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

Progetto e realizzazione di tecniche di Object Fusion nel sistema MOMIS

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Micol Ferrari

Correlatore
Dott. Ing. Domenico Beneventano

Controrelatore
Chiar.mo Prof. Flavio Bonfatti

Anno Accademico 1999 - 2000

Parole chiave:

Conoscenza Intensionale
Conoscenza Estensionale
Regole di join
Object Fusion
Chiave Semantica
Join Map
Join Table

RINGRAZIAMENTI

Ringrazio la Professoressa Sonia Bergamaschi, il Dott. Ing. Domenico Beneventano, l'Ing. Alberto Corni e tutti i ragazzi del gruppo MOMIS per l'aiuto fornito e la costante collaborazione.

Un grazie tutto speciale va alla mia famiglia.

Indice

Introduzione **1**

1 Integrazione Intelligente delle Informazioni **5**
1.1 Sistemi I^3 6
1.1.1 Programma I^3 6
1.1.2 Architettura di riferimento 7
1.1.3 Il mediatore 10
1.1.4 Problematiche da affrontare 12
1.2 Il sistema MOMIS 14
1.2.1 L'approccio adottato 14
1.2.2 L'architettura generale di MOMIS 16
1.2.3 Tools di supporto 18

2 Integrazione degli schemi e Query Processing in MOMIS **21**
2.1 Integrazione degli schemi 21
2.1.1 Esempio di riferimento 22
2.1.2 Integrazione intensionale 23
2.1.3 Integrazione estensionale 30
2.2 Definizione di un modello di rappresentazione dello schema globale 36
2.2.1 Schema Globale 37
2.2.2 Base Extension 38
2.2.3 Schema Virtuale e Gerarchia Estensionale 39
2.3 Query Processing 40

3 Il problema dell'Object Fusion **45**
3.1 Le relazioni estensionali 45
3.2 Identificazione delle istanze 47
3.2.1 Tipi di identificazione 48
3.2.2 Possibili approcci 48
3.3 Le regole di join 50
3.3.1 Reperimento della conoscenza 50

3.3.2 Regole di join in linguaggio naturale . . . 52
3.3.3 Intervento del progettista . . . 58
3.3.4 Formalizzazione delle regole di join . . . 61
3.4 Strutture dati: Join Map e Join Table . . . 64
3.5 Il processo di Object Fusion . . . 66
3.6 Sviluppi futuri . . . 71

4 Stato dell'arte 73
4.1 TSIMMIS . . . 73
4.1.1 Object Fusion basata su oid semantici . . . 74
4.1.2 Query Processing . . . 76
4.1.3 Considerazioni . . . 78
4.2 Interoperabilità semantica . . . 78
4.2.1 Riconciliazione semantica . . . 79
4.2.2 Considerazioni . . . 81

5 Il modulo joinMap: progetto e realizzazione del software 83
5.1 Organizzazione del software . . . 83
5.1.1 Il package globalschema . . . 84
5.1.2 Il package joinMap . . . 91
5.2 L'interfaccia grafica . . . 95
5.2.1 SI-Designer . . . 96
5.2.2 Il pannello "Join Map" . . . 97
5.3 Il software . . . 102

Conclusioni 103

A Glossario I^3 105
A.1 Architettura . . . 105
A.2 Servizi . . . 107
A.3 Risorse . . . 110
A.4 Ontologia . . . 112

B Il linguaggio descrittivo ODL$_{I^3}$ 115

C Esempio di riferimento in ODL$_{I^3}$ 117

D L'architettura CORBA 119

E Le classi java JoinMap e JoinTable 123
E.1 JoinMap.java . . . 123
E.2 JoinTable.java . . . 126

Elenco delle figure

1.1 Diagramma dei servizi I^3 8
1.2 Servizi I^3 presenti nel mediatore 12
1.3 Architettura generale di MOMIS 16
1.4 Architettura di ODB-Tools 18

2.1 Esempio di riferimento 22
2.2 Albero di affinità 26
2.3 Mapping Table di University_Person 29
2.4 Base extension per la classe University_Person 34
2.5 Tabella delle base extension 34
2.6 Gerarchia estensionale di University_Person 35
2.7 Attività di Query Processing 41

3.1 Estensione della classe di entità 47
3.2 Join attraverso classe intermedia 60
3.3 La Join Table di University_Person 66
3.4 Esecuzione delle Basic Query 67
3.5 Fusione tramite outer join 70

5.1 Modello ad oggetti del package globalschema interessato 84
5.2 Modello ad oggetti degli attributi di join 87
5.3 Algoritmo di Costruzione 92
5.4 Algoritmo di Selezione degli Attributi di Join 93
5.5 Algoritmo di Creazione Automatica della Join Table 95
5.6 Architettura di SI-Designer 97
5.7 Il pannello “Join Map” 98
5.8 Il pannello “Join Map”: Join Table automatica 99
5.9 Il pannello “Join Map”: dichiarazione esplicita delle Join Map . . 100
5.10 Il pannello “Join Map”: il matching tra gli attributi di join 101

D.1 Invocazione di un metodo di un oggetto CORBA remoto 120
D.2 Esempio di albero creato dal naming server 121

D.3 Traduzione in Java di una interfaccia IDL di un oggetto CORBA . 122

Introduzione

L'evoluzione avuta negli ultimi anni dalle reti di calcolatori e la crescente presenza di sempre nuove sorgenti di informazioni fanno aumentare non solo la quantità ed il genere di dati accessibili, ma anche la velocità con cui questi possono essere scambiati.
Gli indubbi vantaggi prodotti da questo sviluppo nel panorama informativo, introducono al contempo grosse problematiche legate alle difficoltà d'accesso a questi dati. Infatti, poter accedere ad ingenti moli di dati autonomi, non aumenta necessariamente la qualità finale dell'informazione percepita dall'utente. Il dovere gestire grossi quantitativi di informazioni, implica il doversi destreggiare in problemi di varia natura, quali: la selezione delle sorgenti interessanti, l'analisi e la sintesi dei dati reperiti (spesso duplicati, incosistenti o di varia natura), ecc.
Si delinea, quindi, la necessità di realizzare l'integrazione dei dati che si hanno a disposizione, aumentando la qualità ed il valore dell'informazione da essi fornita. La possibilità di gestire e sviluppare applicazioni intersorgenti capaci di integrare informazioni è un tema di grande interesse, come dimostra la sempre maggiore rilevanza assunta da sistemi, quali i *Datawarehouse*, i *Dataminer*, i *Sistemi di Workflow*,ecc. I domini di impiego di questi sistemi sono innumerevoli: ospedaliero, militare, aziendale, pubblicazioni, ecc [1, 2, 3, 4, 5, 6, 7, 8, 9].

Questa tesi si inserisce in un progetto più ampio denominato **MOMIS** (**M**ediator Envir**O**ment for **M**ultiple **I**nformation **S**ources), sviluppato con l'obiettivo di realizzare l'integrazione semi-automatica delle informazioni contenute all'interno di sorgenti eterogenee e distribuite.
MOMIS adotta un'architettura a tre livelli: il livello centrale è occupato da un modulo *Mediatore*, che realizza la vista aggregata degli schemi costitutivi le singole sorgenti, e che gestisce le query poste dall'utente sullo schema globale.
Tra gli elementi che caratterizzano questo processo, quelli innovativi sono rappresentati dall'impiego di un approccio semantico e dall'uso di logiche descrittive per la rappresentazione degli schemi locali. Questi elementi introducono comportamenti intelligenti, che permettono di realizzare un processo di integrazione

semi-automatico.

L'obiettivo della presente tesi è stato l' analisi delle problematiche inerenti la gestione della conoscenza estensionale ed intensionale, nel processo di riconoscimento e di ricostruzione delle informazioni facenti riferimento alla medesima entità del mondo reale: il processo di Object Fusion. Questo studio ha portato alla definizione di *regole* e alla realizzazione di un modulo software, che permettono di stabilire, in modo semi-automatico, le informazioni funzionali al processo di Object Fusion.

La tesi è organizzata nel seguente modo:

Nel **Capitolo 1** viene introdotto il concetto di Integrazione Intelligente delle Informazioni, descrivendo l'architettura di riferimento I^3e la struttura di un Mediatore. Inoltre, si provvede ad illustrare le scelte implementative adottate per il sistema MOMIS, con particolare riguardo per la sua architettura e per i tools utilizzati.

Nel **Capitolo 2** viene descritta la fase di integrazione degli schemi, illustrando le strutture dati rappresentanti la conoscenza intensionale ed estensionale prodotta. Inoltre, viene data una visione d'insieme della fase di Query Processing, mostrando come questa sfrutti la conoscenza prodotta dalla fase di integrazione degli schemi.

Il **Capitolo 3** si occupa dello studio e della progettazione delle tecniche di Object Fusion, al centro della trattazione di questa tesi. In particolare si provvede alla definizione ed alla formalizzazione di regole e di strutture dati, finalizzate all'individuazione delle informazioni che permettono di identificare istanze facenti riferimento alla medesima entità del mondo reale.

Il **Capitolo 4** realizza una panoramica sullo stato dell'arte: si provvede ad esporre approcci adottati da sistemi affini a MOMIS, nell'affrontare le problemetiche inerenti il processo di Object Fusion.

Il **Capitolo 5** descrive la progettazione e realizzazione del modulo sofware joinMap. Questo modulo si occupa della definizione semi-automatica delle strutture dati funzionali al progetto di Object Fusion. In particolare vengono illustrate le classi ed i metodi, che modellano le strutture dati fondamentali, e l'interfaccia grafica realizzata.

Sono presenti, inoltre, cinque appendici: in Appendice A viene riportato un glossario dei termini usati in ambito I^3; in Appendice B viene mostrata la

grammatica ODL_{I^3}; in Appendice C viene illustrato l'esempio in ODL_{I^3}, che sarà utilizzato come riferimento; in Appendice D viene riportata una panoramica sull'architettura CORBA; in Appendice E è riportato il codice relativo alle classi java *JoinMap* e *JoinTable*, che modellano le strutture dati fondamentali definite in questa tesi.

Capitolo 1

Integrazione Intelligente delle Informazioni

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. Contestualmente all'aumento della probabilità di riperire un dato cercato, aumenta anche la difficoltà di recuperare questo dato in tempi e modi accettabili. Questo perchè le informazioni ed i dati che le quantificano sono di diversa natura (es. testi, immagini, etc ...) ed appartengono a sorgenti eterogenee (es. pagine HTML, DBMS relazionali o ad oggetti, file system, etc ...). Gli standard esistenti (TCP/IP, ODBC, OLE, CORBA, SQL, etc ...) risolvono solo parzialmente i problemi relativi alle diversità hardware e software, dei protocolli di rete e di comunicazione tra moduli; rimangono però irrisolti quelli relativi alla modellazione delle informazioni. Infatti, i modelli di dati e gli schemi si differenziano gli uni dagli altri in modo da dare una struttura logica ai numerosi generi di dati da memorizzare, creando così una eterogeneità semantica non risolvibile dagli standard. Un altro problema non trascurabile è l'*information overload*, ovvero il sovraccarico di informazioni fa sì che l'utente abbia sempre maggiori difficoltà nel discernere ed isolare i dati per lui significativi.
Altre problematiche non trascurabili riguardano: i tempi d'accesso, la salvaguardia della sicurezza ed i costi per il mantenimento della consistenza delle informazioni.

Per far fronte alla molteplicità e complessità degli aspetti appena descritti, le architetture dedicate all'integrazione di sorgenti eterogenee devono essere necessariamente flessibili e modulari.
Gli approcci all'integrazione, descritti in letteratura o effettivamente realizzati, presentano diverse metodologie: la reingegnerizzazione delle sorgenti mediante standardizzazione degli schemi e la creazione di un database distribuito; il *repo-*

sitory indipendence, un approccio che prevede di isolare al di sotto di una vista integrata, le applicazioni ed i dati integrati dalle sorgenti, consentendo la massima autonomia e nascondendo al contempo le differenze esistenti; i *datawarehouse*, che realizzano presso l'utente finale delle viste, ovvero delle porzioni di sorgenti, replicando fisicamente i dati ed affidandosi ad algoritmi di allineamento per assicurarne la consistenza a fronte di modifiche nelle sorgenti.
Nel seguito verrà descritto l'approccio che in letteratura viene indicato come *Intelligent Integration of Information (I^3)* [10], ovvero l'approccio seguito da quei sistemi che realizzano l'*Integration of Information (I^2)*, cioè combinano tra di loro informazioni senza replicare fisicamente i dati, utilizzando tecniche di Intelligenza Artificiale (AI).

1.1 Sistemi I^3

I sistemi che realizzano l'Integrazione Intelligente delle Informazioni, basandosi dulle *descrizioni dei dati*, combinano tra loro informazioni provenienti da diverse sorgenti (o parti selezionate di esse) senza dover ricorrere alla duplicazione fisica dei dati. Questo richiede *conoscenza* ed *intelligenza* volte all'individuazione delle sorgenti e dei dati, nonchè alla loro fusione e sintesi. Ciò viene raggiunto usando tecniche di Intelligenza Artificiale.

1.1.1 Programma I^3

Dal 1992 è operativo il Programma I^3, un progetto di ricerca fondato e sponsorizzato dall'ARPA (Advanced Research Projects Agency), che si prefigge di individuare un'architettura di riferimento che realizzi in maniera automatica l'integrazione di sorgenti di dati eterogenee [11].
I^3 propone l'introduzione di architetture modulari sviluppabili secondo i principi proposti da uno standard che ponga le basi dei servizi da soddisfare dall'integrazione ed abbassi i costi di sviluppo e mantenimento. Questo renderebbe possibile ovviare ai problemi di realizzazione, manutenzione, adattabilità, inoltre la riutilizzazione della tecnologia già sviluppata, rende la costruzione di nuovi sistemi più veloce e meno difficoltosa, con conseguente abbassamento dei costi. Per poter sfruttare un'elevata riusabilità bisogna disporre di interfacce ed architetture standard. Il paradigma suggerito per la suddivisione dei servizi e delle risorse nei diversi moduli si articola su due dimensioni:

- l'orizzontale, divisa in tre livelli: livello utente, moduli intermedi che fanno uso di tecniche di IA, risorse di dati;

- la verticale: molti domini, con un numero limitato (e minore di 10) di sorgenti.

I domini nei vari livelli non sono strettamente connessi, ma si scambiano dati ed informazioni la cui combinazione avviene a livello dell'utilizzatore, riducendo la complessità totale del sistema e permettendo lo sviluppo di applicazioni con finalià diverse.

I^3 si concentra sul livello intermedio della partizione, quello che media tra gli utenti e le sorgenti. In questo livello sono presenti vari moduli, quali:

- **Facilitator** e **Mediator**: ricercano le fonti interessanti e combinano i dati da esse ricevuti;
- **Query Processor**: riformula le query aumentando le loro probabilità di successo;
- **Data Miner**: analizza i dati per estrarre informazioni intensionali implicite.

Nell'Appendice A è presente un glossario di termini comunemente usato in ambito I^3. Questo ha lo scopo di spiegare quei termini che dovessero risultare ambigui o poco chiari, visto il campo recente ed in evoluzione in cui si muove il progetto.

1.1.2 Architettura di riferimento

L'obiettivo del Programma I^3 è di ridurre il tempo necessario per la realizzazione di un integratore di informazioni, fornendo una raccolta e una formalizzazione delle soluzioni prevalenti finora nel campo della ricerca. Come abbiamo visto, la complessitá del processo di integrazione é tale da rendere estremamente utile la proposta di un'architettura di riferimento standard, che rappresenti alcuni dei servizi che un integratore di informazioni deve contenere e le possibili interconnessioni fra di essi. Il programma I^3 individua cinque famiglie di attività omogenee, illustrate in Figura 1.1 unitamente ai loro legami. La reciproca interazione tra queste attività consente di eseguire le operazioni di comunicazione, traduzione ed integrazione dei dati nelle sorgenti.

Sono inoltre evidenti due assi, uno orizzontale ed uno verticale, che permettono di intuire i diversi compiti dei vari servizi. Sull'asse orizzontale si hanno i servizi di Coordinamento ed Amministrazione, che hanno il compito di mantenere informazioni sulle capacità delle sorgenti, vale a dire che tipo di dati sono in grado di fornire e come vanno interrogate. Sempre sull'asse orizzontale si hanno poi i servizi Ausiliari, che sono responsabili delle attività di arricchimento semantico e di supporto.

