzazione dei wrapper, ed in particolare del modulo di Conversione, che ha il compito di convertire una query espressa nel linguaggio MSL in una interrogazione comprensibile dalla sorgente con la quale il wrapper interagisce.

Per facilitare l'intero processo di integrazione, e per poter servire anche sorgenti dalle limitate capacità elaborative, nel progetto TSIMMIS si è ipotizzato di dover esprimere esplicitamente l'intero insieme delle query a cui la sorgente può rispondere (dunque un insieme comunque limitato di interrogazioni) e di predisporre, attraverso i cosiddetti *query templates*, per ogni query supportabile in MSL, la rispettiva traduzione nel linguaggio di interrogazione proprio della sorgente stessa. Un esempio potrà sicuramente semplificare la descrizione di tutto il processo. Supponiamo di interagire con una base di dati universitaria, *WHOIS*, che mantiene informazioni sugli studenti e sui professori di una data università, e con possibilità molto limitate: le uniche operazioni consentite sono il reperimento dei dati di una persona, conoscendone il nome o il cognome (od entrambi). Le interrogazioni supportate dalla sorgente, espresse nel suo linguaggio, potrebbero essere le seguenti:

1. ritrova le persone dato il cognome: `>lookup -ln 'ss'`

2. ritrova le persone dati cognome e nome: `>lookup -ln 'ss' -fn 'ff'`

3. ritrova tutte le informazioni della sorgente: `>lookup`

Ad ognuna di queste interrogazioni corrisponderà un *query template*: le query in entrata al wrapper saranno scritte in MSL, e dovranno essere tradotte, attraverso questi *query template*, nel linguaggio locale. Ad esempio, alle interrogazioni 1 e 2 sopra descritte corrisponderanno le seguenti query MSL:

```
(QT2.1) Query ::= *O :- <O person {<last_name $LN>}>
(AC2.1)         {printf (lookup_query, 'lookup -ln %s',$LN);}
(QT2.2) Query ::= *O :- <O person {<last_name $LN>
                                   <first_name $FN>}>
(AC2.2)         {printf (lookup_query, 'lookup -ln %s -fn %s ',
                         $LN,$FN);}
```

Ad ogni *query template*, è associata una azione, scritta in questo caso in C, che realizza la traduzione da MSL a linguaggio "nativo". Naturalmente, il sistema sarebbe molto limitato se permettesse di rispondere solamente a questo insieme predefinito di interrogazioni, esiste dunque un modulo che permette di determinare, data una query in input (che può essere ricevuta da un mediatore, come pure direttamente da una applicazione o da un'interfaccia utente), se ad essa la sorgente è in grado di dare risposta. In particolare, oltre alle query predefinite, sono supportate tutte le query q tali che:

- q è equivalente ad una query q' direttamente supportata, ovvero gli insiemi delle risposte delle due query coincidono;
- q è sussunta da q' direttamente supportata, ovvero l'insieme risposta di q è incluso nell'insieme risposta di q'.

6.1.4 Il generatore di Mediatori

In TSIMMIS, il mediatore è il modulo che si preoccupa di dare una visione integrata dei dati, agendo su differenti schemi. Supponiamo di dover integrare due sorgenti, di cui la prima è una base di dati relazionale, *Computer_Science*, il cui schema è riportato di seguito:

```
employee(first_name,last_name,title,report_to)
student(first_name,last_name,year)
```

mentre la seconda è una sorgente ad oggetti, *WHOIS*. Ad ogni sorgente corrisponde, come già mostrato nell'architettura, un wrapper: **CS** esporta le informazioni della prima, **WHOIS** quelle della seconda (esempi di oggetti esportati da questi wrapper sono rispettivamente riportati in Figura 6.2 e in Figura 6.3).

```
<&e1,  employee,  set,         {&f1,&l1,&t1, &rep1}>
       <&f1,      first_name,  string,   'Joe'>
       <&l1,      last_name,   string,   'Chung'>
       <&t1,      title,       string,   'professor'>
       <&rep1,    reports_to,  string,   'John Hennessy'>

<&e2,  employee,  set,         {&f2,&l2,&t2}>
       <&f2,      first_name,  string,   'John'>
       <&l2,      last_name,   string,   'Hennessy'>
       <&t2,      title,       string,   'chairman'>
...............etc.

<&s3,  student,   set,         {&f3,&l3,&y3}>
       <&f3,      first_name,  string,   'Pierre'>
       <&l3,      last_name,   string,   'Huyn'>
       <&y3,      year,        integer,  3>
```

Figura 6.2: Oggetti esportati da **CS** in OEM

```
<&p1,  person,    set,        {&n1, &d1, &rel1, &elem1}>
       <&n1,      name,       string,  'Joe Chung'>
       <&d1,      dept,       string,  'cs'>
       <&rel1,    relation,   string,  'employee'>
       <&elem1,   e_mail,     string,  'chung@cs'>
...............etc.
```

Figura 6.3: Oggetti esportati da **WHOIS** in OEM

Si vuole sviluppare un modulo mediatore, chiamato **MED**, che integri tutte le informazioni inerenti una persona. Per esempio, supponendo che la persona in questione si chiami 'Chung', ed appartenga al dipartimento 'CS' (ovvero Computer Science), la risposta che si vorrebbe ottenere è rappresentata in Figura 6.4.

Per realizzare questa integrazione, TSIMMIS fa uso di un sistema di regole, MSL (Mediator Specification Language), le quali permettono di specificare la struttura dell'oggetto integrato, a partire dalle strutture degli oggetti da recuperare nelle sorgenti. Le regole devono essere quindi esplicitamente definite dall'amministratore del sistema, che è l'unico che deve conoscere lo schema di *persona* delle sorgenti e del mediatore. In particolare, riferendoci all'esempio, la rule che realizza il processo indicato sarà:

```
(MS1) Rule:
```

```
<&cp1,  cs_person,  set,          {&mn1, &mrel1, &t1, &rep1, &elm1}>
        <&mn1,      name,         string,  'Joe Chung'>
        <&mrel1,    relation,     string,  'employee'>
        <&t1,       title,        string,  'professor'>
        <&rep1,     reports_to,   string,  'John Hennesy'>
        <&elem1,    e_mail,       string,  'chung@cs'>
```

Figura 6.4: Oggetti esportati da **MED**

```
<cs_person {<name N> <rel R> Rest1 Rest2}>
           :- <person {<name N> <dept 'cs'> <relation R> | Rest1}>
              @whois
              AND decomp(N, LN, FN)
              AND <R {<first_name FN> <last_name LN> | Rest2}>@cs
External:
decomp(string,string,string)(bound,free,free) impl by name_to_lnfn
decomp(string,string,string)(free,bound,bound) impl by lnfn_to_name.
```

La *testa* della regola `MS1` specifica la struttura che avrà l'oggetto unificato: sarà esportato da **MED** con etichetta `cs_person` con un nome (`name`), un ruolo (`relation`), ed un insieme non specificato di altri attributi, ovvero con tutte le informazioni che sarà possibile recuperare dalle due sorgenti (rispettivamente `Rest1` e `Rest2`). Nella *coda* sono invece definiti i percorsi da seguire per recuperare le informazioni: dalla sorgente **WHOIS** si ricavano oggetti di tipo `person` con un nome definito (lo stesso espresso nell'intestazione della rule), un ruolo, e l'attributo dipartimento uguale a 'cs'; dalla sorgente **CS** sono invece recuperati degli oggetti di tipo `R` (dove `R` è il ruolo specificato nell'intestazione, e può valere '`employee`' o '`student`), e di nome e cognome specificato. In ausilio alla rule vi è inoltre una funzione esterna, `decomp`, che realizza la trasformazione da `name` a `first_name` e `last_name` e viceversa.
In tutto il processo non permane alcuna ambiguità: per prima cosa, sono recuperati dalle sorgenti locali gli oggetti la cui struttura (e le cui etichette) sono conformi alla struttura specificata nella *coda* della rule, poi questi oggetti sono unificati e presentati in modo integrato come specificato nella *testa*.

6.1.5 Il Linguaggio LOREL

Evidenziato il punto debole del tipo di integrazione realizzata nella scarsa flessibilità dovuta alle interrogazioni predefinite dai *query template*, si è cercato di porvi

rimedio sviluppando un linguaggio di interrogazione ad hoc: LOREL (Ligtweight Object REpository Language) [64]. Caratteristica saliente di questo linguaggio, la cui sintassi si ispira ad SQL ed OQL, consiste nel fatto che esso è in grado di sfruttare strutture predefinite, se presenti, ma non ne ha necessariamente bisogno, per fornire risposte significative ad un'interrogazione. Come OEM, è un modello ad "oggetti leggeri", nel senso che presenta un ridotto numero di caratteristiche rispetto ai tradizionali modelli ad oggetti. In questo paragrafo si illustrano solo le caratteristiche salienti di questo linguaggio, rimandandone l'approfondimento alla letteratura.

LOREL si distingue dagli altri linguaggi di interrogazione perchè:

- è in grado di recuperare informazioni anche quando certi dati non sono presenti, attraverso un assegnamento parziale delle tuple o degli oggetti;

- non distingue tra attributi multi-valore e a valore singolo (in SQL attributi multivalore non sono ammessi, mentre in OQL sono distinti da una diversa sintassi);

- opera uniformemente su dati che hanno tipi differenti perchè non esegue controllo dei tipi;

- consente di interrogare sorgenti la cui struttura è parzialmente sconosciuta, utilizzando wildcards in sostituzione dei nomi degli attributi.