Sull'asse verticale, i servizi di Coordinamento, di Integrazione e Trasformazione

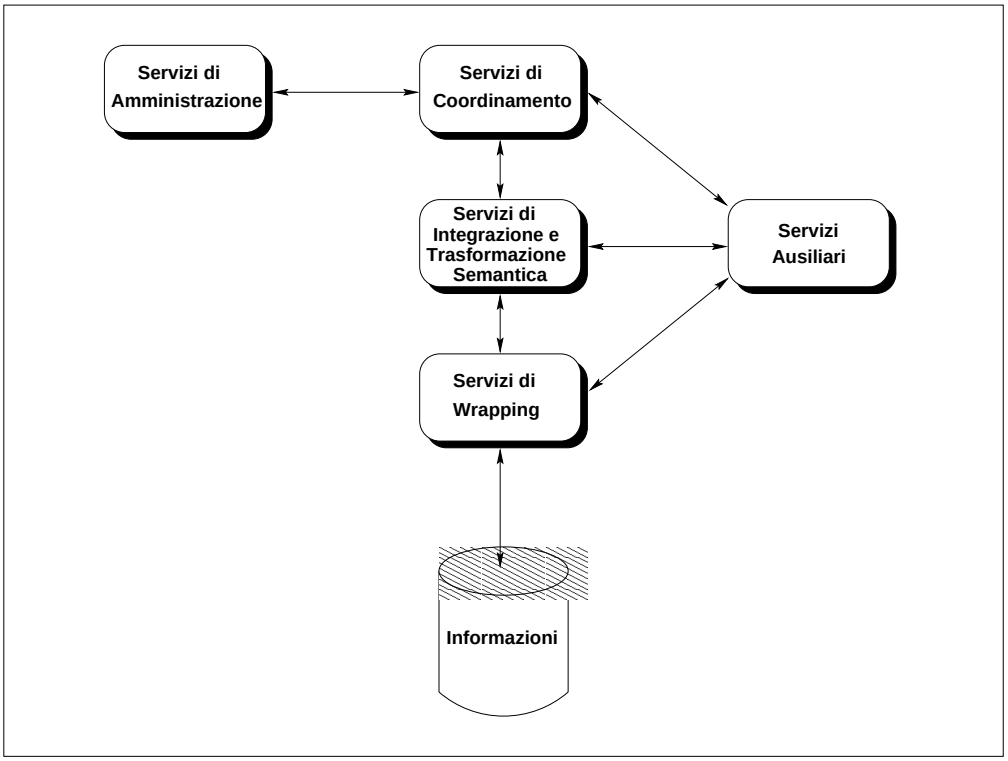

Figura 1.1: Diagramma dei servizi I^3

semantica e di Wrapping evidenziano come avviene lo scambio di informazioni. Analizzando i vari servizi nel dettaglio:

- **Servizi di Coordinamento**

  Sono servizi ad alto livello che costituiscono l'interfaccia con l'utente, dandogli l'impressione di trattare con un sistema omogeneo. Grazie alle funzionalità messe a disposizione dalle altre famiglie di servizi, essi permettono di individuare le sorgenti di dati interessanti, ovvero che probabilmente possono dare risposta ad una determinata risposta dell'utente. Conformemente col tipo di integratore che si è intenzionati di realizzare, i servizi di Coordinamento possono essere:

  – **Facilitation e Brokering Services**: forniscono una selezione dinamica delle sorgenti in grado di soddisfare la richiesta dell'utente. Il sistema usa un deposito di metadadati per individuare il modulo che puó trattare direttamente questa richiesta, in particolare si parla di Brokering quando é coinvolto un modulo alla volta, oppure di Facilita-

tori o Mediatori se vi sono piú moduli interessati. In quest'ultimo caso la query iniziale viene decomposta in un insieme di sottoquery da inviare a differenti moduli che gestiscono sorgenti distinte, successivamente vengono integrate le risposte per fornirne una prentazione globale all'utente. Questo viene realizzato facendo uso di servizi di Query Decomposition e di tecniche di Inferenza (mutuate dall'Intelligenza Artificiale) per una determinazione dinamica delle sorgenti da interrogare.

– **Matchmaking**: il mappaggio fra informazioni integrate e locali é effettuato manualmente da un'operatore in fase di inizializzazione. In questo caso tutte le richieste vengono trattate allo stesso modo.

- **Servizi di Amministrazione**

  Questi servizi sono utilizzati da quelli di Coordinamento per localizzare le sorgenti *utili*, determinare la loro capacità, creare ed interpretare *Template*. I Template sono strutture dati che descrivono i servizi, le fonti ed i moduli da utilizzare per realizzare un determinato task. Queste strutture dati servono quindi per ridurre al minimo le possibilità di decisione del sistema, consentendo di definire a priori le azioni da eseguire a fronte di una determinata richiesta.
  Al posto dei Template è possibile usare le *Yellow pages*, ovvero servizi di directory che mantengono le informazioni sul contenuto delle sorgenti e sul loro stato. In questo modo, le Yellow pages consentono al Mediatore di inviare la richiesta di informazioni alla sorgente giusta o, se non fosse disponibile, ad una equivalente.
  Tra questi tipi di servizio vi sono il *Browser*, che permette di "navigare" tra le descrizioni degli schemi delle sorgenti, recuperando informazioni, e gli *Iterative Query Formulation*, che aiutano l'utente a rilassare o specificare meglio alcuni vincoli dell'interrogazione al fine di ottenere risposte più precise.

- **Servizi di Integrazione e Trasformazione Semantica**

  Questi servizi supportano le manipolazioni necessarie per l'integrazione e la trasformazione delle informazioni. Hanno in input una o più sorgenti di dati, e restituiscono come output la "vista" integrata o trasformata di queste informazioni. Spesso sono indicati come servizi di *Mediazione*, essendo tipici dei moduli mediatori.
  I principali sono:

  - **Servizi di integrazione di schemi**: creano il vocabolario e le ontologie condivise dalle sorgenti, integrano gli schemi in una vista globale, mantengono il mapping tra schemi globali e sorgenti;
  - **Servizi di integrazione di informazioni**: aggregano, riassumono ed astraggono i dati per fornire presentazioni analitiche significative;
  - **Servizi di supporto al processo di integrazione**: sono utilizzati quando una query deve essere scomposta in più sottoquery da inviare a fonti differenti, con la necessità di integrare poi i loro risultati.

- **Servizi di Wrapping**

  Il fine di questi servizi è far sì che le fonti di informazione aderiscano ad uno standard. Sono praticamente dei traduttori dai sistemi locali ai servizi di alto livello dell'integratore.
  I servizi di Wrapping permettono ai servizi di Coordinamento e di Mediazione di manipolare in modo uniforme le sorgenti locali.
  Forniscono un'interfaccia che, seguendo gli standard più diffusi (ad esempio: SQL come linguaggio di interrogazione, CORBA come protocollo di scambio), permette alle sorgenti estratte di essere accedute dal maggior numero possibile di sistemi mediatori.

- **Servizi Ausiliari**

  Aumentano le funzionalità degli altri servizi. Possono svolgere varie funzioni, tra cui: monitoraggio del sistema, propagazione di aggiornamenti, attività di ottimizzazione, etc.

1.1.3 Il mediatore

Secondo la definizione proposta da Wiederhold in [12] "un mediatore è un modulo software che sfrutta la conoscenza su un certo insieme di dati per creare informazioni per una applicazione di livello superiore . . . Dovrebbe essere piccolo e semplice, così da poter essere amministrato da uno, o al più pochi, esperti."
Un mediatore presenta allora i seguenti compiti:

- assicurare un servizio stabile, anche nel caso di cambiamento delle 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.

Il progetto MOMIS, di cui questa tesi fa parte, ha come obiettivo la progettazione e realizzazione di un **Mediatore**, come descritto in [5, 4, 3]. L'ipotesi di avere a che fare esclusivamente con sorgenti di dati strutturati e semistrutturati, ha consentito di restringere il campo applicativo del sistema con una conseguente diminuzione delle problematiche riscontrate in fase di progettazione e realizzazione. L'approccio architetturale scelto è quello classico, che si sviluppa su tre livelli principali:

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

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

3. *wrapper*: ogni wrapper gestisce una singola sorgente, convertendo 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 nella Sezione 1.1.2, l'architettura del mediatore che si è progettato è riportata in Figura 1.2. In particolare sono stati sviluppati i servizi di Integrazione e Trasformazione Semantica. Inoltre l'impostazione architetturale mostra come il sistema mediatore progettato vuole distaccarsi dall'approccio *strutturale*, cioè sintattico, tuttora dominante tra i sistemi presenti sul mercato [13, 14, 15].
Quando si parla di approccio strutturale, si fa riferimento all'uso di un self-describing model per rappresentare gli oggetti da integrare, limitando l'uso delle informazioni semantiche alle regole predefinite dall'operatore. Il sistema non conosce a priori la semantica di un oggetto recuperato da una sorgente, bensì è l'oggetto stesso che, attraverso delle etichette, si autodescrive. I vantaggi di questo approccio sono: la possibilità di integrare in modo completamente trasparente al mediatore basi di dati fortemente eterogenee e magari mutevoli nel tempo; per trattare in modo omogeneo dati che descrivono lo stesso concetto o che hanno concetti in comune, ci si basa sulla definizione manuale di rule, che permettono di identificare i termini, che devono essere condivisi da più oggetti.

Altri progetti, tra cui MOMIS, seguono invece un approccio all'integrazione di tipo *semantico*, che prevede che siano soddisfatti i seguenti punti:

- il mediatore deve conoscere, per ogni sorgente, lo schema concettuale (metadati);

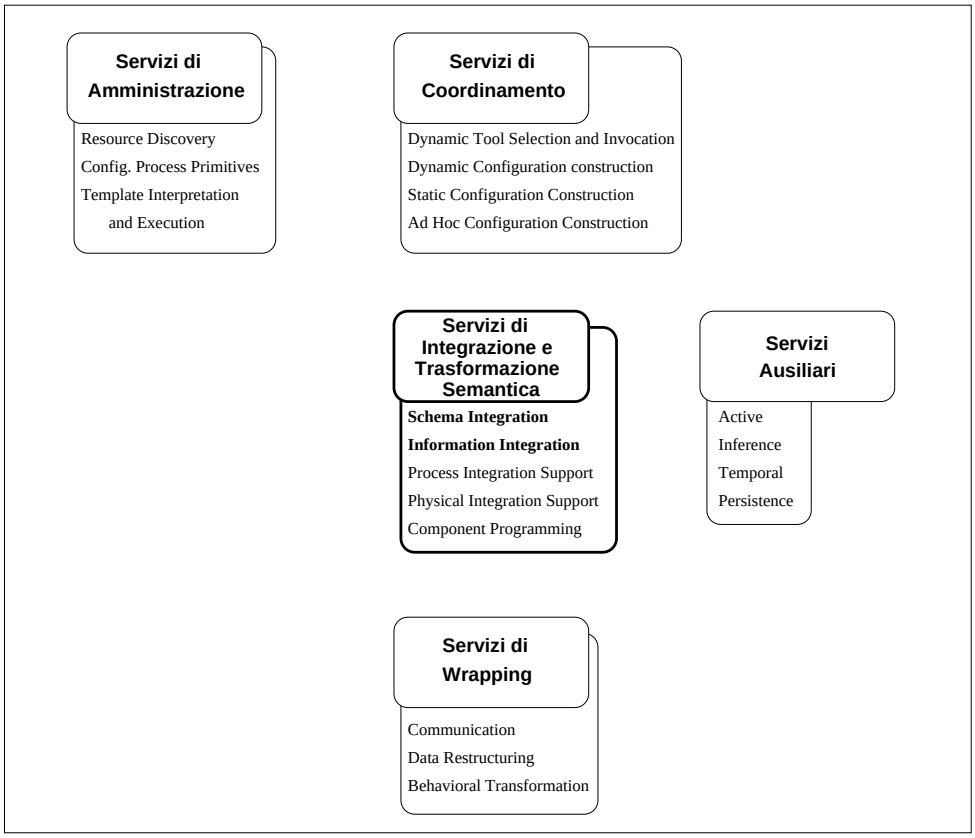

Figura 1.2: Servizi I^3 presenti nel mediatore

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

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

1.1.4 Problematiche da affrontare

Pur avendo a disposizione gli schemi concettuali delle varie sorgenti, non è certamente un compito banale individuare i concetti comuni ad esse e le relazioni che

possono legarli, nè tantomeno è semplice realizzare una loro coerente integrazione. Trascurando le differenze dei sistemi fisici (alle quali dovrebbero provvedere i moduli wrapper) i problemi a livello di mediazione che si è dovuto risolvere (o coi quali si è dovuti scendere a compromessi) sono:

- **Problemi ontologici**
  Come riportato in Appendice A , per *ontologia* si intende, in questo ambito, "l'insieme dei termini e delle relazioni usate in un dominio, per indicare oggetti e concetti". In sostanza con ontologia ci si riferisce a quell'insieme di termini che, in un particolare dominio applicativo, denotano una particolare conoscenza e fra i quali non esiste ambiguitá perché sono condivisi dall'intera comunitá di utenti del dominio applicativo stesso.
  Fra i diversi livelli di ontologia esistenti (*top-level ontology, domain and task ontology, application ontology*, etc . . .) [16, 17], ognuno con le proprie problematiche, si è assunto 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).

- **Problemi semantici**
  Pur ipotizzando che anche sorgenti diverse condividano una visione similare del problema da modellare, e quindi un insieme di concetti comuni, é improbabile che usino la stessa semantica, cioé gli stessi vocaboli e le stesse strutture dati per rappresentare questi concetti.
  Come riportato in [18] la causa principale delle differenze semantiche si può identificare nelle diverse concettualizzazioni del mondo esterno che persone distinte possono avere, ma non è l'unica. Le differenze nei sistemi di DBMS possono portare all'uso di differenti modelli per la rappresentazione della porzione di mondo in questione: partendo così dalla stessa concettualizzazione, determinate relazioni tra concetti avranno strutture diverse a seconda che siano realizzate attraverso un modello relazionale o ad oggetti.

  L'obiettivo dell'integratore, che è fornire un accesso integrato ad un insieme di sorgenti, si traduce allora nel non facile compito di identificare i concetti comuni all'interno di queste sorgenti e risolvere le differenze semantiche che possono essere presenti tra di loro. Possiamo classificare queste contraddizioni semantiche in tre gruppi principali:

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

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

È però possibile 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.

1.2 Il sistema MOMIS

Considerando le problematiche descritte nel paragrafo precedente, nonché alcuni sistemi preesistenti [13, 14, 15, 19, 20, 21, 22, 23, 24, 25, 26] , si è giunti alla progettazione di un sistema intelligente di integrazione di informazioni da sorgenti di dati strutturati e semistrutturati denominato **MOMIS** (**M**ediator Envir**O**nment for **M**ultiple **I**nformation **S**ources). Il contributo innovativo di questo progetto, rispetto ad altri similari, risiede nell'impiego di un approccio semantico e nell'uso di logiche descrittive per la rappresentazione degli schemi delle sorgenti, elementi che introducono comportamenti intelligenti in grado di rendere semi-automatica la fase di integrazione . Un lavoro approfondito è stato svolto anche riguardo alla fase di *query processing* [1, 7, 2, 27] , cioè per il processo che, dalla query posta sullo schema unificato, provvede a generare automaticamente le sottoquery da inviare alle sorgenti ed ad integrare i risultati.
MOMIS nasce all'interno del progetto MURST 40% INTERDATA dalla collaborazione tra i gruppi operativi dell'Università di Modena e Reggio Emilia e di quella di Milano.

1.2.1 L'approccio adottato

MOMIS adotta un approccio di integrazione delle sorgenti *semantico* e *virtuale* [1]. Il concetto di semantico è stato illustrato nella Sezione 1.1.3 . Con virtuale

[28] si intende invece che la vista integrata delle sorgenti, rappresentata dallo schema globale, non viene *materializzata*, ma il sistema si basa sulla decomposizione delle query e sull'individuazione delle sorgenti da interrogare per generare delle subquery eseguibili localmente; lo schema globale dovrà inoltre disporre di tutte le informazioni atte alla fusione dei risultati ottenuti localmente per poter ottenere una risposta significativa. L'obbiettivo di questa tesi è proprio la definizione di una serie di regole, che agevolino il sistema nel reperimento delle informazioni utili per la realizzazione della fusione.

Le motivazioni che hanno portato all'adozione di un approccio come quello descritto sono varie:

1. la presenza di uno schema globale permette all'utente di formulare qualsiasi interrogazione che sia con esso consistente;

2. le informazioni semantiche che comprende 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 che nel tipo delle sorgenti, ed anche nei loro contenuti (non occorre prevedere onerose politiche di allineamento);

Si é 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. Il modello comune dei dati utilizzato nel sistema (ODM$_{I^3}$) è di alto livello e facilita la comunicazione tra il mediatore ed i wrapper. Per definire questo modello si è cercato di seguire le raccomandazioni relative alla proposta di standardizzazione per i linguaggi di mediazione, nata in ambito I^3: un mediatore deve poter essere in grado di gestire sorgenti dotate di formalismi complessi (ad es. quello ad oggetti) ed altre decisamente più semplici (come i file di strutture), è quindi preferibile l'adozione di un formalismo il più completo possibile.

Per la descrizione degli schemi si è arrivati a definire il linguaggio ODL$_{I^3}$[2, 3, 1, 4] che si presenta come estensione del linguaggio standard ODL proposto dal gruppo di standardizzazione ODMG-93.

Per quanto riguarda il linguaggio di interrogazione si è adottato OQL$_{I^3}$che adotta la sintassi OQL senza discostarsi dallo standard. Questo linguaggio risulta estremamente versatile ed espressivo fornendo la possibiltà di sfruttare le informazioni rappresentate nello schema globale.

Inoltre si é cercato di definire uno standard comune di comunicazione tra i vari moduli MOMIS al fine di rendere ancora piú agevole l'ampliamento futuro. Si é

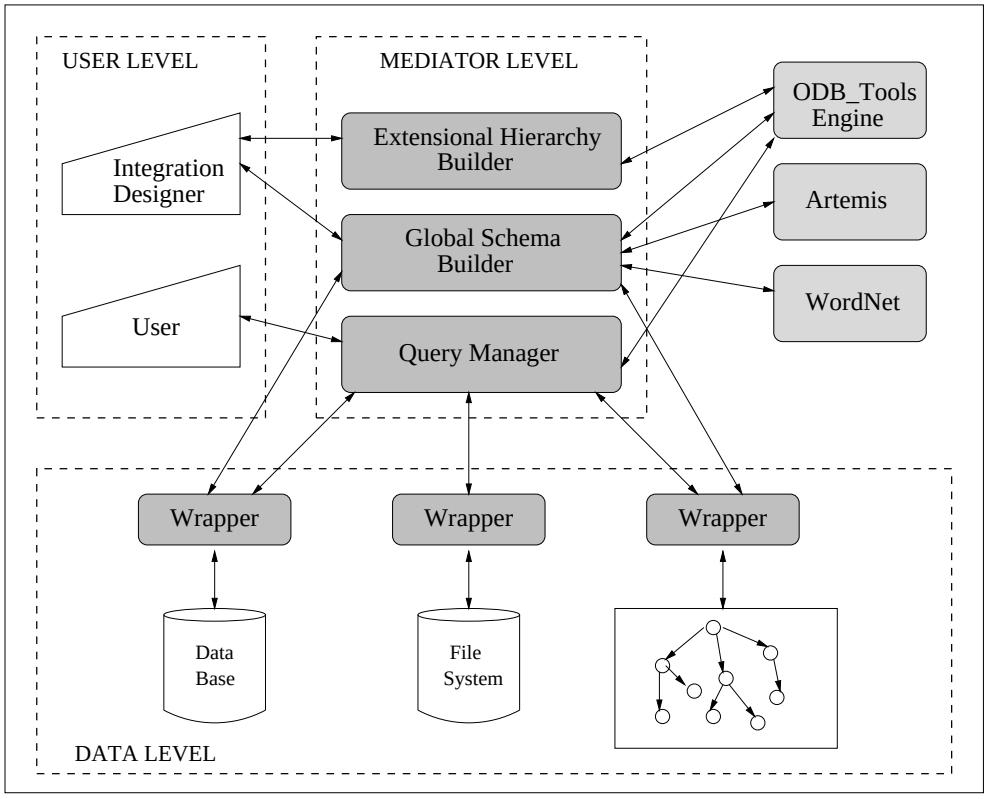

Figura 1.3: Architettura generale di MOMIS

deciso di adottare lo standard CORBA (Common ORB Architecture) per le comunicazioni tra i moduli [29] . CORBA è una tecnologia per l'integrazione, inoltre è ad oggetti ed una modellazione di questo tipo permette di ridurre la complessità di MOMIS: esistono difatti metodologie consolidate per la rappresentazione e progettazione di sistemi ad oggetti (OMT, UML), ma soprattutto per utilizzare un oggetto è sufficiente conoscerne l'interfaccia pubblica e questo favorisce il lavoro degli sviluppatori successivi.

1.2.2 L'architettura generale di MOMIS

Momis è stato progettato per fornire un accesso integrato ad informazioni eterogenee memorizzate sia in sorgenti strutturate, come database relazionali, database ad oggetti e semplici file, sia in sorgenti semistrutturate, come le sorgenti descritte in XML.

Come si può vedere nella Figura 1.3, i componenti del sistema MOMIS sono disposti su tre livelli:

- **Livello Dati**
  A questo livello si trovano i **Wrapper**. Posti al di sopra di ciascuna sorgente, sono i moduli che fungono da interfaccia tra il mediatore vero e proprio e le sorgenti locali di dati. Le funzioni da loro svolte sono:

  – in fase di integrazione forniscono una descrizione delle informazioni contenute nelle sorgenti, utilizzando il linguaggio ODL_{I^3}.

  – in fase di Query Processing, traducono la query ricevuta dal mediatore (espressa in OQL) in una interrogazione comprensibile ed eseguibile dalla sorgente stessa. Inoltre, devono esportare i dati ricevuti in risposta all'interrogazione, presentandoli al mediatore attraverso il modello ODL_{I^3}.

  Collegate ai Wrapper ci sono le sorgenti, per questo spesso si parla di quattro livelli.

- **Livello Mediatore**
  Il mediatore è il cuore del sistema ed è composto da tre moduli, ognuno preposto a funzionalità bel precise.

  – *Global Schema Builder*
  La sua funzione principale è quella di generare lo Schema Globale. Il modulo riceve in input le descrizioni degli schemi locali delle sorgenti espressi in ODL_{I^3} e forniti ognuno dal relativo wrapper. A questo punto (utilizzando strumenti di ausilio quali ODB-Tools Engine, WordNet, ARTEMIS) il Global Schema Builder è in grado di costruire la vista virtuale integrata (**Global Schema**) utilizzando tecniche di clustering e di Intelligenza Artificiale. In questa fase è prevista anche l'interazione con il progettista il quale, oltre ad inserire le regole di mapping, interviene nei processi che non possono essere svolti automaticamente dal sistema (come ad es. l'assegnamento dei nomi alle classi globali, la modifica di relazioni lessicali, . . .).

  – *Extensional Hierarchy Builder*
  Il modulo si occupa della gestione della conoscenza estensionale, calcolando le Base Extension e generando la Gerarchia Estensionale. Come questo avvenga sarà più chiaro nel Capitolo 2.

  – *Query Manager*
  È il modulo di gestione delle interrogazioni. In questa fase la singola query posta in OQL_{I^3}dall'utente sullo Schema Globale (che chiameremo *Global Query*) sarà rielaborata in più *Local Query* (anch'esse

espresse in OQL_{I^3}) da inviare alle varie sorgenti, o meglio ai wrapper predisposti alla loro traduzione, come abbiamo visto. Questa traduzione avviene in maniera automatica da parte del Query Manager utilizzando la conoscenza intensionale ed estensionale definite nella precedente fase di integrazione.

- **Livello Utente**
  L'utente del sistema può interrogare lo schema globale e per lui sarà come interrogare un database tradizionale. La query posta dall'utente sullo schema globale viene passata come input al Query Manager, che interroga le sorgenti e fornisce all'utente la risposta cercata. Tutte queste operazioni, per l'utente, risultano completamente trasparenti.

1.2.3 Tools di supporto

Per realizzare il processo di integrazione degli schemi il sistema mediatore MOMIS sfrutta anche alcuni tool esterni. Questi sono:

- **ODB-Tools**: è uno strumento software sviluppato presso il dipartimento di Ingegneria dell'Università di Modena e Reggio Emilia [30, 31, 32]. Esso si occupa della validazione di schemi e dell'ottimizzazione semantica di interrogazioni rivolte a Basi di Dati orientate agli Oggetti (OODB).
  L'architettura di ODB-Tools, come si vede in Figura 1.4 prevede vari componenti, tra cui:

  – *ODB-Designer* si occupa della validazione di schemi: si può inserire la descrizione di uno schema di database (in ODL) ed il sistema

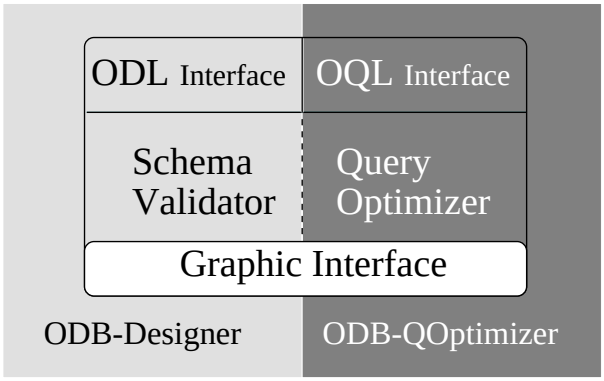

Figura 1.4: Architettura di ODB-Tools

realizzerà automaticamente la sua validazione e la sua riclassificazione (verifica che non vi siano classi incoerenti e calcola relazioni di specializzazione non esplicitate dallo schema).

  - *ODB-Qoptimizer* si occupa dell'ottimizzazione semantica delle interrogazioni: se si inserisce una query (in OQL) posta su di un determinato schema, questa viene automaticamente riformulata in una equivalente, ma più efficiente, sfruttando l'espansione semantica ed i vincoli di integrità.

- **WordNet**: è un DataBase lessicale on-line in lingua inglese. Esso è capace di individuare relazioni semantiche fra termini; cioè dato un insieme di termini, WordNet è in grado di identificare l'insieme di relazioni lessicali che li legano [33].

- **ARTEMIS** (Analisys and Reconciliation Tool Enviroment for Multiple Information Sources) [34]: è uno strumento software sviluppato presso l'Università di Milano e Brescia. Riceve in ingresso il *thesaurus*, cioè l'insieme delle relazioni terminologiche (lessicali e strutturali) precedentemente generate, e sulla base di queste assegna ad ogni classe coinvolta nelle relazioni un coefficiente numerico indicante il suo grado di affinità con le altre classi. Questi coefficienti serviranno per raggruppare le classi locali in modo tale che ogni gruppo (*cluster*) comprenda solo classi con coefficienti di affinità simili.

Capitolo 2

Integrazione degli schemi e Query Processing in MOMIS

MOMIS è stato sviluppato col fine di realizzare un sistema mediatore versatile ed efficiente, che permetta, ad un generico utente, di reperire informazioni su un insieme eterogeneo di sorgenti.
Questo obiettivo viene raggiunto agendo su due fronti. Da un lato sono stati progettati strumenti che, sfuttando la conoscenza intensionale e la conoscenza estensionale, sono in grado di assistere il progettista nella complessa fase di *integrazione degli schemi*, finalizzato alla generazione della *vista globale*. Dall'altro si è realizzato un Query Manager che, posta una interrogazione sulla vista globale ottenuta, automatizza il processo di reperimento ed integrazione delle informazioni. Questa fase prende il nome di *Query Processing*.

2.1 Integrazione degli schemi

Gli schemi locali vengono integrati in MOMIS secondo criteri sia intensionali che estensionali.
Gli aspetti intensionali, a cui si fa riferimento, riguardano il fatto che schemi locali parzialmente sovrapposti possono rappresentare uno stesso concetto adottando strutture diverse. È necessario quindi individuare ed eliminare questi conflitti, provvedendo a definire una rappresentazione unificata ed omogenea dei medesimi concetti descritti nelle varie sorgenti.
L'integrazione intesionale non è però l'unico aspetto che occorre gestire per ottenere un'effettiva integrazione di sorgenti eterogenee. Infatti, come descritto in [2, 35, 36], è necessario risolvere anche conflitti di natura estensionale, ovvero

Sorgente UNIVERSITY (UNI)

```
Research_Staff(name,e_mail,dept_code,section_code)
School_Member(name,faculty,year)
Department(dept_name,dept_code,budget)
Section(section_name,section_code,length,room_code)
Room(room_code,seats_number,notes)
```

Sorgente COMPUTER_SCIENCE (CS)

```
CS_Person(first_name,last_name)
Professor:CS_Person(belong_to:Office,rank)
Student:CS_Person(year,takes:set⟨Course⟩,rank)
Office(description,address:Location)
Location(city,street,number,county)
Course(course_name,taught_by:Professor)
```

Sorgente TAX_POSITION_XML (TP)

```
Student(name,student_code,faculty_name,tax_fee)
ListOfStudent(Student:set⟨Student⟩)
```

Figura 2.1: Esempio di riferimento

derivanti dalle sovrapposizioni delle estensioni, cioè dalla presenza, in sorgenti diverse, di informazioni relative alla stessa entità del "mondo reale".

2.1.1 Esempio di riferimento

Il seguente esempio, che verrà utilizzato per illustrare le fasi di integrazione degli schemi e di Query Processing, fa riferimento alle definizioni degli schemi delle sorgenti espresse in ODL$_{I^3}$e riportate in Appendice C. In Figura 2.1 viene invece presentato in modo schematico per maggiore semplicità.
Esso si riferisce ad una realtà universitaria: le sorgenti da integrare sono tre.

La prima sorgente, `University` (UNI), 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 (`e_mail`), e sul corso da lui tenuto (`section_code`). Per il corso inoltre viene memorizzata l'aula (`Room`) dove questo si svolge, mentre del dipartimento è descritto, oltre al nome (`dept_name`) ed al codice (`dept_code`), il budget (`budget`) che ha a disposizione. Per gli studenti presenti nella ta-

bella `School_Member` sono invece mantenuti il nome (`name`), la facoltà di appartenenza (`faculty`) e l'anno di corso (`year`).

La sorgente `Computer_Science` (CS) contiene invece informazioni sulle persone afferenti a questa facoltà, è un database ad oggetti. Sono presenti sei classi: `CS_Person`, `Professor`, `Student`, `Office`, `Location` e `Course`. I dati mantenuti sono comunque abbastanza simili a quelli della sorgente UNI: per quanto riguarda i professori, sono memorizzati il livello (`rank`), e l'ufficio di appartenenza (`belong_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`), l'anno di corso (`year`) e il livello (`rank`). Il corso ha poi un attributo complesso che lo lega al professore che ne è titolare (`taught_by`), mentre per l'ufficio si tiene l'indirizzo (`address`) e la descrizione (`description`).

È presente infine una terza sorgente, `Tax_Position_xml` (TP), facente capo alla segreteria studenti, che mantiene i dati relativi alle tasse da pagare (`tax_fee`). Alla sorgente in questione appartengono `Student` e `ListOfStudent`. `Tax_Position_xml` è una sorgente semistrutturata.

2.1.2 Integrazione intensionale

Con integrazione intensionale si intende il processo di **Unificazione degli schemi** [8]. L'uso della logica descrittiva OLCD insieme all'uso di tecniche di clustering [37], permettono la realizzazione di una fase semi-automatica di integrazione degli schemi fino a pervenire alla definizione dello *Schema Globale*, direttamente interrogabile dall'utente e che rappresenti l'unione di tutti gli schemi locali, rimuovendone incongruenze e ridondanze.
Il Global Schema Buider è il modulo di MOMIS che si occupa dell'integrazione degli schemi per la generazione dell'unico schema globale da presentare all'utente. Questo modulo, interagendo col progettista, realizza una fase semi-automatica di integrazione, che, partendo dalla descrizione ODL_{I^3}delle sorgenti, porta alla creazione dello schema globale, passando attraverso quattro sottofasi.

Generazione del Thesaurus di relazioni terminologiche

Lo scopo di questa fase è la costruzione del *Common Thesaurus*, ovvero di un tesoro di relazioni terminologiche che rappresenti la conoscenza a disposizione sulle classi da integrare (sui nomi delle classi e sugli attributi). Queste relazioni terminologiche vengono derivate in modo semi-automatico dalla descrizione ODL_{I^3}delle sorgenti attraverso l'analisi strutturale e di contesto delle classi coinvolte. Indicando genericamente con *termine* t_i il nome di una

classe o di un attributo (per identificare in modo univoco una classe, è necessaria la coppia *nome_sorgente.nome_classe*, un attributo, è necessaria la coppia *nome_sorgente.nome_attributo*), si possono definire, all'interno del Thesaurus, le seguenti relazioni:

- SYN (synonym-of): questa relazione è 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 caso di relazione di sinonimia nell'esempio di riferimento: $\langle$`UNI.Section` SYN `CS.Course`$\rangle$.

- BT (broader-term): questa relazione è 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 è $\langle$`UNI.Research_Staff` BT `CS.Professor`$\rangle$.

- NT (narrower-term): concettualmente è la relazione inversa di BT, dunque t_i BT $t_j \rightarrow t_j$ NT t_i. Lo stesso esempio potrebbe infatti essere: $\langle$`CS.Professor` NT `UNI.Research_Staff`$\rangle$.

- RT (related-term): questa relazione è definita tra due termini t_i e t_j che sono generalmente usati nello stesso contesto, tra i quali esiste comunque un legame generico. Un esempio può essere: $\langle$`CS.Student` RT `CS.Course`$\rangle$. La relazione è simmetrica.

L'intero processo che porta, partendo dalle descrizioni degli schemi in ODL$_{I^3}$, alla definizione di un Thesaurus comune, si articola in cinque passi:

1. *Estrazione automatica di relazioni intra-schema*
   Sfruttando le informazioni semantiche presenti negli schemi strutturati, può essere identificato in modo automatico un insieme di relazioni terminologiche (ad esempio relazioni NT e BT derivate dalle gerarchie di generalizzazione).

2. *Estrazione automatica di relazioni inter-schema*
   Le relazioni inter-schema, terminologiche ed intensionali, sono estratte analizzando gli schemi ODL$_{I^3}$nel loro insieme. La loro estrazione è basata sulle relazioni lessicali che sussistono tra nomi di classi ed attributi, derivanti dai significati delle parole usate.
   Le relazioni trovate sono sia inter-schema che intra-schema. Però, se gli schemi sono stati strutturati bene, non dovrebbero venire trovate relazioni intra-schema che non siano già state trovate al passo precedente.

3. *Revisione/Integrazione delle relazioni*
   Il progettista interagisce con il modulo, inserendo esplicitamente tutte le

relazioni terminologiche che non sono state estratte precedentemente, ma che devono comunque essere presenti per pervenire ad una corretta integrazione degli schemi. Inoltre, il progettista ha il compito di inserire gli assiomi estensionali, che descrivono le relazione esistenti tra le estensioni delle classi. Queste relazioni estensionali vincolano le classi che coinvolgono ad avere anche un legame intensionale, che viene registrato nel Common Thesaurus.

4. *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.