Queste caratteristiche sono in varia misura dovute al fatto che, al contrario degli altri linguaggi, LOREL non impone un rigido sistema di tipi, anzi tutti i dati, compresi i valori scalari, sono rappresentati da oggetti (con lo stesso paradigma di OEM, quindi): ogni oggetto ha un identificatore, un'etichetta ed un valore, quest'ultimo può essere tanto uno scalare quanto un set di oggetti innestati nell'oggetto iniziale.

6.1.6 Pregi e difetti di TSIMMIS

Il punto di forza di questo progetto è sicuramente il modello comune OEM adottato che permette l'integrazione tanto di oggetti la cui struttura è fissa e nota o variabile nel tempo, quanto di oggetti con struttura non predefinita (si veda nell'esempio l'uso degli attributi `Rest1` e `Rest2` in Sezione 6.1.4), rendendo possibile ciò che è negato in qualunque ambiente convenzionale orientato agli oggetti che non aderisca ad una semantica di mondo aperto. Queste possibilità rendono l'intero sistema estremamente flessibile, permettendo l'integrazione di sorgenti il cui schema può essere sia parzialmente sconosciuto, sia variabile nel tempo.

Il meccanismo dei *query template* supporta la presenza di sorgenti con diverse capacità di rispondere ad un'interrogazione (è infatti inverosimile pensare che tutte possiedano le funzionalità di un DBMS) e semplifica la fase di interrogazione.

A questo processo può però essere mosso un appunto, infatti esprimendo le capacità di risposta di una fonte di informazioni attraverso query predefinite, è improbabile che si riesca a rappresentare completamente il range di informazioni estraibili dalle sorgenti. Ad esempio può accadere che la fonte non risponda ad una query q ma sia in grado di realizzare una query q' più generale di q. Basterebbe allora, per realizzare q, avere un filtro presso il mediatore, ed eseguire una ulteriore selezione sui dati ricevuti in risposta a q'.

Per quanto riguarda la soluzione proposta con l'introduzione di LOREL, a causa della novità della proposta, un indubbio svantaggio risiede nel fatto che non si possono comunque manipolare le query e gli oggetti facendo affidamento sulle specifiche funzionalità di query processing dei DBMS esistenti, mentre occorre sviluppare moduli dedicati per le fasi di parsing, query rewrite, optimization and execution. Un vantaggio consiste, invece, nel riuscire a superare alcune delle limitazioni di SQL ed OQL, inoltre, rispetto alla soluzione con i query template, viene lasciata all'utente una maggior libertà nella formulazione di interrogazioni (ma indubbiamente anche una maggior complessità).

Un altro problema aperto è rappresentato dal processo di integrazione di informazioni che, di per sé, è ambiguo. In esso non si possono conoscere le sorgenti se non in modo approssimativo, perciò risulta una forzatura eccessiva ipotizzare di lasciare esclusivamente al progettista del sistema il compito di individuare i concetti comuni e di integrarli opportunamente. Questo grosso svantaggio è imputabile anche all'approccio *strutturale* scelto, che non richiede (e quindi non utilizza) gli schemi concettuali delle sorgenti. A questo si contrappone l'approccio *semantico*, nel quale sono sfruttate le informazioni codificate negli schemi al fine di pervenire ad una totale integrazione di tutte le fonti di informazioni.

6.2 GARLIC

L'obiettivo del progetto **GARLIC** [65, 66] (sviluppato presso il centro di ricerca IBM di Almaden) è la costruzione di un Multimedia Information System (MMIS) in grado di integrare dati che risiedono in differenti basi di dati, nonché in un insieme di server di diversa natura, mantenendo la reciproca indipendenza dei server ed evitando la replicazione fisica dei dati. In questo contesto, il termine*Multimediale* deve essere interpretato in senso molto ampio, indicando non solo immagini, video, e audio ma anche testi e tipi di dati specifici di alcune applicazioni (disegni CAD, mappe, . . .). Poiché molte di queste informazioni sono già modellate tramite oggetti, GARLIC fornisce un modello orientato ad oggetti che permette a

tutte le differenti sorgenti di descriversi in modo uniforme. A questo modello si aggiunge, inoltre, un linguaggio di interrogazione, anch'esso orientato ad oggetti (ottenuto con un'estensione al linguaggio SQL) e utilizzato dal livello intermedio dell'architettura GARLIC, i cui compiti sono:

- presentare lo schema globale alle applicazioni,
- interpretare le interrogazioni,
- creare piani di esecuzione per le interrogazioni e
- riassemblare i risultati in un'unica risposta omogenea.

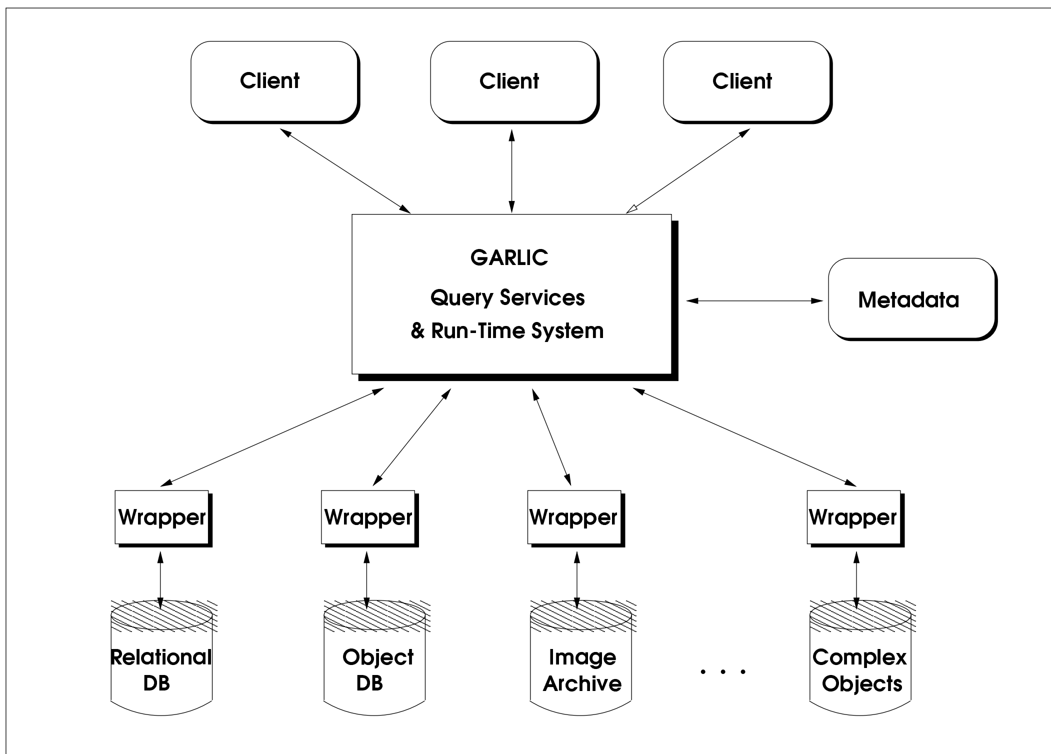

Figura 6.5: Architettura GARLIC

La Figura 6.5 rappresenta l'architettura di GARLIC. Alla base della figura stanno le sorgenti di informazioni che devono essere integrate (tra le quali basi di dati relazionali, ad oggetti, file system, document manager, image manager, ...). Sopra ogni sorgente è posizionato un wrapper, che traduce le informazioni sugli

schemi, sugli accessi ai dati e sulle interrogazioni, dal protocollo interno di GARLIC ai protocolli nativi delle sorgenti. Le informazioni riguardanti lo schema unificato sono mantenute nel deposito di metadati. L'altra sorgente di informazioni alla base della figura (Complex objects) serve invece per memorizzare gli *oggetti complessi* di GARLIC, utilizzati dalle applicazioni per unire i dati originariamente separati (siano essi appartenenti a schemi distinti, o allo stesso schema). I servizi di query processing sono forniti dal componente *Query Services & Run-Time System*: ad esso spetta il compito di presentare alle applicazioni una visione unificata, ad oggetti, del contenuto del sistema e di gestirne le richieste (interrogazioni o modifiche). Lo schema globale è presentato all'utente, e alle applicazioni, attraverso il modello di dati di GARLIC (**G**arlic **D**ata **M**odel): è costituito dalla **unione** degli schemi locali. Fra questi è pure disponibile lo schema degli *oggetti complessi*, creati ad hoc dalle applicazioni per avere una visione integrata di oggetti preesistenti. Il fornire una descrizione dei dati attraverso il GDL (**G**arlic **D**ata **L**anguage) permette inoltre di identificare i dati che si vogliono integrare e parallelamente escludere dalla descrizione quelli che non si vogliono rendere accessibili dall'esterno.

In sostanza GARLIC presenta all'utente ed alle loro applicazioni i benefici dei database caratterizzati da un preciso schema (simile a quello offerto dai DBMS ad oggetti o object-relational) ma senza replicazione fisica. Internamente le differenze da un DBMS tradizionale sono invece più evidenti, infatti il sistema riunisce informazioni provenienti da più sorgenti, utilizzando, per fonderle, due strumenti:

- il modello dei dati orientato agli oggetti
- la memorizzazione degli oggetti complessi[1].

6.2.1 Il linguaggio GDL

Tra i vari compiti dei wrapper vi è quello di fornire una descrizione del contenuto della sorgente da loro servita, utilizzando il **G**arlic **D**ata **L**anguage, o **GDL**. GDL è un variante dell'ODMG[2] **O**bject **D**escription **L**anguage [67]: attraverso le interface, ed un forte sistema dei tipi, si possono descrivere gli oggetti ed il loro comportamento, e memorizzare la loro descrizione in un *repository schema*. I vari *repository* sono quindi registrati come parti di un GARLIC Database, e *fusi* nello schema globale presentato all'utente.

[1]Gli Oggetti Complessi servono nell'integrazione dei dati multimediali con quelli tradizionali, per aggiungere metodi che implementino nuovi comportamenti realizzabili grazie alla visione completa delle informazioni.

[2]Le differenze dallo standard ODL sono dovute alla necessità di esprimere situazioni non presenti in ambiente centralizzato.

<table>
<tr>
<td rowspan="2">

Relational Repository Schema

```
interface Country {
   attribute string name;
   attribute string airlines_served;
   attribute boolean visa_required;
   attribute Image scene;
   }
interface City {
   attribute string name;
   attribute long population;
   attribute boolean airport;
   attribute Country country;
   attribute Image scene;
   }
```

</td>
<td>

Web Repository Schema

```
interface Hotel {
   attribute readonly string name;
   attribute readonly short class;
   attribute readonly doube daily_rate;
   attribute readonly string location;
   attribute readonly string city;
   }
```

</td>
</tr>
<tr>
<td>

Image Server Repository Schema

```
interface Image {
   attribute readonly string file_name;
   double matches (in string file_name);
   void display (in string device_name);
   }
```

</td>
</tr>
</table>

Figura 6.6: GDL schema

Un esempio di GDL è riportato in Figura 6.6, in cui viene considerata una semplice applicazione per una agenzia di viaggio. L'agenzia gestisce informazioni sugli stati e sulle città per le quali organizza viaggi (in un db relazionale), nonché un sito web per la prenotazione di hotels, ed un image server che raccoglie immagini pubblicitarie. La base di dati relazionali ha due sole tabelle: `Country` e `City`. La tabella `Country` mantiene le informazioni sugli stati, ed ha come chiave l'attributo `name`. La tabella `City` invece possiede sia una chiave, `name`, sia una foreign key, `country`. È interessante vedere come sia compito del wrapper individuare le foreign key nello schema originale (in questo caso `country`), e riportarle in GDL come se fossero attributi con dominio complesso (il dominio di `country` diventa così la tabella `Country`), facendone quindi una traduzione da relazionale a visione orientata agli oggetti (e questo è sicuramente un punto a favore di GARLIC). Più improbabile è invece la soluzione adottata per l'attributo `Scene`, che viene messo in relazione con la classe `Image`, ipotizzando dunque di sapere a priori che questo attributo si riferisce ad una tabella di un altro sistema (non dovrebbe comunque essere tra i compiti del wrapper il provvedere al *linking* di classi appartenenti a schemi di sorgenti diverse[3]).
A supporto di questa visione, viene la soluzione adottata nella descrizione della sorgente Web per le prenotazioni di hotels. L'attributo `city`, che andrebbe logicamente collegato alla tabella `City`, in realtà insiste su un dominio di tipo stringa, questo perché, nell'esempio, si suppone che il sito web sia al di fuori del diretto

[3]il wrapper dovrebbe infatti occuparsi, ed essere a conoscenza, solo delle informazioni strettamente relative alla sorgente che gestisce, mentre dovrebbe essere un modulo di livello superiore a provvedere all'integrazione.

controllo dell'agenzia, a differerenza della base di dati relazionale e dell'image server. L'eventuale integrazione del sito con le altre sorgenti è quindi (giustamente) demandata alla creazione di un *oggetto complesso*, ad un livello superiore.
In particolare, in GARLIC, con *oggetto complesso* si definisce una *vista* il cui obiettivo è arricchire (estendendo, semplificando o deformando) uno o più oggetti appartenenti a schemi locali. Questi oggetti complessi sono definiti in modo dichiarativo (allo stesso modo con cui in SQL è possibile definire una vista basata su altre tabelle), utilizzando il linguaggio di interrogazione interno di GARLIC (si tratta di un'estensione orientata agli oggetti dello **S**tandard **Q**uery **L**anguage) e sono visti dall'utente, e dalle applicazioni, come oggetti veri e propri.
Interessante è anche l'uso della parola chiave `readonly`, che permette di discriminare tra fonti aggiornabili e fonti da cui si può esclusivamente estrarre dati, anche se poi non viene spiegato alcun meccanismo di propagazione degli update.

6.2.2 Query Planning

La fase di query planning porta, da una interrogazione posta sullo schema unificato, alla definizione di un insieme di query che le sorgenti locali devono eseguire. I Wrapper hanno una parte attiva in questo processo, infatti le loro conoscenze sono utilizzate per formulare differenti piani di accesso, e per determinare il più efficiente tra questi. Durante la fase di pianificazione della query l'ottimizzatore di GARLIC identifica il frammento maggiore possibile che coinvolge una particolare sorgente, e lo spedisce al corrispondente wrapper, il quale determina zero o più piani di accesso che realizzino, in toto o in parte, la query a lui assegnata. A questo punto l'ottimizzatore memorizza tutti i piani di tutti i wrapper interrogati, ed, eventualmente, aggiunge le operazioni da effettuare nel caso in cui una parte della query originale non sia eseguibile da alcun wrapper. È, infatti, da sottolineare come GARLIC sia in grado di gestire pure sorgenti con particolari restrizioni, ad esempio, oltre ad ipotizzare che una sorgente non sia in grado di effettuare join a più vie, il wrapper può essere a conoscenza di particolarissime limitazioni sulle operazioni realizzabili, come possono essere la lunghezza massima di una stringa, il valore massimo di una costante in una interrogazione,ecc . . . Il vantaggio consiste nel fatto di non dover comunicare tutte le restrizioni all'ottimizzatore, bensì di incapsularle a livello di wrapper. A sua volta l'ottimizzatore, in grado di realizzare funzioni tipiche di un DBMS, potrà effettuare quelle operazioni scartate dalle sorgenti.
Oltre a questo, la possibilità di limitare le funzioni di risposta di una sorgente agevola notevolmente la fase di sviluppo di un wrapper, infatti in un primo momento si possono realizzare solo le funzioni più semplici (rendendo utilizzabili da subito le sue informazioni), rimandando ad un secondo momento lo sviluppo di funzioni di ricerca più avanzate.

6.2.3 Pregi e difetti di GARLIC

Nonostante GARLIC sia da considerare la versione commerciale di TSIMMIS (sono entrambi stati sviluppati in ambienti IBM), l'intera impostazione del progetto è stata modificata. La differenza maggiore è senza dubbio l'abbandono del modello OEM (vedi Sezione 6.1.1), a cui è stato preferito l'utilizzo degli schemi locali (descritti attraverso il modello GDL). A fronte di una perdita di flessibilità dell'intero sistema (è ora praticamente impossibile gestire anche sorgenti semi-strutturate), l'intera architettura risulta semplificata, così come pure è semplificata la fase di integrazione degli schemi.
Purtroppo però non sono stati fatti passi avanti nella automazione della fase di integrazione, quindi l'utente, o l'applicazione, deve definirsi una visione ad-hoc di tutti gli schemi, o deve considerarne semplicemente l'unione (con tutte le duplicazioni di informazioni che ne conseguono).

Molto approfondita è invece la fase di query planning, in senso di tradizionali operazioni di ottimizzazione dei costi di accesso ai DB, che non si è potuto descrivere meglio in questi paragrafi per motivi di spazio, ma a cui si rimanda negli articoli in bibliografia [65, 66].
Sostanzialmente diverse da progetti analoghi sono comunque le funzioni del wrapper: da semplice traduttore di linguaggi e protocolli, in GARLIC il wrapper include molte delle funzioni demandate in altri sistemi al mediatore vero e proprio. Se dal punto di vista architetturale questo potrebbe anche essere considerato un errore, risulta sorprendente (e forse poco credibile) la sostanziale differenza dei tempi di sviluppo dichiarata da TSIMMIS e da GARLIC. Mentre in TSIMMIS si considera ragionevole impiegare circa 6 mesi per realizzare un semplice wrapper, nel progetto GARLIC può essere sviluppato in poche settimane . . .

6.3 SIMS

SIMS [68, 69], sviluppato presso l'Università della Southern California, è un mediatore di informazioni che si occupa di fornire accesso e integrazione ad una molteplicità di sorgenti eterogenee. Il cuore del progetto, ed il suo punto di forza, è la sua dichiarata abilità nel ritrovare e trattare le informazioni in modo *intelligente*, ovvero utilizzando tecniche di intelligenza artificiale. SIMS si distingue infatti dai progetti precedentemente esposti, e ne migliora l'approccio all'interrogazione, per la sua abilità nell'ottimizzare la fase di query processing, essendo in grado di manipolare, attraverso tecniche di intelligenza artificiale, le descrizioni semantiche delle sorgenti.
Rimane invece manuale la fase di integrazione delle informazioni (ovvero la costruzione dello schema globale a partire da quelli locali), nonostante sia stata

automatizzata, attraverso il modulo LIM, la traduzione di tutti gli schemi locali dal modello originale al modello di conoscenza di SIMS, che si basa sulla logica descrittiva LOOM.