5. *Inferenza di nuove relazioni*
   In questa fase vengono inferite nuove relazioni terminologiche utilizzando ODB-Tools, partendo da quelle già introdotte nel Common Thesaurus ed utilizzando le tecniche di inferenza di OLCD. Siccome le relazioni semantiche di equivalenza (SYN) e di generalizzazione (BT) stabilite nel Common Thesaurus tra i nomi di classi, possono entrare in conflitto con le descrizioni strutturali delle classi correlate, si produce uno *schema virtuale* espresso in ODL_{I^3}, che contiene una descrizione degli schemi sorgenti "ristrutturata" sulla base delle relazioni semantiche nel tesoro. Lo schema virtuale costituisce solo uno strumento tecnico per l'inferenza di nuove relazioni e rimane completamente trasparente all'utente.

Analisi delle affinità intensionali tra le classi

In questa fase vengono individuate classi, appartenenti a sorgenti diverse, che descrivono informazioni semanticamente equivalenti. Per questo scopo le classi vengono analizzate e valutate in base al concetto di *affinita'*, in modo da individuare il livello di *similaritá*. In particolare, vengono analizzate le relazioni che esistono tra i loro nomi delle classi (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*.
Il valore numerico assunto da questi coefficienti viene calcolato sulla base del peso (σ_{rel}) che viene associato ad ogni relazione terminologica. Ovviamente questo peso sarà tanto maggiore, quanto più questo tipo di relazione contribuisce a legare due termini. Nelle applicazioni presenti nel sistema MOMIS si è assunto: $\sigma_{syn} = 1$, $\sigma_{bt/nt} = 0.8$, $\sigma_{rt} = 0.5$.
Questa attività è compiuta con il supporto dell'ambiente di ARTEMIS [38]

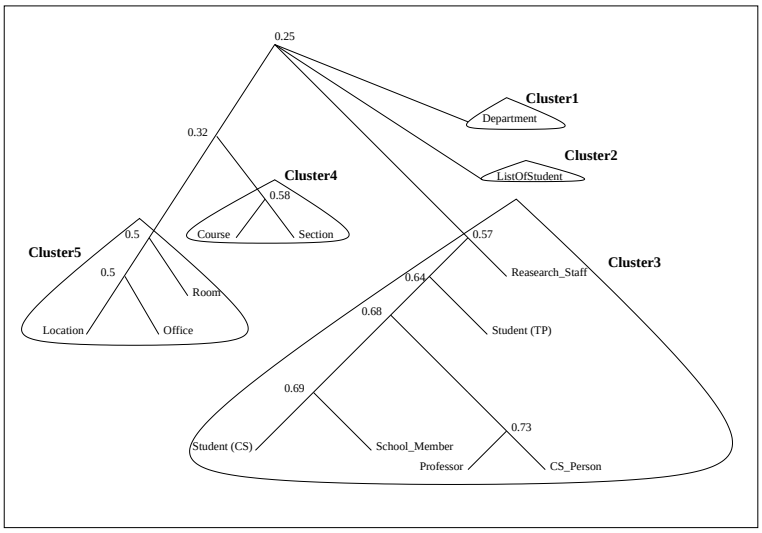

Figura 2.2: Albero di affinità

che , attraverso un processo interattivo, permette di modificare i parametri di valutazione delle affinitá e di validare le scelte fatte.

Creazione dei cluster

In questa fase si creano raggruppamenti di classi affini, ovvero insiemi di classi intensionalmente affini, per le quali si suppone esista anche una qualche sovrapposizione estensionale.
La procedura di clustering adottata opera in modo iterativo, andando a creare insiemi di classi, *cluster* appunto, di dimensioni via via crescenti. Questa operazione viene fatta sulla base dei coefficienti di affinità che legano le classi. Infatti, ad ogni passo viene 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* (Figura 2.2), caratterizzato dall'avere come foglie le classi, come radice l'insieme di tutti i cluster, ed i cui nodi sono proprio i cluster individuati (con valore di affinità calante spostandosi verso la radice).
Dopo aver costruito l'albero di affinità, è necessario selezionare i cluster più appropriati per la definizione delle classi globali. La procedure di selezione dei cluster, in ARTEMIS (modulo preposto a questa fase), viene mantenuta interattiva attraverso la modifica del valore di soglia. Infatti il progettista specifica il valore di una soglia T, ed i cluster che prevedono un Global Affinity Cofficient superiore o uguale a T sono selezionati e proposti. Ovviamente, più il valore di T è alto,

più si ottengono cluster piccoli e molto omogenei tra di loro. In ARTEMIS il valore di dafault per T è 0.5.

Generazione dello Schema Globale del Mediatore

In questa fase, da ogni cluster si costruisce una *classe globale* la cui estensione è costituita dall'unione delle estensioni delle classi sorgenti che costituiscono il cluster, mentre l'intensione è ricavata dall'unione "ragionata" degli attributi delle stesse. Con "ragionata" si fa riferimento al fatto che, nel determinare l'intensione di una classe globale, vengono seguiti i seguenti passi:

- ad ogni $classe\_globale_i$ è associata l'unione degli attributi di tutte le classi appartenenti al $cluster_i$ 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 e vengono riorganizzati all'interno di gerarchie. Per ognuna di queste gerarchie è mantenuto solo il termine più generale, mentre sono rimossi tutti gli altri.

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 prevede l'intervento del progettista, e nella scelta del nome da associare alla classe globale (il tool si limita a proporre un nome), e nella definizione dei *mapping* che caratterizzano la Mapping Table. Queste informazioni servono da un lato all'utente finale per poter utilizzare in modo efficace la vista globale, dall'altro al Query Manager per svolgere in maniera automatica la fase di Query Processing (come verrà spiegato in Sezione 2.3).
I tipi di mapping, tra gli attributi globali e quelli locali, gestiti finora dal sistema sono:

- *mapping semplice*
  L'attributo globale corrisponde ad un singolo attributo locale. L'analisi di relazioni di sinonimia porta alla fusione di più attributi, appartenenti a classi locali diverse, in un unico attributo. Analogamente nel caso di relazioni di

specializzazione che legano tra di loro attributi locali appartenenti a classi locali diverse.

- *and mapping*
  L'attributo globale corrisponde alla concatenazione, in uno specificato ordine, di più attributi locali appartenenti alla medesima classe locale.

- *complex mapping*
  L'attributo globale mappa su altre classi.
  Si ha questo mapping quando l'attributo globale ha come dominio un'altra classe locale, o quando gli attributi locali costituiscono una foreign key.

- *union mapping*
  Descrive la regola di *or mapping* che prevede la sostituzione dell'attributo globale con un solo attributo locale, scelto tra i possibili candidati della classe locale. Questa scelta viene fatta sulla base del valore assunto da un attributo di riferimento (*tag attribute*), appartenente all'insieme di attributi locali della classe locale considerata.

- *default mapping*
  L'attributo locale assume un valore predefinito. Si verifica la necessità di questa operazione (l'esplicitazione del valore viene effetuata dal progettista) quando alcune informazioni sono presenti negli schemi delle sorgenti sottoforma di metadato e il sistema non le considera.

- *null mapping*
  L'attributo globale non ha corrispondenza tra gli attributi locali.

Il processo di unificazione degli schemi applicato all'esempio di riferimento porta all'individuazione di cinque classi globali:

Global Schema **University_Schema**

- University_Person (name,dept,e_mail,section,faculty,year,belong_to,takes,rank,studcode,tax)
- Department (dept_name,dept_code,budjet)
- List (Student)
- Section (section_name,section_code,taught_by,length,room_code)
- Location (description,address,city,number,street,county,room_code,seats_number,notes)

University_Person	name	dept	e_mail	section	faculty	
UNI.Research_Staff	name	dept_code	e_mail	section_code	null	...
UNI.School_Member	name	null	null	null	faculty	...
CS.CS_Person	first_name and last_name	null	null	null	null	...
CS.Student	first_name and last_name	null	null	null	faculty_name	...
CS.Professor	first_name and last_name	null	null	null	null	...
TP.Student	name	null	null	null	null	...

	year	belong_to	takes	rank	studcode	tax
...	null	null	null	"professor"	null	null
...	year	null	null	"student"	null	null
...	null	null	null	null	null	null
...	year	null	takes	rank	null	null
...	null	belong_to	null	rank	null	null
...	null	null	null	"student"	student_code	tax_fee

Figura 2.3: Mapping Table di University_Person

Inoltre, per ognuna di queste classi globali viene definita la corrispondente Mapping Table. La Mapping Table corripondente alla classe globale University_Person e rappresentata in Figura 2.3.
Questa fase si conclude con la realizzazione della descrizione in ODL_{I^3}delle classi globali generate. In particolare, per ogni attributo globale, oltre alla specifica del nome e del tipo, viene aggiunta una serie di *mapping rule* in ODL_{I^3}, attraverso le quali vengono specificate le informazioni su come questo attributo verrà accoppiato con attributi locali, come pure informazioni su valori di default o nulli.
Per la classe globale University_Person, ad esempio, si ha:

```
interface University_Person
(extent Research_Staffers, School_Members, Students, CS_Person,
 Professors, Students
 key name)
{attribute string name
     mapping rule University.Research_Staff.name,
                  University.Scool_Member.name,
                  (Computer_Science.CS_Person.first_name and
                   Computer_Science.CS_Person.last_name),
                  (Computer_Science.Professor.first_name and
                   Computer_Science.Professor.last_name),
                  (Computer_Science.Student.first_name and
                   Computer_Science.Student.last_name),
                  Tax_Position_xml.Student.name)
 attribute string rank
     mapping rule  University.Research_Staff = "Professor",

     ...}
```

2.1.3 Integrazione estensionale

L'integrazione intensionale non è però l'unico aspetto che occorre gestire per ottenere un'effettiva integrazione di sorgenti eterogenee, infatti è necessario risolvere anche conflitti derivanti dalle sovrapposizioni delle estensioni, cioè dalla presenza, in sorgenti diverse, di informazioni relative alla stessa entità del mondo reale.
Con integrazione estensionale si fa riferimento al processo di **Fusione delle istanze** [2], ovvero al processo di *"fusione"* degli oggetti recuperati dalle varie sorgenti, col fine di ricostruire le estensioni delle classi di entità del *"dominio applicativo"*. Il modulo che si occupa di realizzare ciò è l'Extensional Hierarchy Builder. L'approccio seguito in MOMIS si basa sulla teoria della *formal context analysis* [39], che 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 dell realtà. Gli elementi che caratterizzano questo approccio teorico sono i seguenti.

Definizione degli assiomi estensionali

Gli *assiomi estensionali* descrivono le relazioni insiemistiche esistenti tra le estensioni delle sorgenti.

Definizione 1 (Stato di una classe) *Lo stato di una classe* **C**[1] *all'istante t, scritto* $Stato_C^t$*, è costituito dall'insieme degli oggetti che popolano la classe* **C** *all'istante t. Lo stato di una classe viene spesso indicato come "l'estensione della classe".*

Date due classi A e B, sono individuabili quattro tipologie di relazioni estensionali tra di esse:

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

Una parte degli assiomi estensionali viene ricavata direttamente dalla descrizione

[1] C può essere definito come un'espressione logica che coinvolge più classi.

degli schemi. Infatti una relazione ISA viene espressa tramite l'assioma di inclusione. Però la maggior parte degli assiomi estensionali, presenti sulle classi, viene esplicitata dal progettista.
L'analisi estensionale fatta in MOMIS si basa su due presupposti:

1. per classi appartenenti ad uno stesso cluster, e per le quali non è specificata nessuna relazione (e non è possibile ricavare nessuna relazione implicita), si assume che le loro estensioni siano sovrapposte;

2. tra classi appartenenti a cluster diversi si suppone sussista una relazione di disgiunzione estensionale.

Ogni assioma definito *vincola* le classi coinvolte ad avere anche un legame intensionale.
Estendendo la notazione usata per le relazioni intensionali **NT, BT, SYN**, gli assiomi estensionali possono essere definiti in ODL_{I^3}come:

- $A \;\; SYN_{Ext} \;\; B$: le istanze della classe A sono le stesse della classe B. Questo implica una relazione intensionale di tipo SYN tra le due classi e due relazioni ISA;

- $A \;\; NT_{Ext} \;\; B$: le istanze della classe A sono un sottoinsieme di quelle della classe B. Questo assioma viene scomposto in una relazione intensionale di tipo NT e una relazione ISA;

- $A \;\; BT_{Ext} \;\; B$: le istanze della classe A sono un sovrainsieme di quelle della classe B. Viene generata una relazione intensionale di tipo BT ed una ISA tra le classi;

- $A \;\; DISJ_{Ext}$[2] B: le istanze della classe A sono diverse da quelle della classe B. Questo assioma non implica nessun tipo di relazione intensionale, esiste solamente un legame di tipo BOTTOM[3] tra le due classi coinvolte.

L'assioma che definisce la *sovrapposizione estensionale* non necessita di notazione in quanto, come giá specificato, viene considerato di default per le classi appartenenti allo stesso cluster.

In MOMIS le relazioni estensionali vengono espresse come rule nel linguaggio ODL_{I^3}:

- relazione di *inclusione*:
  `rule RE1 forall x in B then x in A`

[2]Questa notazione non era presente nella rappesentazione delle relazioni intensionali ma é stata introdotta per poter rappresentare il concetto di *disgiunzione estensionale*.

[3]Nella logica descrittiva un tipo o classe *bottom* rappresenta un concetto incongruente, cioé che non puó essere in nessun caso popolato da dati o istanze.

- relazione di *disgiunzione*:
  ```
  rule RE2 forall x in (A and B) then x in bottom
  ```

- relazione di *equivalenza*:
  ```
  rule RE3 forall x in A then x in B
  rule RE4 forall x in B then x in A
  ```

Si può notare come le relazioni di sovrapposizione non debbano essere espresse in modo esplicito, questo deriva dal precedente presupposto 1.
Per come sono stati definiti gli assiomi estensionali di tipo SYN_{Ext}, NT_{Ext} e BT_{Ext}, si intuisce che il legame che generano tra le classi coinvolte è molto forte poichè implica sia un legame tra gli schemi che un legame tra le istanze. Per questo motivo, le classi legate da relazioni di questo tipo validate, devono appartenere necessariamente allo stesso cluster. Infatti, le relazioni intensionali implicate dagli assiomi estensionali, vengono registrate nel Common Thesaurus con peso pari ad uno (in modo da forzare le classi locali coinvolte ad appartenere allo stesso cluster).
Le relazioni intensionali implicate dagli assiomi estensionali vengono validate automaticamente quando non determinano inconsistenze nelle descrizioni degli schemi (il progettista ha comunque la possibilità di validarle manualmente). Quindi, risulta evidente che l'ipotesi che afferma che, classi locali appartenenti a cluster diversi sono disgiunte, può, in casi sfortunati, non essere vera. Infatti, la non validazione della relazione estensionale a livello di Common Thesaurus, potrebbe portare classi locali parzialmente sovrapposte a finire in cluster diversi, per il fatto che hanno coefficienti di affinità bassi.
Quindi, la conoscenza estensionale è fondamentale nel processo di determinazione dei cluster (*Arricchimento dei cluster*).

Individuazione delle base extension

Il prerequisito fondamentale per il calcolo delle *base extension* è che tra le sorgenti coinvolte siano stati risolti tutti i conflitti intensionali, relativi ai nomi ed ai tipi degli attributi.
Una base extension rappresenta un sottinsieme 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 di una stessa base extension abbiano lo stesso insieme di proprietà. Le base extension sono individuate dalle relazioni estensionali presenti tra le classi locali e sono caratterizzate dall'avere:

- *estensione*: l'insieme di entità formate dall'intersezione delle classi locali che le compongono;
- *intensione*: l'unione degli attributi globali descritti nelle classi locali dell'insieme.

Uno degli scopi principali dell'introduzione del concetto di base extension è dovuto al fatto di poter considerare una singola istanza componente di una ed una sola base extension, mentre, se consideriamo le singole classi locali, una generica istanza può appartenere a più di una classe.
Chiariamo il concetto di base extension con un esempio. Si supponga che, per la classe globale Univerity_Person dell'esempio di riferimento, siano stati definiti i seguenti assiomi estensionali:

1. `UNI.School_Member` SYN_{Ext} `TP.Student`
2. `CS.Student` NT_{Ext} `UNI.School_Member`
3. `CS.Professor` NT_{Ext} `UNI.Research_Staff`
4. `CS.Professor` $DISJ_{Ext}$ `UNI.School_Member`
5. `UNI.Research_Staff` $DISJ_{Ext}$ `TP.Student`
6. `UNI.Research_Staff` $DISJ_{Ext}$ `CS.Student`

   Analizzando la sorgente `Computer_Science`, dalle relazioni di ereditarietà, si possono ricavare gli assiomi:

7. `CS.Student` NT_{Ext} `CS.CS_Person`
8. `CS.Professor` NT_{Ext} `CS.CS_Person`

L'insieme delle base extension che si ottiene, attraverso la definizione delle rule sopraelencate, è rappresentato in Figura 2.4.
L'insieme delle base extension può avere anche una rappresentazione di natura tabellare (Figura 2.5), dove vengono messe in risalto le singole classi locali componenti. Leggendo la tabella per colonne, si ha: '1', se la classe della riga corrispondente, comprende nella propria estensione la bese extension; '0' in caso contrario. Il procedimento che porta alla individuazione delle base extension è di tipo incrementale: partendo da un insieme non completo di rule estensionali (non si pretende che il progettista sia in grado di specificare, sin dal principio, l'intero insieme degli assiomi estensionali) è possibile calcolare il relativo insieme di base extension, che può essere utilizzato per la deduzione di nuova conoscenza

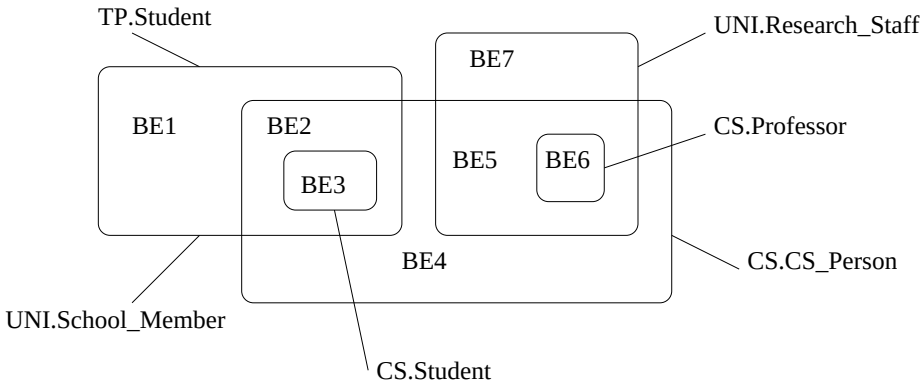

Figura 2.4: Base extension per la classe University_Person

CL \ **BE**	BE1	BE2	BE3	BE4	BE5	BE6	BE7
UNI.School_Member	1	1	1	0	0	0	0
UNI.Research_Staff	0	0	0	0	1	1	1
CS.CS_Person	0	1	1	1	1	1	0
CS.Student	0	0	1	0	0	0	0
CS.Professor	0	0	0	0	0	1	0
TP.Student	1	1	1	0	0	0	0

Figura 2.5: Tabella delle base extension

estensionale, da integrare a quella già sfruttata. Il processo viene ripetuto iterativamente, fino al raggiungimento di una soluzione soddisfacente. Alla base dell'algoritmo che permette di individuare le base extension [40], vi è il concetto di *existence requirement*: un'espressione logica che deve essere soddisfatta da almeno una base extension dell'insieme calcolato in base alle rule estensionali definite. Deve essere verificato in ogni momento la : $State^{t}_{ER} \neq \emptyset$, dove ER è un existence requirement. Occorre naturalmente prevedere un algoritmo che sia in grado di controllare la correttezza degli assiomi specificati dal progettista, e di individuare eventuali incongruenze.

Generazione della gerarchia estensionale

Ad ogni classe globale viene associata una *gerarchia estensionale* (o *concept lattice*), costruita sulla base delle informazioni relative alle base extension, e

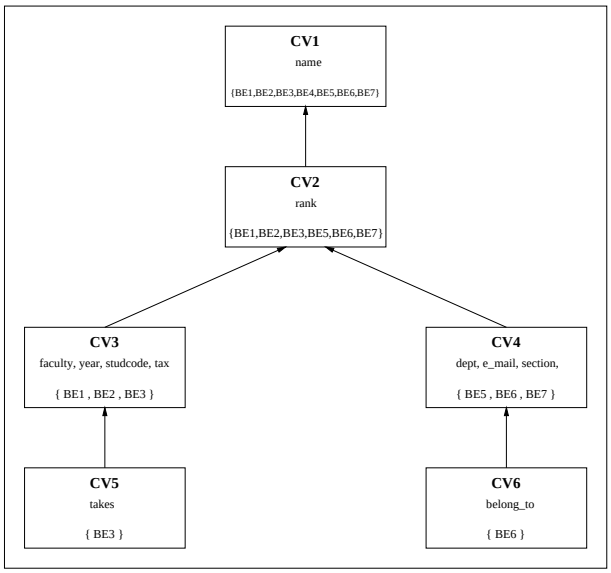

Figura 2.6: Gerarchia estensionale di University_Person

costituita da un insieme di *classi virtuali* totalmente nuove.

Lo scopo principale è quello di individuare le relazioni di specializzazione che legano tra loro le intensioni delle varie base extension di una classe globale. In questo modo, partendo da una query posta dall'utente e quindi da una serie di attributi globali, attraverso la gerarchia estensionale, si riesce a ricavare l'insieme ottimo di base extension alle quali inviare le query. Si ricava in questo modo un insieme di istanze che devono essere combinate attraverso determinate chiavi di join (il modo in cui reperirle è al centro della trattazione di questa tesi e sarà esposto nel Capitolo 3), per arrivare a ricostruire le entità descritte nelle varie sorgenti.

Infatti se ci si limitasse, per ottenere la risposta globale, alle informazioni intensionali presenti nella Mapping Table, si recupererebbero le proprietà richieste dalla Basic Query, ma non si riuscirebbe a ricostruire gli oggetti virtuali rappresentanti le entità descritte in termini globali. La gerarchia estensionale permette il passaggio da un'informazione di tipo intensionale ad una di tipo estensionale. Le classi virtuali che la costituiscono sono caratterizzate dall'avere:

- *intensione*: corrisponde all'intersezione degli schemi delle base extension che le costituiscono;

- *estensione*: corrisponde all'unione di tutte le base extension che hanno almeno tutti gli attributi presenti nell'intensione della classe.

Le varie classi virtuali vengono organizzate in una struttura gerarchica, sulla base di relazioni di ereditarietà, col fine di ottimizzare la procedura di ricerca della *classe virtuale target*. Cosa sia la classe virtuale target e come venga gestita, risulterà più chiaro in Sezione 2.3.
Facendo riferimento alle rule estensionali ed alle base extension individuate in precedenza, per la classe globale University_Person, viene generata la gerarchia estensionale rappresentata in Figura 2.6. Il processo che per ogni classe globale, partendo dalle basse extension, arriva a creare il concept lattice rappresentante la gerarchia estensionale è stato codificato in un algoritmo ed esaurientemente descritto in [2].
Per capire l'utilità della gerarchia estensionale, si consideri la seguente query:

```
select name, e_mail, belong_to
from University_Person
```

Prendendo in considerazione solo la Mapping Table, si nota che nessuna delle classi locali di University_Person possiede entrambi gli attributi `e_mail` e `belong_to`. Sfruttando solamente la conoscenza intensionale fornitaci dalla Mapping Table (Figura 2.3), potremmo arrivare a generare una risposta recuperando `e_mail` da `UNI.Research_Staff` e `belong_to` da `CS.Professor`, ma tale risposta risulterà inevitabilmente incompleta, in quanto ottenuta facendo riferimento a istanze singole che non hanno un criterio di fusione definito. Avendo a disposizione anche la conoscenza estensionale fornita dalla gerarchia estensionale in Figura 2.6, si può pervenire ad una risposta più completa. Dalla gerarchia estensionale si evince infatti, che la classe virtuale $CV6$ ha nella propria intensione tutti gli attributi cercati. L'estensione di questa classe è fornita dalla base extension $BE6$ e quindi fondendo le classi `UNI.Research_Staff`, `CS.CS_Person` e `CS.Professor` (che costituiscono l'estensione della base extension $BE6$), si ottiene la risposta desiderata (priva di duplicazioni, che determinerebbero una risposta errata). La minimalità di tale risposta è dovuta allo sfruttamento delle informazioni derivanti dalle relazioni estensionali introdotte.

2.2 Definizione di un modello di rappresentazione dello schema globale

In questa sezione verrá definita una formalizzazione della conoscenza generata all'interno della fase di integrazione degli schemi.

2.2.1 Schema Globale

Sia **L** un insieme di *nomi di classi locali* (denotati da $L_1, L_2, \ldots$) e **AL** un insieme di *nomi di attributi locali* (denotati da $al_1, al_2, \ldots$). Gli attributi locali di un classe locale sono determinati tramite la funzione $A_L : \mathbf{L} \rightarrow 2^{\mathbf{AL}}$.

Sia **G** un insieme di *nomi di classi globali* (denotati da $G_1, G_2, \ldots$) e **AG** un insieme di *nomi di attributi globali* (denotati da $ag_1, ag_2, \ldots$). Gli attributi globali di un classe globale sono determinati tramite la funzione $A_G : \mathbf{G} \rightarrow 2^{\mathbf{AG}}$.

Definizione 2 (Schema Globale) *Data un insieme **G** ed un insieme **L**, uno* schema globale SG *di **L**, è una funzione* SG

$$SG : \boldsymbol{G} \rightarrow 2^{\boldsymbol{L}}$$

tale che $SG(G_1), SG(G_2), \ldots, SG(G_k)$ *sia un partizionamento di **L**.*

La *Mapping Table* è la struttura dati che contiene tutte le informazioni riguardanti il passaggio dalla rappresentazione globale agli schemi locali, definisce quindi la conoscenza intensionale di una classe globale G. É rappresentata da una matrice, le cui colonne sono gli attributi globali di G, ag, e le righe sono le classi locali di G, L; i suoi elementi rappresentano la corrispondenza fra la classe locale L e l'attributo globale ag.

Definizione 3 (Mapping Table) *Data una classe globale G, una* Mapping Table *di G, MT è una matrice* $||SG(G)|| \times ||A_G(G)||$

$$MT = \begin{pmatrix} m_{11} & m_{12} & \ldots \\ m_{21} & m_{22} & \ldots \\ \vdots & \vdots & \ddots \end{pmatrix}$$

i cui elementi $m_{kh} = MT[L_k][ag_h]$ *possono assumere i seguenti valori:*

- $al_i \in A_L(L_k)$
  mapping semplice: $\exists!\ al_i \in A_L(L_k) : al_i \leftrightarrow ag_h$
- `null`
  null mapping: *not* $\exists\ al_i \in A_L(L_k) : al_i \leftrightarrow ag_h$
- `<costante>`
  default mapping: (*not* $\exists^4\ a_i \in A(c_k) : a_i \leftrightarrow ga_h$) $\vee$ ($\exists\ a_i \in A(c_k) : a_i = ga_h =$ `<costante>`)

[4] In questo caso il valore di default é inserito dal progettista.

- al_1 `and` ... `and` al_M
  and mapping: $\exists\{al_1, \cdots, al_M\} \subseteq A_L(L_k) : \{al_1, \cdots, al_M\} \leftrightarrow ag_h$
- $al_1, \ldots, al_M$
  complex mapping: $\exists\{al_1, \cdots, al_M\} \subseteq A_L(L_k) : \{al_1, \cdots, al_M\} \leftrightarrow ag_h$
- **case** *al* **of** <costante>$_1$ *:* al_1, ... <costante>$_M$ *:* al_M
  union mapping: $\exists\{al, al_1, \cdots, al_M\} \subseteq A_L(L_k)$: *se* $al =$ <costante>$_i$,
  con $i = 1, \ldots, M \Rightarrow al_i \leftrightarrow ag_h$

2.2.2 Base Extension

Informalmente per *Base Extension* si intende un partizionamento dell'insieme complessivo di tutti gli oggetti rappresentati dalle sorgenti: sono sottoinsiemi disgiunti delle estensioni delle classi e sono ottenute dall'intersezione delle stesse. Più formalmente, un *insieme di Base Extension,* scritto $BES_{A1,A2,...,An}$ delle classi $A1, A2, ..., An$ viene definito da una formula booleana in DCF (Forma Canonica Disgiuntiva) sulle variabili $A1, A2, ..., An$. Ogni *minterm* di detta formula rappresenta una singola Base Extension.

Per applicare tale definizione nel contesto di una classe globale costituito da un insieme di classi locali, si parte dalla definizione di *Istanza di una classe globale*.

Definizione 4 (Istanza di una classe globale) *Data una classe globale G ed un dominio **D**, un'istanza di G sul dominio **D** è una funzione $\mathcal{I}$*

$$\mathcal{I} : SG(G) \rightarrow 2^{\boldsymbol{D}}$$

Un'istanza della classe globale G verrà detta *legale* se soddisfa gli assiomi estensionali definiti sulle classi locali $SG(G)$.

Definizione 5 (Base Extension) *Data una classe globale G ed una sua istanza legale $\mathcal{I}$, un* set di base extension *di G rispetto ad $\mathcal{I}$ è una coppia $(\boldsymbol{B}, F)$, dove **B** è un insieme di* nomi di base extension *(denotati da $B_1, B_2, \ldots,$) e F è una funzione*

$$F : \boldsymbol{B} \rightarrow 2^{SG(G)}$$

tale che

1. $\bigcup_{B \in \boldsymbol{B}} F(B) = SG(G)$

2.

$$\bigcup_{B \in \boldsymbol{B}} \Big(\bigcap_{L \in F(B)} \mathcal{I}(L) - \bigcup_{L \in (SG(G) - F(B))} \mathcal{I}(L) \Big)$$

sia un partizionamento di $\mathcal{I}(G)$.

Un set di base extension di una classe globale G viene rappresentato tramite una tabella le cui righe riportano le classi locali della classe globale, cioè $SG(G)$, le colonne riportano le base extension, cioè gli elementi di **B**: una X in corrispondenza dell'elemento della tabella (L, B) indicherà che $L \in F(B)$.

La parte intensionale di una base extension viene determinata a partire dall'insieme di classi locali che la compongono, considerando come attributi della base extension l'unione degli attributi globali che hanno un mapping non nullo in tali classi locali. Formalmente:

Definizione 6 (Attributi di una base extension) *Data una classe globale G, un suo set di base extension $(\boldsymbol{B}, F)$ ed una Mapping Table di G, MT, si definiscono gli attributi di $B \in \boldsymbol{B}$ come segue:*

$$A_{BE}(B) = \{ag \in A_G(G) \mid \exists L \in F(B), MT[L][ag] \textit{è un mapping non nullo}\}.$$

2.2.3 Schema Virtuale e Gerarchia Estensionale

Nella sezione precedente una classe globale è stata caratterizzata tramite il suo set di base extension e ad ogni base extension sono stati associati un insieme di attributi globali. Ora si considera una query sulla classe globale ed il problema che si vuole affrontare è il seguente: quali solo le base extension che occorre considerare per rispondere all'interrogazione, cioè quali sono le base extension che hanno almeno tutti gli attributi specificati nell'interrogazione? Il problema può essere risolto semplicemente controllando per tutte le base extension il rispettivo insieme di attributi. Un'altra possibilità che velocizza la ricerca è quella di creare un *indice* che dato l'insieme di attributi dell'interrogazione restituisca direttamente l'insieme delle base extension interessate. A tale scopo, le base extension di una classe globale vengono raggruppate in *classi virtuali*; una classe virtuale è descritta da un insieme di attributi globali (*intensione*) e da un insieme di base extension (*estensione*) in modo tale che ogni attributo è comune a tutte le base extension e, viceversa, ogni base extension ha tutti gli attributi. Formalmente:

Definizione 7 (Schema Virtuale) *Data una classe globale G ed un suo* set di base extension $BE = (\boldsymbol{B}, F)$*, lo schema virtuale di G rispetto a BE è una tripla $(\boldsymbol{V}, INT, EST)$, dove*

- ***V** è un insieme di* nomi di classi virtuali *(denotati da $V_1, V_2, \ldots, V_k$)*
- $INT : \boldsymbol{V} \rightarrow 2^{A_G(G)}$, intensione *dello schema virtuale*
- $EST : \boldsymbol{V} \rightarrow 2^{\boldsymbol{B}}$, estensione *dello schema virtuale*

tale che

1. $\forall V \in \boldsymbol{V}, \forall ag \in INT(V), \forall B \in EST(V)$ *si ha che* $ag \in A_{BE}(B)$
2. $\forall ag \in A_G(G), \exists V \in \boldsymbol{V} : ag \in INT(V)$
3. $\forall B \in \boldsymbol{B}, \exists V \in \boldsymbol{V} : B \in EST(V)$

Le classi virtuali di una classe globale vengono organizzate in una gerarchia, detta *gerarchia estensionale*, sulla base della relazione di inclusione tra i rispettivi insiemi di attributi:

Definizione 8 (Gerarchia Estensionale) *Dato uno schema virtuale* $(\boldsymbol{V}, INT, EST)$ *di una classe globale G, la* Gerarchia Estensionale *è costruita sulla base della relazione* $ISA_{EXT} \subseteq \boldsymbol{V} \times \boldsymbol{V}$ *tale che,* $\forall V, V' \in \boldsymbol{V}$*:*

$$V \; ISA_{EXT} \; V' \;\; \texttt{iff} \;\; INT(V) \supseteq INT(V')$$

Dalle definizioni date si può verificare che:

$$V \; ISA_{EXT} \; V' \;\; \texttt{iff} \;\; EST(V) \subseteq EST(V')$$

Nella sezione seguente verrà mostrato come lo schema virtuale e la relativa gerarchia estensionale di una classe globale sono utili per l'eleborazione di una query posta dall'utente.