Il sistema SIMS segue le seguenti idee di fondo:

- *Rappresentazione e Modellazione della conoscenza*: è usata per descrivere il dominio che accomuna le informazioni, come pure le strutture ed il contenuto delle sorgenti di informazioni stesse. Il modello di ogni sorgente deve indicare il modello dei dati da questa utilizzato, il linguaggio di interrogazione, la dimensione, e deve descrivere il contenuto di tutti i suoi campi usando la terminologia di un predefinito modello comune del dominio (denominato *domain model*, e che costituisce in pratica lo schema globale);

- *Pianificazione e Ricerca*: è utilizzata per costruire una sequenza di query dirette alle singole sorgenti, a partire dalla interrogazione dell'utente;

- *Riformulazione*: dopo aver determinato un insieme di piani di accesso alle sorgenti, SIMS identifica, applicando un modello dei costi ed un algoritmo di ottimizzazione semantica, il più efficiente tra questi. Questa ricerca del piano migliore è aiutata anche dal fatto di avere a disposizione gli schemi descrittivi, semanticamente ricchi, sia delle singole sorgenti, sia del modello globale.

Componenti di SIMS

Per adempiere a tutte le sue funzioni, e per utilizzare tecniche *intelligenti*, SIMS fa uso di una serie di strumenti, caratteristici dei sistemi KBMS:

1. **LOOM**: è il sistema di rappresentazione della conoscenza utilizzato per descrivere sia le fonti locali di informazioni, sia lo schema globale, sia la fonte di informazione rappresentata dalle risposte alle query già ottenute, e memorizzate, da una determinata applicazione. Si tratta di una logica descrittiva derivato dal KL-ONE (modello sviluppato da Brachman nel 1976 in [26]).

2. **LIM**: è un'interfaccia (**L**OOM **I**nterface **M**odule) per mediare tra LOOM e le basi di dati. In particolare provvede a tradurre le descrizioni degli schemi delle sorgenti in linguaggio LOOM, nonché a tradurre query dirette alle sorgenti da LOOM al linguaggio di interrogazione proprio della singola sorgente. Deve essere sviluppata ad-hoc per ogni interfaccia che andrà a servire, ma automatizza la fase di wrapping degli schemi locali.

3. **Prodigy**: È il modulo che risolve i problemi di selezione delle sorgenti e pianificazione delle interrogazioni: partendo da una query sullo schema globale, utilizzando una serie di operatori da applicare alla query ed alla conoscenza memorizzata, ottiene uno stato finale caratterizzato dalle risposte che soddisfano la query.

6.3.1 Integrazione delle sorgenti

Poichè SIMS fa uso di un approccio "semantico", è assolutamente necessario che possieda le descrizioni dettagliate di tutte le fonti informative, creandone un *modello*. Per ogni singola sorgente, il *modello* deve contenere le seguenti informazioni:

- descrivere il *contenuto* informativo della sorgente;
- specificare se si tratta di una sorgente "classica"(nel qual caso si dovrà utilizzare l'interfaccia LIM) o di una sorgente di conoscenza LOOM;
- descrivere le dimensioni della base di dati e delle sue tabelle, nonché la loro locazione, per poter stimare il costo di un determinato piano di accesso;
- definire le chiavi delle tabelle, se esistono.

In aggiunta ai modelli delle sorgenti, viene definito un modello del dominio applicativo, definito *domain model*, che costituirà lo schema globale, l'unico col quale l'utente dovrà interagire. Questo è costituito da una base di conoscenza terminologica organizzata in modo gerarchico (attraverso il linguaggio LOOM), dove i nodi rappresentano tutti gli oggetti, le azioni, gli stati, possibili all'interno del dominio.

Le entità del dominio non devono però necessariamente corrispondere a classi appartenenti ad una determinata sorgente, il *domain model*, dunque, deve essere inteso come la descrizione del dominio applicativo dal punto di vista dell'utente, e solo con esso l'utente avrà a che fare. In particolare, per porre una query,

```
(retrieve (?depth)
          (:and (port ?port)
                (port.name ?port ''SAN-DIEGO'')
                (port.depth ?port ?depth)
```

Figura 6.7: Esempio di query SIMS

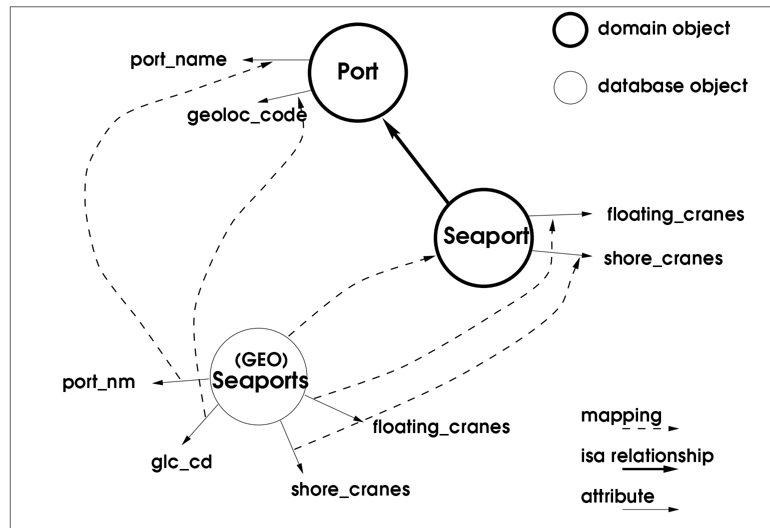

Figura 6.8: Mapping tra *domain model* e modello locale

l'utente compone uno statement LOOM (un esempio di interrogazione è riportato in Figura 6.7, che richiede la profondità del porto di SAN-DIEGO) usando solo la terminologia del *domain model*, essendo in questo modo esonerato dal dover conoscere tutte le sorgenti integrate nel sistema (benché possa pure interagire direttamente con alcune di esse, se ha particolare familiarità con i loro termini).

La parte critica dell'intero processo è invece la fase di integrazione delle sorgenti, ovvero il collegare lo schema globale alle varie sorgenti locali. Questo consiste nel descrivere tutti i concetti e le relazioni di una sorgente attraverso i termini del *domain model*, tale mapping deve essere fatto per tutte le sorgenti, ed in modo particolarmente accurato. Un esempio qualitativo di questo mapping è riportato in Figura 6.8: se l'utente richiede tutti gli elementi della classe globale `Seaport`, andrà interrogata la classe `Seaports` della sorgente `GEO`.

6.3.2 Query Processing

L'intero processo che porta dalla formulazione di una query da parte dell'utente, posta sullo schema globale, alla presentazione dei risultati, è rappresentato in Figura 6.9.

Selezione delle fonti informative Il primo passaggio da realizzare, una volta in possesso dell'interrogazione dell'utente formulata usando la terminologia dello schema globale, consiste nell'identificare le appropriate sorgenti che saranno interessate dalla query. Per esempio, se l'utente richiede informazioni sui `porti`

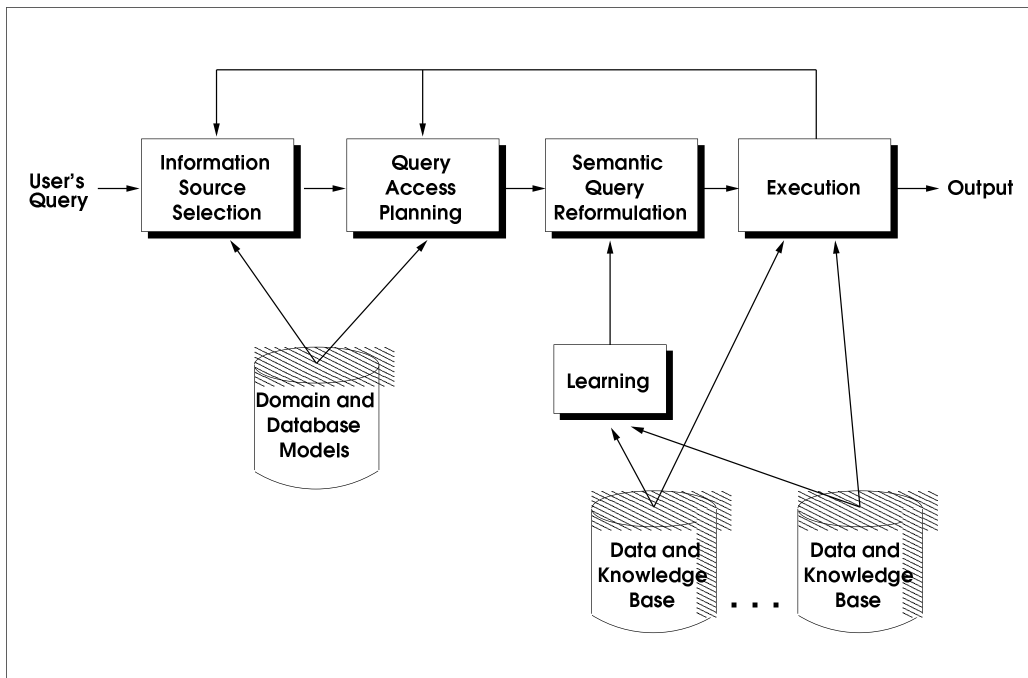

Figura 6.9: Query Processing

ed esiste in una base di dati un concetto che contiene i porti, il mapping è facilissimo, come pure la riformulazione della query. In realtà, molto spesso non esisterà un mapping diretto e sarà necessario "$riformulare$" la query utilizzando termini propri delle sorgenti locali. A questo scopo, vengono utilizzati una serie di operatori di riformulazione, diretti a definire una query equivalente alla originale. Tra di essi, operatori di generalizzazione (che fanno uso delle relazioni tra classe e super-classe e spostano ad un livello superiore dell'albero gerarchico la query, magari aggiungendovi delle limitazioni ai domini degli attributi in modo da ottenere ancora una query equivalente), operatori di specializzazione (che attraverso tecniche di intelligenza artificiale cercano di riclassificare la query ad un livello gerarchico inferiore, in modo da determinare l'insieme di classi da interrogare), operatori di partizione.

Piano di accesso Il piano di accesso determina un ordine per le query trovate nella fase precedente, cercando di massimizzare il parallelismo delle operazioni. Per fare questo vengono analizzati i costi di accesso alle sorgenti ed i costi di eventuali passaggi aggiuntivi che il sistema dovrà realizzare per produrre i risultati finali. L'ordine di esecuzione è determinato esaminando quali passi del

piano di accesso si basano su risultati raccolti in altre sorgenti, facendo uso del modulo Prodigy, che per questa attività presenta funzioni analoghe a quelle di un ottimizzatore DBMS.

Query Plan Reformulation Oltre ad una ottimizzazione basata sul modello dei costi di accesso, SIMS è pure in grado di realizzare una query reformulation di tipo semantico. L'idea di base è trasformare la query risultante dal piano di accesso in una query semanticamente equivalente che può essere eseguita in maniera più efficiente. Questa ottimizzazione è realizzata sia a livello di singola sorgente, sia a livello globale. Nella singola sorgente, il problema della riformulazione è analogo al problema dell'ottimizzazione semantica di una query all'interno di una base di dati. La riformulazione si basa quindi su un processo di inferenza che utilizza delle informazioni estratte dal database e codificate attraverso l'uso di *rule* e di limitazioni sui range dei domini. Analogamente viene effettuata una ottimizzazione della query globale, in modo da realizzare efficientemente la fase di processing dei risultati parziali ottenuti dalle singole sorgenti.

6.3.3 Pregi e difetti di SIMS

Non sono pochi gli aspetti interessanti ed innovativi che caratterizzano questo progetto, dovuti fondamentalmente al massiccio uso di tecniche di Intelligenza Artificiale, permettendo di sfruttare tutti i pregi di un approccio semantico. Queste tecniche sono utilizzate sia in fase di identificazione delle sorgenti (in assenza di mapping diretti si determinano a run time le sorgenti che sono interessate dalla query), sia in fase di ottimizzazione della query stessa.
Altro aspetto da sottolineare, assente in progetti analoghi, è l'utilizzo di una base di dati (o meglio, di conoscenza) interna al mediatore stesso sia per riprocessare le risposte delle sorgenti (ma questo è un aspetto già incontrato), sia per memorizzare le risposte già ottenute, utilizzandole successivamente, in parte o in toto, come sorgente aggiuntiva (evitando in questo modo di andare a recuperare dati che sono presenti nella memoria del sistema).

Rimane intentata, come negli altri progetti, la automazione della fase di integrazione degli schemi locali, realizzata dal nostro sistema, mentre risulta possibile usufruire della conoscenza semantica degli schemi sorgenti, cosa che negli altri approcci è stata trascurata.

6.4 Osservazioni

È stato scelto di illustrare questi tre sistemi perchè presentano approcci complementari al vasto problema dell'integrazione delle informazioni:

- TSIMMIS realizza l'integrazione di sorgenti semistrutturate, trascurando necessariamente la possibilità di ottimizzazione delle fasi di query processing possibili usufruendo dei mapping fra gli schemi;
- GARLIC, viceversa, mostra un approccio più orientato ad una visione DBMS e commerciale, però trascura di trattare aspetti più profondi impliciti nelle ontologie delle sorgenti autonome;
- SIMS coglie l'importanza della conoscenza semantica delle sorgenti e la utilizza per completare la fase di integrazione nel processo di ottimizzazione delle query, però non supporta la fase di integrazione degli schemi, né estende la possibilità di esprimere la conoscenza inter sorgenti, oltre che fra le sorgenti e lo schema globale.

Il sistema MOMIS coglie molti dei principi ed aspetti presenti in SIMS, cercando però di potenziare le fasi sia di generazione dello schema globale, sia di query processing. L'integrazione viene, infatti, eseguita sfruttando componenti intelligenti per la realizzazione di un modulo che, in modo semi-automatico, guidi il progettista nell'integrazione degli schemi locali. Il query processing viene esteso in modo da prevedere la gestione delle conoscenze estensionali, al fine di migliorare la risposta (sia in termini di qualità, sia di costo).

Conclusioni

Il progetto MOMIS si colloca nell'ambito dell'Integrazione di Informazioni, ed ha come obiettivo lo sviluppo di un sistema in grado di fornire un accesso integrato e facilitato ad un insieme di sorgenti eterogenee distribuite.
Ciò che contraddistingue MOMIS, rispetto altri sistemi realizzati nello stesso ambito, sono il tipo di approccio seguito e l'introduzione di comportamenti intelligenti. L'approccio, che possiamo definire semantico-virtuale, si basa sull'analisi degli schemi forniti dalle sorgenti per costruire uno schema integrato che presenti, all'utente, le informazioni locali in modo omogeneo. La rappresentazione ottenuta costituisce una vista virtuale, pertanto non contiene alcun dato e sarà necessario prevedere una fase di integrazione che fornisca tutte le informazioni necessarie, al sistema, per reperire i dati dalle sorgenti e combinarli in modo corretto. Il risultato di questa fase consiste, quindi, nella produzione di un insieme di conoscenze intensionali ed estensionali; le prime permettono la risoluzione dei conflitti tra schemi, mentre le seconde indicano il modo in cui le entità del dominio applicativo sono rappresentate sull'insieme di sorgenti.
Il punto di forza di MOMIS consiste nel realizzare questa integrazione mediante un processo semi-automatico, che vede coinvolti il sistema stesso ed il progettista, permettendo non solo l'unificazione degli schemi delle sorgenti, ma anche la creazione di uno schema globale vero e proprio, direttamente interrogabile dall'utente.

Un aspetto sicuramente innovativo, è poi costituito dalla presenza di componenti intelligenti che portano ad un miglioramento del sistema sia nella fase di integrazione, sia in quella di gestione delle interrogazioni. Nel primo caso, infatti, questi elementi possono essere impiegate per verificare le indicazioni fornite dal progettista e per estenderle mediante nuove relazioni inferite in modo automatico, mentre, nel secondo, può essere sfruttata l'espansione semantica e la sussunzione per riformulare le interrogazioni, generandone altre equivalenti ma più efficienti.

In questa tesi, sono state analizzate e approfondite le caratteristiche del sistema, allo scopo di individuare le fasi che compongono il query processing. L'obiet-

tivo è stato, dunque, quello di progettare ed implementare il componente Query Manager, in modo che fosse in grado di individuare le sorgenti coinvolte in un'interrogazione, per poi recuperare ed integrare i dati in esse contenute. In particolare sono state formalizzate le informazioni generate nella fase di integrazione, per capire come potessero essere usate al fine di ottenere una risposta corretta, completa e minima.
I moduli software implementati in questa tesi individuano le informazioni che devono essere gestite dal Query Manager per l'esecuzione delle interrogazioni ed implementano le funzionalità necessarie all'acquisizione, decomposizione e trasformazione delle interrogazioni.

I prossimi sviluppi del progetto dovrebbero essere volti al completamento sia della fase di integrazione, sia di quella di gestione delle interrogazioni. L'integrazione degli schemi deve essere dotata dei componenti necessari all'individuazione e generazione delle conoscenze estensionali, inoltre è opportuno prevedere tool grafici, di ausilio al progettista nella generazione dello schema integrato. Per quanto riguarda il query processing occorre, invece, introdurre le funzionalità necessarie alla generazione del piano di esecuzione associato alle query, cioè per l'individuazione delle basic query che la compongono e delle operazioni da eseguire sui risultati ottenuti.

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'Universitá di Stanford, nel 1995. Il glossario é 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 é 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 puó 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, proprietá, 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 capacitá, sui costi . . .

A.2 Servizi

- Servizio = funzionalitá fornita da uno o piú 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 molteplicitá di risorse, spesso eteorgenee.
- accoppiamento temporale = servizio di mediazione per riconoscere e risolvere differenze nelle unitá di misura temporali utilizzate dalle risorse.

- accoppiamento spaziale = servizio di mediazione per riconoscere e risolvere differenze nelle unitá 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 velocitá 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.
- pubblicitá (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) = abilitá 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 qualitá, rilevanza, costo.
- proprietario = individuo o organizzazione che ha creato, o ha i diritti di un oggetto, e lo puó sfruttare.
- proprietario di un servizio = individuo o organizazzione responsabile di un servizio.
- database = risorsa che comprende un insieme di dati con uno schema descrittivo.

- warehouse = database che contiene o dá accesso a dati selezionati, astratti e integrati da una molteplicitá 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.
- interoperabilitá = capacitá di interoperare.
- eterogeneitá = incompatibilitá 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, unitá 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 molteplicitá 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, ereditarietá, 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 molteplicitá 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 conformitá ad una ontologia.
- algebra dell'ontologia = insieme delle operazioni per definire relazioni tra ontologie.
- consistenza temporale = é raggiunta se tutti i dati si riferiscono alla stessa istanza temporale ed utilizzano la stessa granularitá temporale.
- specifico ad un dominio = relativo ad un singolo dominio, presuppone l'assenza di incompatibilitá semantiche.
- indipendente dal dominio = software, strumento o conoscenza globale applicabile ad una molteplicitá di domini.