2.3 Query Processing

Il risultato della fase di integrazione degli schemi, eseguito dai due moduli preposti Global Schema Builder e Extensional Hierarchy Builder, è costituito da: lo schema globale, le Mapping Table, le base extension e le gerarchie estensionali.
Il Query Manager, il modulo di MOMIS che si occupa della gestione delle query, utilizza tutte le informazioni, sia intensionali che estensionali, prodotte dalla fase di integrazione degli schemi, per realizzare la fase di *Query Processing*. La fase di Query Processing ha come fine ultimo l'ottenimento di una risposta corretta, e quanto più possibile completa e minima, alla query globale. Questa fase prevede, come è mostrato in Figura 2.7, l'esecuzione in sequenza di tre attività:

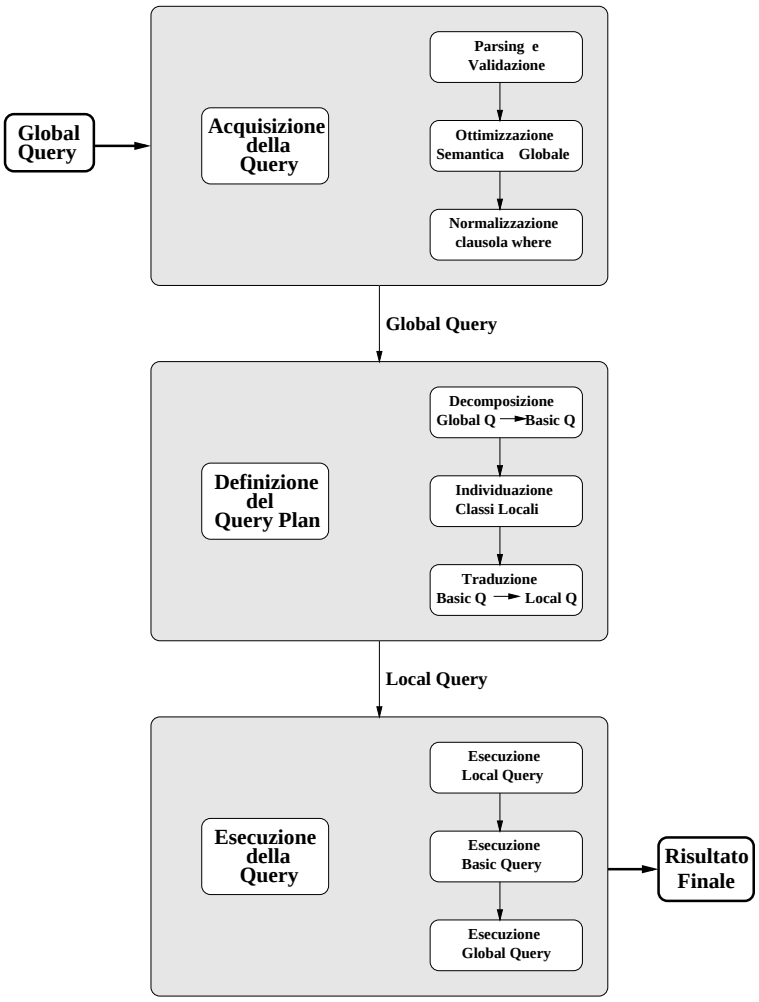

Figura 2.7: Attività di Query Processing

1. *Acquisizione della Query*;
2. *Definizione del Query Plan*;
3. *Esecuzione della Query*.

La prima attività è l'*Acquisizione della Query*, che a sua volta è suddivisa in tre momenti distinti.
Inizialmente avviene la fase di *parsing e validazione*, in cui si verifica la correttezza della query, sia da un punto di vista sintattico che semantico. In questa attività viene impiegato un modulo di riconoscimento grammaticale, che ha il compito di verificare la correttezza sintattica della query rispetto alla grammatica

OQL e di produrne poi un'immagine in memoria centrale. La validazione della query prevede la verifica della correttezza semantica di questa, accertando che le classi e gli attributi coinvolti appartengano effettivamente allo schema integrato a cui la query è rivolta.
In secondo luogo avviene l'*Ottimizzazione semantica globale*. Questa é realizzabile se sono presenti regole di integritá inter-sorgenti definite sullo schema globale. Questi vincoli sono espressi con rule ODL_{I^3}e vengono impiegati da ODB_Tools per riformulare la query producendone una semanticamente equivalente, ma eseguibile in modo piú efficiente. Ad esempio vengono eliminati predicati ridondanti oppure vengono aggiunte nuove condizioni che possono abbreviare i tempi di risposta per la possibile presenza, nelle sorgenti, di indici sui predicati introdotti.
Infine vi è la *Normalizzazione della clausola where*. Alla fine delle due fasi precedenti, la clausola where è rappresentata da un'espressione booleana innestata ricorsivamente. Per poter proseguire, il Query Manager deve trasformare questa generica struttura booleana in forma normale congiuntiva. Avere la clausola where in questa forma, consente di ottimizzare la fase di esecuzione della query, in quanto vengono valutati il maggior numero di predicati in and e quindi viene minimizzata la quantità di risultati restituiti dalle sorgenti locali, con una conseguente velocizzazione della fase di integrazione.

La seconda attività svolta dal Query Manager nella fase di Query Processing, è la *Definizione del Query Plan*, in cui la query posta in termini globali viene ricondotta agli schemi locali delle singole sorgenti. In questa fase viene generato il *plan*, che contiene le informazioni per generare le query locali e per ricostruire la risposta globale.
Il primo passo che viene svolto è la decomposizione della Global Query (espressa sull'intero schema globale) in Basic Query, ovvero in una query contenente tutte le richieste rivolte ad una singola classe globale. In una Basic Query non sono ammessi: join espliciti tra classi diverse (è però possibile la navigazione attraverso aggregazioni ed associazioni per ricostruire oggetti complessi), subquery, operatori di ordinamento o di conversione, restituzione di strutture complesse. In questa fase di decomposizione, viene anche creata la *Global Assembler Query*, che si occupa di generare il risultato finale della Global Query, partendo dai risultati parziali delle singole Basic Query.
Arrivati a questo punto la Basic Query viene analizzata, per ricercare gli attributi globali contenuti nei predicati di proiezione e di selezione. Il passo successivo consiste nell'esplorare la gerarchia estensionale alla ricerca delle *Classi Virtuali Target*. Queste classi sono le classi virtuali più generali che dispongono di tutte le proprietà richieste. Quando le classi virtuali target individuate sono più di una, si seleziona la più generalizzata (cioè con estensione maggiore, in termini

di numero di base extension). Essendo note le base extension interessate dalla query, sono automaticamente note le classi locali a cui rivolgersi per ottenere i dati.
Una volta determinati l'insieme delle base extension e l'insieme delle classi locali interessati, si procede ad una semplificazione (eliminazione di base extension,eliminazione di classi locali) del Query Plan, per minimizzare il numero di query da inviare alle sorgenti. Questa semplificazione si attua nei casi in cui si verificano delle dominazioni tra base extension, e nei casi in cui si hanno relazioni estensioni particolari (equivalenza o specializzazione, in cui nessuna delle due classi coinvolte determina un contributo informativo maggiore) tra classi locali.
Noto l'insieme minimo di classi da interrogare, si procede, con l'ausilio della Mapping Table, alla generazione delle Query Locali. Il Query Manager esprime le query locali in OQL, lasciando ai wrapper l'incombenza di tradurle nel linguaggio adatto in ogni sorgente.
In questa fase di generazione delle query locali, è necessario però considerare anche quei predicati che non sono mappati interamente in nessuna classe. Per questo motivo ad ogni Basic Query viene associata una struttura detta *Basic Query Assembler*, che contiene all'interno della propria clausola where i predicati "esclusi". Il Query Manager, dopo aver concluso la fase di *Esecuzione della Query* (che verrà descritta di seguito) ed aver quindi integrato i risultati provenienti dalle sorgenti locali, dovrà eseguire su di essi la Basic Query Assembler per ottenere la risposta desiderata.
Il piano d'accesso ottenuto fino ad adesso, può subire dei miglioramenti. Questi miglioramenti si possono ottenere analizzando eventuali predicati di selezione della Basic Query, che contengono valori di default. Un'altra semplificazione è ottenibile nel caso in cui sia possibile raggruppare più Local Query da inviare ad una stessa sorgente, effettuando localmente i join.
Inoltre è possibile realizzare l'*ottimizzazione semantica locale* (che consente di realizzare query meno onerose), sfruttando, tramite l'impiego di ODB-Tools, le regole di integrità definite sulle singole sorgenti. Quest'ultima fase di semplificazione è posta a livello di mediatore, in quanto non tutte le sorgenti potrebbero essere in grado di realizzarla.

La terza attività che costituisce il *Query Processing* è l'*Esecuzione della Query* [27], dove il Query Manager utilizza le informazioni del plan, generato nelle fasi precedenti, per ricomporre i risultati ottenuti dalle sottoquery, presentando all'utente una risposta integrata. Questa fase prevede tre livelli: esecuzione delle Local Query, esecuzione delle Basic Query ed esecuzione delle Global Query.
Per realizzare questa attività il Query Manager sfrutta un DBMS, che gli permette di creare delle tabelle in grado di memorizzare i risultati temporanei degli stadi intermedi della ricostruzione della risposta. Così il Query Manager può eseguire

varie operazioni di ricostruzione e fusione attraverso query (join e outer join) poste su queste tabelle temporanee.
Come effettivamente il Query Manager realizzi l'*Esecuzione della Query* sarà esposto nella Sezione 3.5. In particolare, si presterà maggiore attenzione alle fasi di ricostruzione e fusione, mettendo in evidenza, essendo l'argomento di questa tesi, quali siano le informazioni sulla base delle quali sia possibile realizzare le operazioni di join ed outer join.

Capitolo 3

Il problema dell'Object Fusion

Il sistema mediatore MOMIS è stato studiato e sviluppato con il fine di permettere ad un generico utente di effettuare interrogazioni, e di conseguenza di raccogliere informazioni, in un contesto eterogeneo e distribuito.
Perchè ciò sia possibile è necessario risolvere conflitti intensionali, fornendo una rappresentazione unificata ed omogenea degli stessi concetti modellati in sorgenti diverse. Inoltre, è indispensabile risolvere conflitti dovuti a sovrapposizioni di carattere estensionale, ovvero è necessario riconoscere quando e dove si ha la presenza di informazioni relative alla stessa entità del *mondo reale* in classi ed in sorgenti diverse.

In questo capitolo verranno analizzate le problematiche inerenti la gestione della conoscenza intensionale ed estensionale, nel processo di riconoscimento e di ricostruzione delle informazioni facenti riferimento alla medesima entità: il processo di **Object Fusion**.

3.1 Le relazioni estensionali

Prima di procedere all'analisi delle problematiche che riguardano le sovrapposizioni fra istanze, è indispensabile fare luce sulla differenza che intercorre tra *Oggetti* di un database e *Entità* di un determinato contesto applicativo.

Osservando un certo contesto applicativo, si possono isolare entità caratterizzate dall'avere un particolare insieme di proprietà e di comportamenti. Queste entità esprimono concetti ben definiti del mondo reale, i quali però possono essere modellati in modi diversi (prestando più attenzione a certi aspetti piuttosto che ad altri) ognuno dei quali corretto.
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, aspetti differenti della stessa entità. Quindi l'entità è un concetto astratto che prescinde da una particolare rappresentazione, mentre l'oggetto è uno strumento di modellazione che, sfruttando una determinata struttura, ne "fotografa" alcuni aspetti.
Arrivati a questo punto, risulta chiaro che sorgenti diverse possono contenere oggetti corrispondenti alla stessa entità, e questi oggetti possono avere proprietà comuni tra di loro, ma anche proprietà completamente nuove, esclusive per il singolo oggetto.
Quindi il processo di integrazione non si deve limitare al reperimento degli oggetti cercati dalle singole sorgenti, ma deve coincidere con la ricomposizione delle entità di cui questi oggetti sono una modellazione.
Perchè ciò sia possibile è necessario definire quali siano le relazioni tra le estensioni e le intensioni delle classi coinvolte, e soprattutto come queste relazioni permettano al sistema mediatore di identificare univocamente le informazioni reperite da ricomporre. Solo in questo modo è possibile ricostruire l'entità a cui le varie informazioni fanno riferimento.

Per capire meglio quale sia la natura della problematica di cui si sta parlando, è d'aiuto il seguente esempio.
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 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 informazione (la stessa persona sarebbe presente più volte), e l'incompletezza della risposta (le persone duplicate hanno un

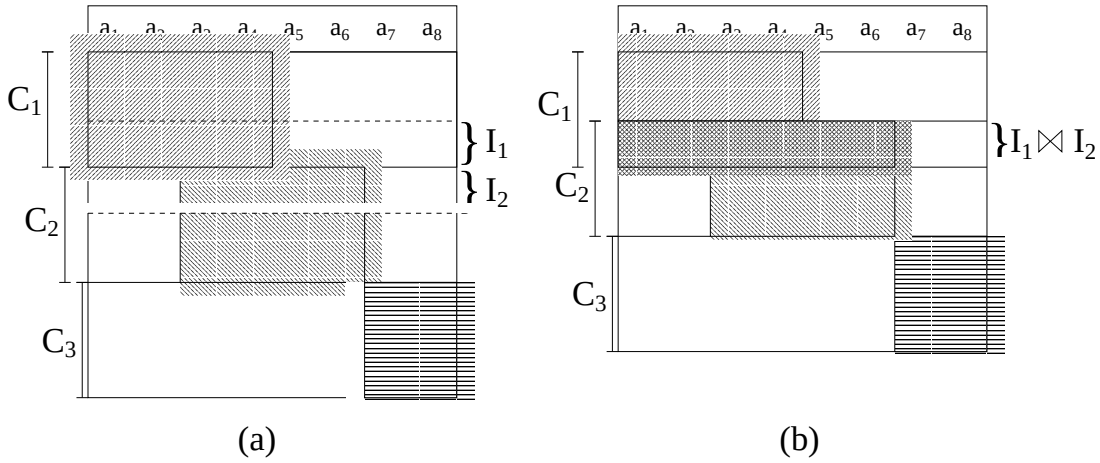

Figura 3.1: Estensione della classe di entità

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, un sottoinsieme delle loro istanze, pur avendo attributi diversi ed appartenendo a sorgenti differenti, individua le 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.
Appurata la necessità di una "$fusione$" in fase di ricostruzione della risposta globale, a partire dalle singole risposte locali, è necessario stabilire come questa debba essere realizzata e sulla base di che cosa.
Nel prosieguo della trattazione si cercherà di mettere in luce gli aspetti salienti che portano a questo processo e lo rendono possibile.

3.2 Identificazione delle istanze

Una volta ricevuti i dati richiesti dalle sorgenti in risposta alle varie query locali a loro inviate, si pone il problema di come riorganizzare queste informazioni per presentarle all'utente.
Il problema che si presenta è il riconoscimento da parte del sistema mediatore di tutte le informazioni che fanno riferimento alla stessa entità del mondo reale. Risulta evidente che ciò di cui si ha bisogno è un'opportuna identificazione degli oggetti reperiti dalle basi di dati sorgenti. Questo è necessario perchè il sistema

mediatore MOMIS non è proprietario degli oggetti che reperisce dalle sorgenti locali e che poi gestisce opportunamente, ma ne costituisce solamente una vista.

3.2.1 Tipi di identificazione

Una opportuna identificazione degli oggetti persistenti nelle basi di dati consente il riconoscimento delle istanze uguali nei valori, ma distinte come entità del mondo reale e di reperirle in maniera automatica.

A seconda del sistema di basi di dati con cui si ha a che fare, si possono incontrare varie metodologie di identificazione delle entità modellate.
Nei **sistemi relazionali**, le tuple presenti sono considerate come set di valori, quindi per distinguere tuple uguali nei valori, ma distinte come istanze si fa ricorso all'uso di *chiavi*, altrimenti dette UID (User-defined IDentifier). Le chiavi devono essere definite come tali esplicitamente dal progettista. La scelta di questi campi identificativi impone un'approfondita conoscenza semantica e un'accurata valutazione in fase di progettazione.
In particolare, un'identificatore del tipo appena descritto deve rispondere positivamente alle seguenti caratteristiche: deve essere unico per ogni possibile tupla legale del database; non deve essere nullo (o parzialmente nullo nel caso di chiave composta) onde allontanare l'eventualità di una perdita dei riferimenti incrociati; qualora l'*uid* scelto fosse di tipo numerico, la sua gestione dovrebbe essere completamente trasparente all'utente, sebbene l'*uid* stesso non lo sia e possa essere recuperato in qualsiasi momento.
Nei **sistemi ad oggetti** l'identificazione delle istanze presenti avviene tramite gli OID (Object-IDentifier). Questi identificatori alfanumerici vengono assegnati agli oggetti in modo univoco al momento della loro istanziazione e restano tali, senza mutare, durante tutta l'esistenza degli oggetti in questione. La gestione e la natura degli *oid* rimane completamente trasparente all'utente.

3.2.2 Possibili approcci

Le strade che si possono battere nella ricerca di una via che possa permettere una identificazione completamente automatica delle istanze nell'ambito di un sistema mediatore sono tante e varie.

Si potrebbe decidere di fare riferimento all'*oid* degli oggetti.
Qualora tra le sorgenti da integrare ve ne fossero di quelle che non prevedono un'identificazione basata su *oid*, il Wrapper delegato, esportando gli elementi dalle sorgenti, ne genererebbe uno casuale run-time.
Sono evidenti i problemi a cui si va incontro con un'approccio di questo tipo.

Infatti, due oggetti provenienti da due sorgenti/Wrapper diversi potrebbero avere il medesimo *oid*, ma fare riferimento a due distinte entità del mondo reale. Al tempo stesso due oggetti con *oid* diversi potrebbero essere istanze collegate e fare parte della medesima informazione.