Appendice B

Esempio in ODL$_{I^3}$

Di seguito é riportata la descrizione, attraverso il linguaggio ODL$_{I^3}$, dell'esempio di riferimento. Si introducono inoltre le rule estensionali relative all' insieme di classi che formano il cluster University_Person .

```
UNIVERSITY source:
interface University_Worker
( source relational University
  extent University_Worker
  key first_name, last_name
  foreign_key dept_code )
{ attribute string first_name;
  attribute string last_name;
  attribute integer dept_code;

  attribute integer pay; };

interface Research_Staff
( source relational University
  extent Research_Staffers
  keys first_name, last_name
  foreign_key dept_code, section_code )
{ attribute string first_name;
  attribute string last_name;
  attribute string relation;
  attribute string e_mail;
  attribute integer dept_code;
  attribute integer section_code;
  attribute integer pay; };

interface School_Member
( source relational Universit
  extent School_Members
  keys first_name, last_name
{ attribute string  first_nam
  attribute string  last_name
  attribute string  faculty;
  attribute integer year; }
```

```
interface Department
( source relational University
  extent Departments
  key dept_code )
{ attribute string dept_name;
  attribute integer dept_code;
  attribute integer budget;
  attribute string dept_area; };

interface Section
( source relational University
  extent Sections
  key section_code
  foreign_key room_code )
{ attribute string section_name;
  attribute integer section_code;
  attribute integer length;
  attribute integer room_code; };

interface Room
( source relational University
  extent Room
  key room_code )
{ attribute integer room_code;
  attribute integer seats_number;
  attribute string notes; };
```

COMPUTER_SCIENCE source:

```
interface CS_Person
( source object Computer_Science
  extent CS_Persons
  key name )
{ attribute string name; };

interface Professor : CS_Person
( source object Computer_Science
  extent Professors )
{ attribute string title;
  attribute Division belongs_to;
  attribute string rank; };

interface Student : CS_Person
( source object Computer_Science
  extent Students )
{ attribute integer year;
  attribute set<Course> takes;
  attribute string rank; };

interface Division
( source object Computer_Science
  extent Divisions
  key description )
{ attribute string description;
  attribute Location address;
  attribute integer fund;
  attribute integer employee_nr;
  attribute string sector; };

interface Location
( source object Computer_Science
  extent Locations

interface Course
( source object Computer_Science
  extent Courses
```

```
  keys city, street, county, number)
{ attribute string city;
  attribute string street;
  attribute string county;
  attribute integer number; };
```

```
  key course_name )
{ attribute string course_name;
  attribute Professor taught_by;
```

```
Tax_Position source:

interface University_Student
( source file Tax_Position
  extent University_Students
  key student_code )
{ attribute string name;
  attribute integer student_code;
  attribute string faculty_name;
  attribute integer tax_fee; };
```

Le rule estensionali sul cluster University_Person sono:

```
rule RE1a forall x in School_Member then x in University_Student;

rule RE1b forall x in University_Student then x in School_Member;

rule RE2 forall x in Research_Staff then x in University_Worker;

rule RE3 forall x in Student then x in School_Member;

rule RE4 forall x in Professor then x in Research_Staff;

rule RE5 forall x in (Professor and School_Member)
                                     then x in bottom;

rule RE6 forall x in (Research_Staff and University_Student)
                                     then x in bottom;

rule RE7 forall x in (Research_Staff and Student)
                                     then x in bottom;
```

Appendice C

Esempio di produzione di documentazione

In questa sezione viene riportato il codice relativo alla classe **LocalQuery** allo scopo di mostrare la documentazione prodotta dal componente Javadoc, a partire da commenti realizzati rispettando un opportuno formalismo. Il comando da usare per produrre la documentazione è:

```
mkdir doc
javadoc -private -d doc `find . -name '*.java'`
```

Il codice della classe è il seguente:

```
package queryman;
import java.util.Vector;
import java.util.Iterator;
import oql.*;
import globalschema.*;

/** e` la specializzazione della classe <i>Query</i> che rappresenta
 *  interrogazioni locali, cioe` quelle eseguite sulle sorgenti*/
public class LocalQuery extends Query implements Cloneable
{
  //
  // PROPERTIES
  //
  /** indica il nome della sorgente a cui deve essere inviata la quer
  public String sourceName;
  //
  // CONSTRUCTORS
```

```
//
/** genera un'istanza della classe completamente vuota */
public LocalQuery()
  {
    super();
    sourceName = "";
  }
/** genera un'istanza della classe inizializzando la <i>fromClause</i> del
*  campo <i>struct</i>.
*  @param classN e` una stringa contenente il nome della classe locale
*         coinvolta nella query
*  @param iter e` una stringa che specifica l'iteratore usato
*  @param dis indica la presenza della clausola distinct
*/
public LocalQuery(String classN, String iter, boolean dis)
  {
    super();
    Vector vec = new Vector(0);
    FromElement fr;
    Oql_Identifier cl = new Oql_Identifier(classN);
    Oql_Identifier it = new Oql_Identifier(iter);
    fr = new FromElement(cl,it,true);
    vec.add(fr);
    //struct =new Oql_SelectExpr(new Vector(0),vec,dis);
  }

/** genera una query locale a partire da una basic query e dalle
 *  informazioni di mapping contenute nello schema globale.<br>
 *
 *  @param locClassName indica il nome della classe locale a cui e`
 *         rivolta la query
 *  @param join e` un vettore contenente l'insieme dei nomi degli
 *         attributiche devono essere aggiunti a quelli di
 *         proiezione per consentire la ricomposizione della
 *         risposta
 *  @param maTa e` la mapping table contenente le informazioni
 *         relative alla classe locale a cui e` rivolta
 *         l'interrogazione
 *  @param queryStr e` la struttura della query che deve essere
 *         trasformata
 *  @param itName indica il nome dell'iteratore usato nella basic Query
```

```
 *  @exception -Exception errori generati durante la trasformazione
 *              della query
 */
public LocalQuery(String localClassName, Vector join,
     MappingTable maTa, Oql_SelectExpr queryStr, String itName)
throws Exception
  {
    Oql_SimpleQuery query = new Oql_SimpleQuery(itName);
                         // e` la struttura dati in
                         // cui viene costruita
                         // la query locale
    Oql_SimpleQuery q = null;
    UnionQuery uq = null;
    String s = "";
    int i = 0;
    LocalClass lc = null;
    FromElement froEle = null;
    TransOutput querySet = null;
    TransEle queryEle = null;
    Oql_Identifier iden1 = null, iden2 = null;
    // viene inizializzato il campo relativo al nome della sorgente
    //
    if( (lc = (LocalClass)(maTa.localClasses.get(localClassName)))
      throw new Exception("Translation Error: nome di classe
                           locale non presente");
    sourceName = lc.getSourceName();
    System.out.println("LQ : source name: " + sourceName);
    //
    // viene inizializzata la clausola from della Simple query
    //
    iden1 = new Oql_Identifier(lc.getClassName());
    iden2 = new Oql_Identifier(lc.toIterator());
    froEle = new FromElement((Oql_Query)iden1,iden2,true);
    System.out.println("LQ : target class : " + iden1.toString());
    System.out.println("LQ :  class iterator : " + iden2.toString()
    query.addClass(froEle);
    querySet = queryStr.translateQuery(localClassName,maTa,query);
    //
    // querySet contiene ora l'insieme di query che compongono
    // la query locale. Si ottera` quindi
    // un insieme di oggetti di tipo "Oql_SipmleQuery" dai quali e`
```

```
        // possibile ricavare le UnionQuery da inserire nel campo
        // "subQueries"
        //
        System.out.println("LQ :  abbiamo tradotto");
        for( i=0; i<querySet.elements.size(); i++)
          {
            queryEle = (TransEle)querySet.elements.get(i);
            uq = new UnionQuery();
            q = (Oql_SimpleQuery)queryEle.mappingField.get(0);
            completeQuery(q,join);
            if(q.hasProjection())
              {
                uq.text = q.toString();
                uq.plan = q.plan;
                s = "LQ" + (i+1);
                subQueries.put(s,uq);
              }
          }
        //
        // si inizializza null il campo plan
        //
        plan = null;
        System.out.println("LQ : usciamo dalla lq");
      }
    //
    // METHODS
    //
    /** stampa la rappresentazione del piano */
    public String printPlan()
      {
        return "";
      }
    /** routine per l'esecuzione della query. <br> */
    public Object executeQuery()
      {
        Object obj = null;
        return obj;
      }

    /** ritorna in una stringa il testo della query.*/
    public String toString()
```

```
    {
      Strig s = "";
      UnionQuery uqu = null;
      Iterator it = subQueries.keySet().iterator();
      while(it.hasNext())
        {
          uqu = subQueries.get(it);
          s = s + uqu.toString() + "\n \n";
        }
      return s;
    }
    }

  /** questa routine aggiunge agli attributi di proiezione della quer
   *  passata quelly inidcati nel campo <i>join</i>.<br>
   *  Il metodo serve per completare le query ottenute dal processo
   *  di trasformazione inserendo gli attributi necessari per fare
   *  il join con le altri classi della base extension. Se le
   *  informazioni estensionali non sono gestite allora il campo joir
   *  dovra‘ essere vuoto o null in
   *  modo che non venga fatta nessuna modifica alla query.
   *
   *  @param queryTrasf e‘ un oggetto di tipo <i>Oql_SimpleQuery</i>
   *         contenente la query da aggiornare
   *  @param jat e‘ un vettore di stringhe contenente i nomi degli
   *         attributi locali da
   */
  private void completeQuery(Oql_SimpleQuery queryTrasf, Vector jat)
    {
      int i = 0;
      Oql_Identifier joinA = null;
      if(jat != null)
          for( i=0; i<jat.size(); i++ )
            {
              joinA = new Oql_Identifier((String)jat.get(i));
              queryTrasf.selectClause.add(joinA);
            }
    }
}
```

Di seguito viene riportata la documentazione prodotta:

.

.

.

Appendice D

Grammatica OQL

La grammatica OQL viene descritta usando la notazione BNF e utilizzando la seguente simbologia:

- `symbol` indica una espressione OQL che non viene tradotta ma che è necessario specificare in quanto è prevista dalla sintassi.

- **symbol** indica un elemento terminale del linguaggio.

- *symbol_name* indica che deve essere specificato un identificatore il cui significato semantico è indicato dalla prima parte del nome.

- *symbol_literal* indica che deve essere specificato un simbolo di tipo literal. Ad es. “una stringa” viene indicato come *string_literal*

⟨Query⟩ ::= **(** ⟨Query⟩ **)** |
⟨SelectExpr⟩

⟨SelectPreamble⟩ ::= **select distinct** |
select

⟨SelectExpr⟩ ::= ⟨SelectPreamble⟩ ⟨ProjectionList⟩
⟨FromClause⟩
⟨WhereClause⟩

⟨ProjectionList⟩ ::= ⟨ProjectionAttributes⟩ |
*****

⟨ProjectionAttributes⟩ ::= ⟨Attribute⟩ |
⟨ProjectionAttributes⟩ **,** ⟨Attribute⟩

⟨Attribute⟩ ::= ⟨Projection⟩ |
⟨Property⟩

⟨Projection⟩ ::= **(** ⟨Projection⟩ **)** |
⟨Identifier⟩ **,** ⟨Property⟩ |
⟨Property⟩ **as** ⟨Identifier⟩ |

⟨Property⟩ ::= **(** ⟨Property⟩ **)** |
⟨Basic⟩ |
⟨Identifier⟩ |
⟨Accesor⟩

⟨Basic⟩ ::= **nil** |
true |
false |
⟨FloatLiteral⟩ |
⟨IntegerLiteral⟩ |
⟨StringLiteral⟩

⟨StringLiteral⟩ ::= “ ⟨String⟩ “

⟨IntegerLiteral⟩ ::= ⟨UnsignedLong⟩ |
⟨Sign⟩ ⟨UnsignedLong⟩

⟨FloatLiteral⟩ ::= **.** ⟨UnsignedLong⟩ |
⟨Sign⟩ **.** ⟨UnsignedLong⟩ |
⟨IntegerLiteral⟩ **.** ⟨UnsignedLong⟩

⟨Sign⟩ ::= + | −

⟨Digit⟩ ::= **0** | **1** | . . . | **9**

⟨Char⟩ ::= **a** | **b** | . . . | **z** | **A** | **B** | . . . | **Z**

⟨UnsignedLong⟩ ::= ⟨Digit⟩ |
⟨Digit⟩ ⟨UnsignedLong⟩

⟨String⟩ ::= ⟨Char⟩ | ⟨Digit⟩ | _ |
⟨Char⟩ ⟨String⟩ |
⟨Digit⟩ ⟨String⟩ |
_ ⟨String⟩

⟨Identifier⟩ ::= ⟨Char⟩ |
⟨Char⟩ ⟨String⟩

⟨Accessor⟩ ::= ⟨Identifier⟩ ⟨Path⟩ ⟨Accessor⟩ |
⟨Identifier⟩ ⟨Path⟩ ⟨Identifier⟩

⟨Path⟩ ::= . | −>

⟨FromClause⟩ ::= **from** ⟨VariableDeclaration⟩

⟨VariableDeclaration⟩ ::= ⟨Identifier⟩ |
⟨Identifier⟩ **as** ⟨Identifier⟩ |
⟨Identifier⟩ ⟨Identifier⟩

⟨WhereClause⟩ ::= ϵ |
where ⟨Predicates⟩

⟨Predicates⟩ ::= **(** ⟨Predicates⟩ **)** |
⟨Comparison⟩ |
⟨BooleanExpr⟩ |
⟨CollectionExpr⟩

⟨Comparison⟩ ::= ⟨Operand⟩ ⟨ComparisonOperator⟩
⟨Quantifier⟩ ⟨Operand⟩ |
⟨Property⟩ **like** ⟨StringLiteral⟩

⟨ComparisonOperator⟩ ::= $> \mid >= \mid < \mid <= \mid = \mid != $

⟨Quantifier⟩ ::= ϵ |
some |
any |
all

⟨Operand⟩ ::= ⟨Basic⟩ |
⟨SimpleExpr⟩ |
⟨Property⟩

⟨SimpleExpr⟩ ::= ⟨Property⟩ + ⟨Property⟩ |
⟨Property⟩ - ⟨Property⟩ |
⟨Property⟩ / ⟨Property⟩ |
⟨Property⟩ * ⟨Property⟩ |
- ⟨Property⟩ |
+ ⟨Property⟩ |
⟨Property⟩ **mod** ⟨Property⟩ |
abs(⟨Property⟩ **)** |
⟨Property⟩ || ⟨Property⟩

⟨BooleanExpr⟩ ::= **not** ⟨Predicates⟩ |
⟨Predicates⟩ **and** ⟨Predicates⟩ |
⟨Predicates⟩ **or** ⟨Predicates⟩

⟨CollectionExpr⟩ ::= **for all** ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
exists ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
exists(⟨Accessor⟩ **)** |
unique(⟨Accessor⟩ **)** |
⟨Property⟩ **in** ⟨Condition⟩ |
count(⟨Property⟩ **)** |
sum(⟨Property⟩ **)** |
min(⟨Property⟩ **)** |
max(⟨Property⟩ **)** |
avg(⟨Property⟩ **)**

Appendice E

Restrizione del OQL per le BasicQuery

Le "*base extension*" costituiscono di fatto una restizione del linguaggio OQL. Di seguito viene riportata la descrizione in forma BNF di tale restrizione.

⟨Query⟩ ::= **(** ⟨Query⟩ **)** |
⟨SelectExpr⟩

⟨SelectPreamble⟩ ::= **select distinct** |
select

⟨SelectExpr⟩ ::= ⟨SelectPreamble⟩ ⟨ProjectionList⟩
⟨FromClause⟩
⟨WhereClause⟩

⟨ProjectionList⟩ ::= ⟨ProjectionAttributes⟩ |
*****

⟨ProjectionAttributes⟩ ::= ⟨Attribute⟩ |
⟨ProjectionAttributes⟩ **,** ⟨Attribute⟩

⟨Attribute⟩ ::= ⟨Projection⟩ |
⟨Property⟩

⟨Projection⟩ ::= **(** ⟨Projection⟩ **)** |
⟨Identifier⟩ **,** ⟨Property⟩ |
⟨Property⟩ **as** ⟨Identifier⟩ |

⟨Property⟩ ::= **(** ⟨Property⟩ **)** |
⟨Basic⟩ |
⟨Identifier⟩ |
⟨Accesor⟩

⟨Basic⟩ ::= **nil** |
true |
false |
⟨FloatLiteral⟩ |
⟨IntegerLiteral⟩ |
⟨StringLiteral⟩

⟨StringLiteral⟩ ::= “ ⟨String⟩ “

⟨IntegerLiteral⟩ ::= ⟨UnsignedLong⟩ |
⟨Sign⟩ ⟨UnsignedLong⟩

⟨FloatLiteral⟩ ::= **.** ⟨UnsignedLong⟩ |
⟨Sign⟩ **.** ⟨UnsignedLong⟩ |
⟨IntegerLiteral⟩ **.** ⟨UnsignedLong⟩

⟨Sign⟩ ::= + | −

⟨Digit⟩ ::= **0** | **1** | . . . | **9**

⟨Char⟩ ::= **a** | **b** | . . . | **z** | **A** | **B** | . . . | **Z**

⟨UnsignedLong⟩ ::= ⟨Digit⟩ |
⟨Digit⟩ ⟨UnsignedLong⟩

⟨String⟩ ::= ⟨Char⟩ | ⟨Digit⟩ | _ |
⟨Char⟩ ⟨String⟩ |
⟨Digit⟩ ⟨String⟩ |
_ ⟨String⟩

⟨Identifier⟩ ::= ⟨Char⟩ |
⟨Char⟩ ⟨String⟩

⟨Accessor⟩ ::= ⟨Identifier⟩ ⟨Path⟩ ⟨Accessor⟩ |
⟨Identifier⟩ ⟨Path⟩ ⟨Identifier⟩

⟨Path⟩ ::= . | − >

⟨FromClause⟩ ::= **from** ⟨VariableDeclaration⟩

⟨VariableDeclaration⟩ ::= ⟨Identifier⟩ |
⟨Identifier⟩ **as** ⟨Identifier⟩ |
⟨Identifier⟩ ⟨Identifier⟩

⟨WhereClause⟩ ::= ϵ |
where ⟨Predicates⟩

⟨Predicates⟩ ::= **(** ⟨Predicates⟩ **)** |
⟨Comparison⟩ |
⟨BooleanExpr⟩ |
⟨CollectionExpr⟩

⟨Comparison⟩ ::= ⟨Operand⟩ ⟨ComparisonOperator⟩ ⟨Quantifier⟩
⟨Operand⟩ |
⟨Property⟩ **like** ⟨StringLiteral⟩