Un'altra soluzione potrebbe essere quella di riunificare le risposte ricevute inerenti alla medesima informazione, sulla base di campi chiave a livello locale (qualora le chiavi fossero dichiarate). Anche questa soluzione però rimane parzialmente inefficace, come dimostrano i seguenti due casi.
Come primo esempio, si faccia riferimento a database che contengono informazioni su persone. Inoltre, si supponga che nelle tabelle a cui le varie query sono state rivolte vi sia una chiave, e che questa chiave sia il campo `nome`.
Nulla assicura al sistema che persone con nomi uguali, in sorgenti diverse siano la stessa persona. Si potrebbero verificare molto facilmente casi di omonimia, che in fase di fusione porterebbero a risultati errati.
Un'altro esempio che mostra ancora meglio l'inefficacia di questo approccio è quello che potrebbe riguardare il contesto applicativo di un insieme di magazzini. Si supponga che, nelle varie tabelle delle sorgenti da integrare, la chiave identificativa sia il campo `codice`. È molto probabile che si usino codici diversi per identificare lo stesso oggetto, anche se il dominio degli attributi `codice` è il medesimo in tutte le sorgenti. Si ricade insomma nelle stesse problematiche che riguardano l'identificazione mediante *oid*.

Infine, un altro approccio, che dal punto di vista teorico sembra la soluzione ideale, è quello che prevede la presenza di una chiave universale o di un *oid* comune, condivisi da tutte le sorgenti (o almeno da tutte le sorgenti che fanno parte del sistema), che individui univocamente al loro interno tutti gli oggetti.
Anche questa soluzione, ad un'analisi più attenta, mostra qualche problema. Infatti, è difficile pensare di poter realizzare concretamente questo tipo di identificazione, perchè anche nella migliore delle ipotesi, questa chiave sarebbe comunque valida e unica solo all'interno di un contesto limitato. Infatti si potrebbe prendere come "*chiave universale*" il `codice fiscale`, che perde però significato all'esterno di una determinata nazione.

Nel definire l'approccio da seguire per MOMIS nella soluzione del problema dell'Object Fusion, quello che si è fatto non è stato allontanarsi dalle metodologie di identificazione descritte in precedenza (in quanto non completamente efficaci), bensì cercare in esse spunti ed idee da poter adattare opportunamente alle conoscenze ed alle implementazioni presenti nel nostro sistema mediatore.
In particolare, si è deciso di sfruttare, per identificare le varie istanze gestite dal sistema mediatore, le chiavi definite (quando questo avviene) a livello locale nelle

varie sorgenti. Questa conoscenza, che si deve considerare di carattere intensionale, viene poi arricchita dalla conoscenza estensionale rappresentata dagli assiomi estensionali. Infatti, le informazioni apportate dagli assiomi estensionali hanno permesso di formulare alcune ipotesi, che hanno rafforzato il potere identificativo delle chiavi locali. Come questo avvenga risulterà più chiaro nella seguente sezione.

3.3 Le regole di join

Il problema dell'Object Fusion consiste nel determinare uno o più identificatori semantici che consentano di individuare le entità del dominio in modo univoco. Questa identificazione rende poi possibile la ricostruzione delle informazioni che può avvenire tramite una serie di operazioni di join basati sull'uguaglianza di tali campi chiave. Come effettivamente avvenga questa operazione di join risulterà più chiaro in Sezione 3.5. Quello che si è cercato di fare è stato isolare, per ogni coppia di classi locali interessata, una serie di situazioni in cui la conoscenza estensionale ed intensionale a disposizione del sistema mediatore rendesse automatico il processo di identificazione delle istanze. Questo, di conseguenza, rende automatica l'operazione di join impiegata nel processo di Object Fusion.
Anticipo subito che la casistica ricavata (che è stata organizzata come una serie di regole), che si è dimostrata automatica, non ricopre tutte le situazioni in cui ci si può imbattere. Risulta evidente quindi che l'interazione con il progettista, nel processo di identificazione delle istanze finalizzato all'Object Fusion, è essenziale. Di conseguenza, è richiesto da parte di questi una profonda conoscenza della semantica sia del contesto applicativo, sia delle singole sorgenti.

3.3.1 Reperimento della conoscenza

Nel processo che ha portato alla formulazione delle *Regole di join* si è fatto uso di informazioni semanticamente significative e da un punto di vista estensionale e da un punto di vista intensionale.
In particolare le informazioni di cui si sta parlando sono le seguenti:

- Relazioni di tipo SYN_{Int} validate tra attributi locali, presenti nel Common Thesaurus.
- Relazioni di tipo BT_{Int}, NT_{Int} validate tra attributi locali, presenti nel Common Thesaurus. Queste relazioni devono portare alla formulazione, per gli attributi locali coinvolti, di un medesimo attributo globale nella Mapping Table.

- Legame tra attributi locali ed attributi globali, presenti nella Mapping Table.
- Assiomi estensionali definiti dal progettista sulla base di conoscenze a priori.
- Definizione degli schemi che implicano relazioni di specializzazione tra classi o associazioni con vincolo di integrità referenziale.
- Selezione, tra i campi delle classi locali, di quelli definiti chiave.

Risulta evidente che le informazioni fornite dai primi tre punti concorrono ad individuare attributi locali che sono mappati da un medesimo attributo globale. In questo senso si farà tesoro nella formulazione delle regole di mapping rappresentati nella Mapping Table. Questo è reso necessario anche dal fatto che i campi chiave identificativi, che bisogna individuare, devono essere di natura globale. Infatti il modulo di MOMIS preposto all'esecuzione del processo di Object Fusion, il Query Manager, sfrutta nell'operazione di fusione delle informazioni gli attributi globali. Sarà suo compito risalire ai corrispondenti campi locali. Come ciò avvenga sarà accuratamente spiegato in Sezione 3.5.
Per quanto concerne l'ultimo punto elencato le situazioni che si possono presentare sono sostanzialmente tre:

- Chiavi semanticamente omogenee e che quindi corrispondono al medesimo insieme di attributi globali presenti nella Mapping Table.
- Chiavi semanticamente disomogenee e che quindi non corrispondono al medesimo insieme di attributi globali presenti nella Mapping Table.
- Assenza di uno o più campi chiave (la definizione di campi chiave in sorgenti ad oggetti, semistrutturate o file è opzionale).

Fondamentale è sottolineare che l'ambito in cui sono state formulate le *Regole di join* è quello di una medesima classe globale. Questo perchè, per come le classi globali sono state costruite, entità affini appartengono ad una medesima classe globale e classi globali diverse non hanno entità in comune.
Questa asserzione, come sarà meglio chiarito alla fine del capitolo, può subire delle modifiche, come si è reso evidente in fase di realizzazione della tesi. Tali modifiche aprono nuovi orizzonti alle rule che realizzano il reperimento delle informazioni per l'Object Fusion e lasciano spazio a nuove prospettive di ricerca, che influenzeranno l'intero processo di integrazione degli schemi.

3.3.2 Regole di join in linguaggio naturale

Date due classi locali appartenenti alla medesima classe globale, è possibile, sfruttando la conoscenza descritta nella sezione precedente, formulare una serie di regole che permettono di capire quando sia necessario effettuare una fusione tra determinate istanze delle due classi, e soprattutto sulla base di che cosa realizzare tale operazione. È importante notare che quando si parla di fusione si fa riferimento all'operazione di join tra due classi: tale operazione basa la sua corretta esecuzione sull'uguaglianza di determinati campi identificativi delle istanze. Tali campi semanticamente identificativi sono le *chiavi*.

Chiavi semanticamente omogenee

Entrambe le classi coinvolte sono in possesso di un insieme di campi chiave dichiarati. Inoltre questi campi chiave sono legati da una relazione di sinonimia (SYN_{Int} nel Common Thesaurus) che porta alla definizione di un medesimo attributo globale mappato nella Mapping Table.
Ad esempio nella classe globale University_Person dell'esempio di riferimento si ha la seguente situazione:

`UNI.School_Member.name` SYN_{Int} `TP.Student.name`

Supponiamo che gli attributi locali `name` siano chiave nelle due classi. Questi attributi sono legati da una relazione di sinonimia (come è stato specificato sopra) e sono mappati dal medesimo attributo globale `name`. Allora abbiamo che `School_Member.name` e `Student.name` sono due chiavi semanticamente omogenee.

Un caso analogo si presenta quando i campi chiave sono legati da una relazione di specializzazione (NT_{Int}/BT_{Int} nel Common Thesaurus) che porta alla definizione di un medesimo attributo globale nella Mapping Table, ricavato tramite un And Mapping.
Ad esempio, sempre nella classe globale University_Person, si può trovare:

`CS.Student.first_name` NT_{Int} `UNI.School_Member.name`
`CS.Student.last_name` NT_{Int} `UNI.School_Member.name`

La coppia di attributi locali `Student.first_name` e `Student.last_name` è chiave per la classe locale `CS.Student`. Tali attributi vengono mappati in and nell'attributo globale `name`, lo stesso in cui è mappato l'attributo locale `School_Member.name`, chiave per la classe locale

`UNI.School_Member`. Le chiavi delle due classi locali, anche in questo caso, sono semanticamente omogenee.
Combinando questa conoscenza intensionale con la conoscenza fornita dagli assiomi estensionali si possono formulare, per ogni coppia di classi locali appartenenti alla medesima classe globale, le seguenti regole:

1. Sulle due classi locali è definito un assioma di equivalenza o di inclusione, o vi è una relazione di specializzazione. Questa conoscenza estensionale mette in luce la presenza di istanze in comune tra le due classi, e quindi la necessità di effettuare una fusione tra le informazioni delle due classi.
   Siccome gli assiomi estensionali sono definiti dal progettista, è plausibile supporre che tale conoscenza sulle estensioni gli derivi proprio dall'analisi delle chiavi. Siccome poi quest'ultime sono semanticamente omogenee, esse rappresentano proprio la medesima informazione. Allora, è logico individuare in queste chiavi i campi significativi su cui effettuare il join.
   Di questo rafforzamento del potere identificativo delle chiavi dato dagli assiomi estensionali si terrà conto durante tutta la trattazione delle *Regole di join*.

2. Tra le due classi è definito un assioma di disgiunzione, allora non vi è la necessità di trovare una via per effettuare il join, in quanto le due classi non hanno nessuna informazione in comune, ovvero facente riferimento alla medesima entità del mondo reale.

3. Tra le due classi non è stato enunciato nessun assioma estensionale. Allora è necessario ricercare se su di esse non vi sia una relazione estensionale implicita.
   È possibile che tra due classi vi sia una relazione di inclusione implicita, come risulta chiaro dal seguente esempio. Si supponga che siano definiti i seguenti assiomi estensionali:

   `UNI.School_Member` SYN_{Ext} `TP.Student`
   `CS.Student` NT_{Ext} `UNI.School_Member`

   Anche se non viene dichiarato esplicitamente, tra le classi `CS.Student` e `TP.Student` è implicita la seguente relazione estensionale:

   `CS.Student` NT_{Ext} `TP.Student`

In questo caso, e anche nel caso in cui la relazione estensionale implicita sia di equivalenza, per la scelta dei campi significativi per realizzare il join, si procede come descritto al punto 1.
La relazione estensionale che rimane implicita può essere anche di tipo disgiuntivo, come chiarisce il seguente esempio.

`UNI.Research_Staff` $DISJ_{Ext}$ `UNI.School_Member`
`UNI.School_Member` SYN_{Ext} `TP.Student`

Allora è iplicita la seguente relazione estensionale:

`UNI.Research_Staff` $DISJ_{Ext}$ `TP.Student`

In questo caso, come spiegato al punto 2, non vi è necessità di trovare campi di join, in quanto quest'ultimo non viene effettuato.
Se non si trova nessuna relazione estensionale tra le due classi, nè esplicita nè implicita, allora, per come è costruita una classe globale, tra le due classi vi è una relazione di sovrapposizione.
In questo caso però le informazioni a disposizione del sistema non sono sufficienti per stabilire automaticamente i campi su cui effettuare il join. Il fatto che tra le due classi vi sia solo una relazione di sovrapposizione (vi sono entità modellate nella prima classe che non sono modellate nella seconda, e viceversa), fa sì che si possa verificare il caso in cui chiavi uguali identificano entità diverse. Esemplare sarebbe il caso in cui si verificasse una situazione di omonimia.
È allora necessario un intervento da parte del progettista, che può consistere o in una validazione delle due chiavi semanticamente omogenee, o in una scelta alternativa sui campi su cui effettuare il join.

Chiavi semanticamente disomogenee

Qualora tra le due classi condiderate non si verifichino le condizioni di omogenità tra chiavi appena descritte, le considerazioni da fare sono in parte diverse e portano alla formulazione delle regole qui di seguito.

1. Si supponga di avere a che fare con le classi locali A e B. Tra A e B è stato formulato un assioma di inclusione:

classe A NT_{Ext} classe B

Allora tra le due classi si verifica la necessità di effettuare il join.
Tra i campi, non dichiarati come chiave, della classe A ve ne è uno che risulta essere sinonimo[1] del campo chiave della classe B. Allora il campo non chiave di A è implicitamente una chiave alternativa non dichiarata, e come tale, può essere usata per effettuare il join tra A e B. Per meglio chiarire la dinamica di questa scelta, si consideri il seguente esempio.
Sia definito l'assioma estensionale di inclusione:

`TP.Student` NT_{Ext} `CS.Student`

La classe locale `TP.Student` prevede come chiave identificativa il campo `student_code`, mappato dall'attributo globale `studcode`; la classe locale `CS.Student` prevede come campi chiave identificativi la coppia di attributi locali `first_name` e `last_name`, mappati in and nell'attributo globale `name`. Le due chiavi sono evidentemente semanticamente disomogenee.
Però tra i campi non dichiarati chiave di `TP.Student`, vi è il campo `name`, mappato dall'attributo globale `name`. Tra i campi `CS.Student.first_name` e `TP.Student.name`, ed i campi `CS.Student.last_name` e `TP.Student.name` vi è una relazione intensionale di tipo NT_{Int} che porta alla formulazione per tutti del medesimo attributo globale. Quindi `TP.Student.name` risulta essere semanticamente omogeneo ai campi che costituiscono la chiave di `CS.Student`, individua una chiave alternativa per `TP.Student` e pertanto l'attributo globale corrispondente `name` può essere utilizzato per effettuare il join tra le due classi coinvolte `TP.Student` e `CS.Student`. Non vale invece il viceversa, nel senso che se fosse:

`CS.Student` NT_{Ext} `TP.Student`

il campo `name` non potrebbe venire scelto in maniera automatica per effettuare il join. Infatti, in questo caso nulla garantisce che non si verifichino casi di omonimia all'interno di `TP.Student`, non essendo `TP.Student.name` inizialmente dichiarato chiave.

[1] Col il termine sinonimo si intende una situazione di omogeneità semantica, come descritta nel caso di chiavi semanticamente omogenee. Durante il resto della trattazione delle Regole di join si sfrutterà questo termine.

Nel caso particolare di chiavi composte, qualora si verifichi la situazione in cui la chiave (considerando già gli attributi globali corrispondenti) della classe B è contenuta nella chiave della classe A (sempre tenendo presente l'assioma di inclusione definito all'inizio della trattazione), allora i campi che individuano la prima chiave possono essere sfruttati per l'esecuzione del join. Ovvero sia:

K_A = (at_1,at_2) attributi globali corrispondenti alla chiave di A
K_B = (at_1) attributi globali corrispondenti alla chiave di B

Allora il join può essere effettuato sul campo: at_1

Qualora non si verifichi nessuna di queste situazioni, allora è necessario l'intervento esplicito del progettista per selezionare i campi delle due classi su cui effettuare il join, e soprattutto per dichiarare il legame tra i valori da essi assunti.

2. Siano date ancora le classi locali A e B, e sia definito un assioma di equivalenza tra di esse:

   classe A SYN_{Ext} classe B

   Se nella classe B vi è un campo non chiave sinonimo del campo chiave della classe A, allora questo campo può essere selezionato come campo identificativo su cui effettuare il join. Analogamente, se nella classe A vi è un campo non chiave sinonimo della chiave dichiarata nella classe B, allora questo campo può essere utilizzato per realizzare il join.
   Come già visto nel caso di relazioni di inclusione, qualora si presenti la necessità di gestire chiavi composte, se una delle due chiavi include l'altra (considerando gli attributi globali corrispondenti), allora la chiave inclusa può essere utilizzata per realizzare il join.
   Nel caso sfortunato in cui nessuna di queste eventualità si presenti, l'intervento del progettista è inevitabile.

3. Se tra le due classi locali è definito un assioma estensionale di disgiunzione, non vi è necessità di realizzare il join e quindi di ricercare su quali campi effettuarlo.

4. Nel caso in cui tra le due classi locali in analisi non sia definito esplicitamente nessun assioma estensionale, è indispensabile produrre una ricerca

che stabilisca se tra di esse non vi sia una relazione implicita. Se viene ritrovata una relazione implicita, allora si deve procedere, a seconda della relazione trovata, come mostrato ai punti 1, 2 e 3.
Qualora non venga trovata alcuna relazione estensionale implicita, le due classi risultano sovrapposte per l'appartenenza alla medesima classe globale. Come nel caso di chiavi semanticamente omogenee in questa situazione l'intervento del progettista è indispensabile.

Assenza di chiavi

Siccome solo nel caso di sistemi di basi di dati di tipo relazionale, la dichiarazione di chiavi identificative per le entità modellate è obbligatoria, è possibile trovarsi di fronte a classi sprovviste di chiave. In particolare, data una coppia di classi locali, l'assenza di campi dichiarati chiave può interessare una sola classe o entrambe. Le considerazioni da fare in questo caso riprendono in parte le considerazioni fatte per le chiavi semanticamente disomogenee, e portano ad enunciare le seguenti regole.