⟨ComparisonOperator⟩ ::= > | >= | < | <= | = | **!** =

⟨Quantifier⟩ ::= ϵ |
some |
any |
all

⟨Operand⟩ ::= ⟨Basic⟩ |
⟨SimpleExpr⟩ |
⟨Property⟩

⟨SimpleExpr⟩ ::= ⟨Property⟩ + ⟨Property⟩ |
⟨Property⟩ - ⟨Property⟩ |
⟨Property⟩ / ⟨Property⟩ |
⟨Property⟩ * ⟨Property⟩ |
- ⟨Property⟩ |
+ ⟨Property⟩ |
⟨Property⟩ **mod** ⟨Property⟩ |
abs(⟨Property⟩ **)** |
⟨Property⟩ || ⟨Property⟩

⟨BooleanExpr⟩ ::= **not** ⟨Predicates⟩ |
⟨Predicates⟩ **and** ⟨Predicates⟩ |
⟨Predicates⟩ **or** ⟨Predicates⟩

⟨CollectionExpr⟩ ::= **for all** ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
exists ⟨Identifier⟩ **in** ⟨Property⟩ **:** ⟨Condition⟩ |
exists(⟨Accessor⟩ **)** |
unique(⟨Accessor⟩ **)** |
⟨Property⟩ **in** ⟨Condition⟩ |
count(⟨Property⟩ **)** |
sum(⟨Property⟩ **)** |
min(⟨Property⟩ **)** |
max(⟨Property⟩ **)** |
avg(⟨Property⟩ **)**

Bibliografia

[1] Simone Montanari. Un approccio intelligente all' integrazione di sorgenti eterogenee di informazione. Tesi di laurea, Università di Modena, Facoltà di ingegneria Informatica, 1996-1997.

[2] Alberta Rabitti. Architettura di un mediatore per un sistema di integrazione di sorgenti distribuite ed autonome. Tesi di laurea, Università di Modena, Facoltà di ingegneria Informatica, 1997-1998.

[3] S.Bergamaschi, S.Castano, S.De Capitani di Vimercati, S.Montanari, and M.Vincini. An intelligent approach to information integration. *Accepted for: Formal Ontology in Information Systems FOIS98*.

[4] S.Bergamaschi, S.Castano, S.De Capitani di Vimercati, S.Montanari, and M.Vincini. exploiting schema knowledge for the integration of heterogeneous sources. *Accepted for: Sistemi Evoluti per Basi di Dati, SEBD98*.

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

[6] Arpa i^3 reference architecture. Available at http://www.isse.gmu.edu/I3_Arch/index.html.

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

[8] Gio Wiederhold. Knowledge versus data. In Springer Vergal, editor, *On Knowledge Base Management Systems: Integrating Artificial Intelligegence and Database Technologies*, pages 77–82. Springer Vergal, 1986. Chapter 7.

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

[10] S. De Capitani di Vimercati S. Bergamaschi, S. Castano and M. Vincini. Momis: An intelligent system for the integgration of semistructured data, November 1998.

[11] R. Hull. Managing semantic heterogeneity in databases: A theoretical perspective. Technical report, Bell Laboratories, 1996.

[12] Sap hone page. Available at http://www.sap.com.

[13] Sonia Bergamaschi. Extraction of informations from highly heterogeneous source of textual data. In *First International Workshop CIA-97 COOPERATIVE INFORMATION AGENTS - DAI meets Database Systems, University of Kiel, Kiel, Germany*, 1997.

[14] W.H. Inmon and C. Kelley. Rdb/vms: Developing the data warehouses, 1993.

[15] Object Request Broker Task Force. The common object request broker: Architecture and specification, December 1993.

[16] R.G.G. Cattell. *The Object Database Standard: ODMG-93, Release 1.1*. Morgan Kaufmann Publishers, Inc., 1994.

[17] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency checking in complex object database schemata with integrity constraints. Technical Report 103, CIOC–CNR, Viale Risorgimento, 2 Bologna, Italia, September 1994.

[18] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Applied Intelligence: The International Journal of Artificial Intelligence, Neural Networks and Complex Problem Solving Technologies*, 4:185–203, 1994.

[19] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. Odb-qoptimizer: a tool for semantic query optimization in oodb. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, April 1997.

[20] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. a description logics based tool for schema validation and semanti query optimization in object oriented databases. Technical report, sesto convegno AIIA, 1997.

[21] Domenico Beneventano, Sonia Bergamaschi, Claudio Sartori, and Maurizio Vincini. Odb-tools: a description logics based tool for schema validation and

semantic query optimization in object oriented databases. In *Proc. of Int. Conference of the Italian Association for Artificial Intelligence (AI*IA97)*, Rome, 1997.

[22] 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.

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

[24] S.Bergamaschi, S.Castano, S.De Capitani di Vimercati, S.Montanari, and M.Vincini. An intelligent system for the integration of semistructured and strucurer data. *Submitted for: Information Systems. special Issue on Semistructured Data*.

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

[26] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[27] B. Nebel. Terminological cycles: semantics and computational properties. In J. Sowa, editor, *Principles of Semantic Networks*. Morgan Kaufmann, San Mateo, Cal. USA, 1991.

[28] R.J. Brachman and H.J. Levesque. The tractability of subsumption in frame-based description languages. In *AAAI*, pages 34–37, 1984.

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

[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 - Lecture Notes in Computer Science N. 173*, pages 51–67. Springer-Verlag, 1984.

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

[36] D. Beneventano, S. Bergamaschi, A. Garuti, C. Sartori, and M. Vincini. ODB-reasoner: un ambiente per la verifica di schemi e l'ottimizzazione di interrogazioni in OODB. In *Quarto Convegno Nazionale su Sistemi Evoluti per Basi di Dati - SEBD96, San Miniato*, pages 181–200, 1996.

[37] M. Buchheit, M. A. Jeusfeld, W. Nutt, and M. Staudt. Subsumption between queries to object-oriented database. In *EDBT*, pages 348–353, 1994.

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

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

[40] J. Gilarranz, J. Gonzalo, and F. Verdejo. Using the eurowordnet multilingual semantic database. In *Proc. of AAAI-96 Spring Symposium Cross-Language Text and Speech Retrieval*, 1996.

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

[42] 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.

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

[44] S. Castano, V. De Antonellis, and S. De Capitani Di Vimercat. Global viewing of heterogeneous data sources. Technical Report 98-08, Dip. Elettronica e Informazione, Politecnico di Milano, Milano, Italy, 1998.

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

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

[47] S. Castano, V. De Antonellis, M.G. Fugini, and B. Pernici. Conceptual schema analysis: Techniques and applications. *accepted for publication on ACM Transactions on Database Systems*, 1997.

[48] I. Schmitt and C. Türker. An Incremental Approach to Schema Integration by Refining Extensional Relationships. In G. Gardarin, J. French, N. Pissinou, K. Makki, and L. Bougamin, editors, *Proc. of the 7th ACM CIKM Int. Conf. on Information and Knowledge Management, November 3–7, 1998, Bethesda, Maryland, USA*, pages 322–330, New York, 1998. ACM Press.

[49] I. Schmitt and G. Saake. Merging inheritance hierarchies for database integration. *IEEE Trans. on Knowledge and Data Engineering*.

[50] C. Carpineto and G. Romano. Galois: An order-theoretic approach to conceptual clustering. In *Macine Learning Conference*, pages 33–40, 1993.

[51] S. Shenoy and M. Ozsoyoglu. A system for semantic query optimization. *ACM-SIGMOD*, pages 181–195, May 1987.

[52] S. Shenoy and M. Ozsoyoglu. Design and implementation of a semantic query optimizer. *IEEE Trans. Knowl. and Data Engineering*, 1(3):344–361, September 1989.

[53] Maurizio Vincini. Utilizzo di tecniche di intelligenza artificiale nell'integrazione di sorgenti informative eterogenee. Tesi di laurea, Università di Modena, Facoltà di ingegneria Informatica, 1997-1998.

[54] Alberto Zanoli. Si-designer, un tool di ausilio all'integrazione di sorgenti di dati eterogenee distribuite: progetto e realizzazione. Tesi di laurea, Università di Modena, Facoltà di ingegneria Informatica, 1997-1998.

[55] Byacc/java. Available at http://www.lincom-asg.com/ rjamison/byacc/.

[56] Daniel P.Miranker and Vasilis Samoladas. Alamo: an architecture for integrating heterogenous data sources. In *Proceedings of the 4th KRDB Workshop*, Athens, Greece, August 1997.

[57] Oliver M.Duschka and Micheal R.Genesereth. Infomaster - an information integration toolkit. Technical report, Department of Computer Science. Stanford University, 1996.

[58] V.S. Subrahmanian, Sibel Adali, Anne Brink, J.ames, J. Lu, Adil Rajput, Timothy J. Rogers, Robert Ross, and Charles Ward. Hermes: A heterogeneous reasoning and mediator system. Available at http://www.cs.umd.edu/projects/hermes/overview/paper/index.html.

[59] Alon Levy, Dana Florescu, Jaewoo Kang, Anand Rajaraman, and Joanne J. Ordille. The information manifold project. Available at http://www.research.att.com/ levy/imhome.html.

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

[61] 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.

[62] H. Garcia-Molina Y.Papakonstantinou. Object fusion in mediator systems (extended version). 1997.

[63] Y.Papakonstantinou, H.Garcia-Molina, and J.Widom. Object exchange across heterogeneous information sources. Technical report, Stanford University, 1994.

[64] D.Quass, A.Rajaraman, Y.Sagiv, J.Ullman, and J.Widom. Querying semistructured heterogeneous informations. Technical report, Stanford University, 1996.

[65] 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.

[66] 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.

[67] R. Cattel, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1996.

[68] 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.

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