1. Siano date le due classi locali A e B, e su di esse sia definito un assioma estensionale di equivalenza. Si supponga che solo la classe A sia sprovvista di chiave. In questo caso, come avviene nel caso di chiavi semanticamente disomogenee, si ricerca tra i campi della classe A un campo sinonimo del campo chiave di B. Qualora la ricerca abbia risultato positivo, il campo identificativo trovato, può essere utilizzato in fase di join.
   Nel caso sfortunato in cui la ricerca non porti risultati, oppure nel caso in cui entrambe le classi A e B siano sprovviste di chiave, si rende indispensabile l'intervento del progettista.

2. Sia tra le due classi locali A e B definito un'assioma di inclusione, si supponga:

   classe A NT_{Ext} classe B

   Se una sola delle due classi è sprovvista di chiave, e questa è la classe inclusa (A nel nostro caso), allora è necessario ricercare tra i campi di quest'ultima se ne esiste uno sinonimo del campo chiave della classe B. Se così è, allora il campo trovato puo' essere usato per realizzare il join.
   In tutti gli altri casi, compreso quello in cui entrambe le classi locali coinvolte sono prive di chiave, è necessario che il progettista intervenga.

3. La relazione estensionale esistente tra le due classi locali in analisi, è di disgiunzione: le due classi non hanno informazioni in comune e quindi il join tra di esse non è richiesto.

4. Nel caso in cui non vi è nessuna relazione estensionale esplicita tra le classi e solo una delle due classi è sprovvista di chiave, ancora una volta e' necessaria una ricerca mirata a trovare eventuali relazioni implicite. Qualora questa ricerca produca qualcosa, a seconda del tipo di di relazione reperita, si procede come spiegato nei punti precedenti.
   Se invece la ricerca non produce alcun risultato (classi locali sovrapposte) o entrambe le classi non hanno campi chiave, è richiesto l'intervento esplicito del progettista.

3.3.3 Intervento del progettista

Il processo che porta all'individuazione dei campi semanticamente significativi, sulla base dei quali rendere possibile la realizzazione della fusione delle informazioni in risposta alle varie query locali, risulta essere solo in parte un procedimento automatico. Ciò è reso palesemente evidente nella Sezione 3.3.2, dove nell'enunciare le *Regole di join*, si parla più volte di "intervento del progettista".
La natura di questo intervento è varia e dipende dalle relazioni intensionali ed estensionali che legano le classi locali interessate dalla fusione.
Nel caso più semplice, l'azione svolta dal progettista si limita ad essere una validazione dei campi da selezionare per realizzare il join. Per chiarire il concetto di validazione, si consideri il caso delle classi locali `UNI.Research_Staff` e `CS.CS_Person`. Le due classi prevedono entrambe la dichiarazione di una chiave, che è rispettivamente:

$K_{UNI.Research\_Staff}$ = (`name`)
$K_{CS.CS\_Person}$ = (`first_name,last_name`)

Queste chiavi sono legate da una relazione di tipo NT_{Int} e sono mappate nel medesimo attributo globale `name`. Quindi, per quanto detto in precedenza, le due chiavi sono semanticamente omogenee.
Tra le due classi non risulta definito alcun assioma estensionale, di conseguenza `UNI.Research_Staff` e `CS.CS_Person` sono considerate sovrapposte in quanto appartenenti alla stessa classe globale. Coerentemente con quanto detto nella Sezione 3.3.2, nell'ambito delle chiavi semanticamente omogenee, non si può procedere alla selezione automatica del campo chiave `name` individuato, come campo su cui effettuare il join.

Il progettista, arrivati a questo punto, munito di un'approfondita conoscenza semantica ed estensionale, decide se effettivamente il join debba essere eseguito sui campi chiave individuati. Se così è il suo intervento si limita ad una selezione esplicita dei campi di join (già comunque individuati dal sistema). Nel nostro esempio quindi l'attributo globale di join trovato sarebbe `name`.
Nel caso sfortunato in cui il progettista si renda conto che effettuare il join sui campi individuati dal sistema porterebbe ad una fusione delle informazioni errata, il suo intervento assume una connotazione diversa. Nel nostro esempio il join automatico sul campo `name` porterebbe ad una ricostruzione della informazione sbagliata, qualora si verificassero casi di omonimia.
L'intervento del progettista diventa più complesso ed è maggiormente invasivo nei casi in cui la sola validazione delle chiavi risulta insufficiente, e vi è quindi la necessità di arricchire la conoscenza in possesso al sistema mediatore. I casi in cui ciò risulta necessario sono:

- presenza di chiavi semanticamente disomogenee nei casi evidenziati nella Sezione 3.3.2;
- assenza di chiavi dichiarate nei casi evidenziati nella Sezione 3.3.2;
- presenza di chiavi semanticamente omogenee nel caso di relazioni di sovrapposizione.

Il problema in questi casi è che il sistema mediatore non è in grado di identificare univocamente le informazioni facenti referimento alla medesima entità del mondo reale, e di conseguenza non è neanche in grado di stabilire una corrispondenza, ai fini della fusione, tra di esse. In questo caso, è necessario che il progettista definisca su quali campi significativi effettuare il join, ma soprattutto è indispensabile che dichiari esplicitamente la corrispondenza esistente tra i valori assunti da tali campi nelle varie istanze delle classi coinvolte.
La scelta effettuata da MOMIS, per permettere al progettista di intervenire, consiste nell'inserimento esplicito, da parte di quest'ultimo, di una serie di tabelle in cui viene definito il matching richiesto tra i valori assunti dagli attributi di join di due classi con informazioni in comune. Queste tabelle vengono gestite dal sistema alla stregua di classi locali, e possono, o essere raccolte in una sorgente dedicata (la "Sorgente di Join") o essere distribuite tra le varie sorgenti su cui il sistema lavora (scelta fatta per MOMIS). A questo punto risulta evidente che il join non avviene più direttamente tra le istanze delle due classi locali in analisi, ma deve passare attraverso un join "intermedio" con la classe in cui è esplicitato il matching tra i valori contenuti nei campi chiave.
Per capire meglio la dinamica dell'intervento del progettista, può essere d'aiuto il

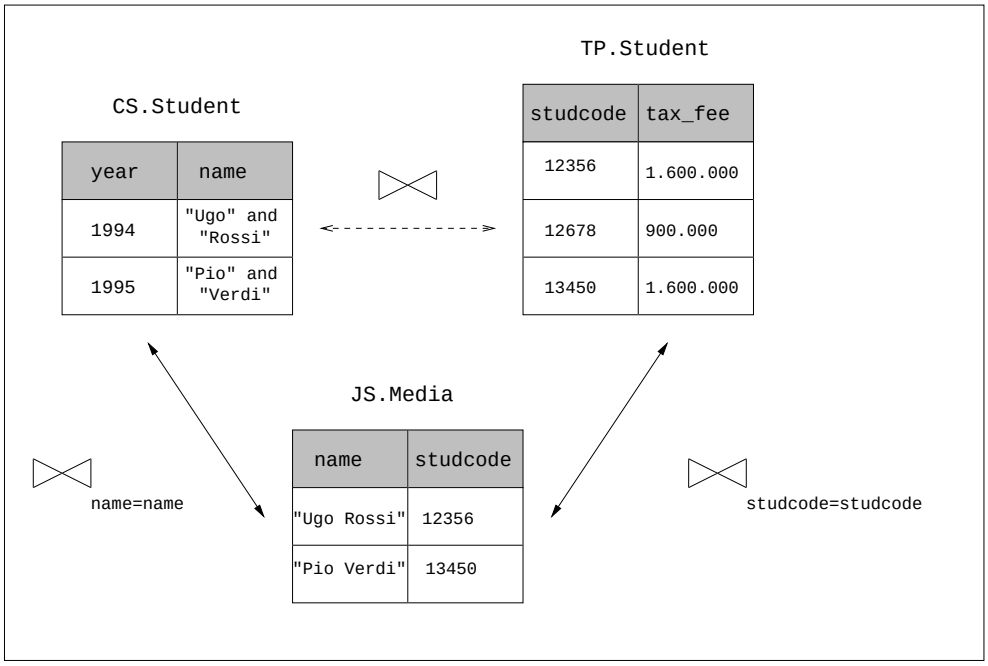

Figura 3.2: Join attraverso classe intermedia

seguente esempio.

Date le classi locali `CS.Student` e `TP.Student`, su di esse è definito l'assioma estensionale di inclusione:

`CS.Student` NT_{Ext} `TP.Student`

La classe `CS.Student` prevede la chiave $K_{CS.Student}$ = (`first_name, last_name`) a cui corrisponde l'attributo globale `name`.
La classe `TP.Student` prevede la chiave $K_{TP.Student}$ = (`student_code`) a cui corrisponde l'attributo globale `studcode`.
È evidente che le due chiavi risultano essere semanticamente disomogenee. Seguendo il punto 2. delle *Regole di join* nel caso di chiavi semanticamente disomogenee, è opportuno effettuare una ricerca tra i campi non chiave di `CS.Student` (o analogamente per `TP.Student`) per vedere se ne esiste uno sinonimo. Questo caso non si verifica, quindi è necessario l'intervento del progettista che provvede ad esplicitare in una tabella, che prevede i campi `name` e `studcode`, il matching tra i valori assunti dai campi chiave di `CS.Student` con i valori assunti dai campi chiave di `TP.Student`. Inoltre, il progettista

deve provvedere a dichiarare esplicitamente le corrispondenze tra gli attributi globali che mappano i campi chiave delle due classi locali `CS.Student` e `TP.Student`, e gli attributi globali che mappano gli attributi locali della classe intermedia.
Risulta chiaro a questo punto che il join tra `CS.Student` e `TP.Student` non è più diretto ma passa attraverso un join intermedio con la classe esplicitata dal progettista. Quindi, supponendo che la classe intermedia si chiami `Media` e che si trovi nella sorgente `JS`, il processo di fusione passa attraverso i seguenti join:

$$(\texttt{CS.Student} \bowtie_{\texttt{name=name}} \texttt{JS.Media}) \bowtie_{\texttt{studcode=studcode}} \texttt{TP.Student}$$

Prendendo in considerazione alcuni valori d'esempio per le classi in analisi, il processo di fusione, rappresentato con modellazione tabellare in Figura 3.2, permette di ricostruire, nel caso di istanze facenti riferimento alla stessa entità del mondo reale, il seguente gruppo di informazioni:

("Ugo Rossi",12356,1.600.000,1994)

("Pio Verdi",13450,1.600.000,1995)

Come le informazioni per il join vengano utilizzate dal sistema, e come il join venga effettivamente realizzato verrà accuratamente esposto nella Sezione 3.5.

3.3.4 Formalizzazione delle regole di join

In questa sezione verrà definita una formalizzazione della conoscenza generata dalla definizione delle *Regole di join*. In particolare, i concetti definiti nella Sezione 3.3.2 vengono descritti attraverso una rappresentazione formale, che consente di indicare in maniera non ambigua i passi da eseguire per individuare i campi semanticamente identificativi, su cui effettuare il join nel processo di Object Fusion.
Come primo passo si procede a dare una definizione del concetto di omogeneità semantica per gli attributi locali di una classe, già visto ed utilizzato nelle pagine precedenti.

Definizione 9 (Omogeneità semantica) *Date due classi locali $L_h(X)$ e $L_j(Y)$ appartenenti alla medesima classe globale G, siano $V \subseteq X$ e $W \subseteq Y$ due sottinsiemi degli attributi locali di L_h e L_j rispettivamente.*

Sia definita la funzione a_G

$$a_G : \{A_1, A_2, \ldots, A_M\} \rightarrow Ag$$

che associa ad un set di attributi locali, l'attributo globale $Ag \in A_G(G)$ *che li mappa.*

Allora si dice che "V è semanticamente omogeneo a W" se e solo se:

$\forall\{A_1, A_2, \ldots, A_K\} \subseteq V$: $\exists a_G(A_1, A_2, \ldots, A_K) \Rightarrow \exists\{B_1, B_2, \ldots, B_P\} \subseteq W$: $((A_1$*[and]*A_2*[and]*$\ldots A_K)SYN_{Int}(B_1$*[and]*B_2*[and]*$\ldots B_P)) \wedge (a_G(A_1, A_2, \ldots, A_K) \equiv a_G(B_1, B_2, \ldots, B_P))$

$\wedge$

$\forall\{B_1, B_2, \ldots, B_P\} \subseteq W$: $\exists a_G(B_1, B_2, \ldots, B_P) \Rightarrow \exists\{A_1, A_2, \ldots, A_K\} \subseteq V$: $((B_1$*[and]*B_2*[and]*$\ldots B_P)SYN_{Int}(A_1$*[and]*A_2*[and]*$\ldots A_K)) \wedge (a_G(B_1, B_2, \ldots, B_P) \equiv a_G(A_1, A_2, \ldots, A_K))$

con $K, P \in \{1, \ldots, M\}$

Sfruttando questa definizione, è possibile esprimere le *Regole di join* nel seguente modo:

Definizione 10 (Regola di join) *Date due classi locali* $L_h(X)$ *e* $L_j(Y)$ *appartenenti alla medesima classe globale* G*, sia:*

- $er \in \{SYN_{Ext}, NT_{Ext}$[2]$, BT_{Ext}, DISJ_{Ext}, null$[3]$*\} la relazione estensionale che sussiste tra di loro;*
- $k_h = \{A_1, A_2, \ldots, A_K\} \subseteq X$ *una chiave per la classe locale* L_h*;*
- $k_j = \{B_1, B_2, \ldots, B_P\} \subseteq Y$ *una chiave per la classe locale* L_j*;*
- $K_h = \{k_{h1}, \ldots, k_{hM}\}$ *l'insieme delle chiavi della classe locale* L_h*;*
- $K_j = \{k_{j1}, \ldots, k_{jN}\}$ *l'insieme delle chiavi della classe locale* L_j*;*
- AJ *l'insieme degli attributi globali di join;*

[2]La relazione nella trattazione verrà considerata nel seguente ordine: $L_h NT_{Ext} L_j$.

[3]Caso in cui non viene espressa nessuna relazione estensionale e quindi, per l'appartenenza delle due classi alla medesima classe globale, si suppone una relazione di sovrapposizione.

- *sia definita la funzione* a_G

  $$a_G : \{A_1, A_2, \ldots, A_F\} \rightarrow \{Ag_1, \ldots, Ag_N\}$$

  che associa ad un set di attributi locali di una data classe locale, l'insieme di attributi globali $\{Ag_1, \ldots, Ag_N\} \subseteq A_G(G)$ *che li mappa.*

Allora tra le due classi viene formulata una regola, che permette di stabilire su quali attributi globali effettuare il join, e che può assumere le seguenti caratteristiche:

- *if* $\exists k_h \in K_h \wedge \exists k_j \in K_j$*:"*$k_h$ *e* k_j *sono semanticamente omogenee" ⇒*
  - **case** *er* **of** SYN_{Ext}, NT_{Ext} *:* $a_G(k_h) \in AJ$*;*
  - **case** *er* **of** $null$ *:* `<intervento del progettista>`*;*
  - **case** *er* **of** $DISJ_{Ext}$ *:* $AJ \equiv \emptyset$*.*
- *if* $\nexists k_h \in K_h \wedge \nexists k_j \in K_j$*:"*$k_h$ *e* k_j *sono semanticamente omogenee" ⇒*
  - **case** *er* **of** SYN_{Ext} *:*
    - *if* $a_G(k_h) \subseteq a_G(k_j) \Rightarrow a_G(k_h) \in AJ$*;*
    - *if* $a_G(k_j) \subseteq a_G(k_h) \Rightarrow a_G(k_j) \in AJ$*;*
    - *if* $\exists\{B_1, B_2, \ldots, B_P\} \subseteq Y$ *:"*k_h *e* $\{B_1, B_2, \ldots, B_P\}$ *sono semanticamente omogenei" ⇒* $a_G(k_h) \in AJ$*;*
    - *if* $\exists\{A_1, A_2, \ldots, A_K\} \subseteq X$ *:"*k_j *e* $\{A_1, A_2, \ldots, A_K\}$ *sono semanticamente omogenei" ⇒* $a_G(k_j) \in AJ$*;*
    - *in tutti gli altri casi :*`<intervento del progettista>`*.*
  - **case** *er* **of** NT_{Ext} *:*
    - *if* $a_G(k_j) \subseteq a_G(k_h) \Rightarrow a_G(k_j) \in AJ$*;*
    - *if* $\exists\{A_1, A_2, \ldots, A_K\} \subseteq X$ *:"*k_j *e* $\{A_1, A_2, \ldots, A_K\}$ *sono semanticamente omogenei" ⇒* $a_G(k_j) \in AJ$*;*
    - *in tutti gli altri casi :*`<intervento del progettista>`*.*
  - **case** *er* **of** $null$ *:* `<intervento del progettista>`*.*
  - **case** *er* **of** $DISJ_{Ext}$ *:* $AJ \equiv \emptyset$*.*
- *if* $K_h \equiv \emptyset \wedge K_j \not\equiv \emptyset$ *:*
  - **case** *er* **of** SYN_{Ext}, NT_{Ext} *:*

  - ∗ *if* $\exists\{A_1, A_2, \ldots, A_K\} \subseteq X$ *:*$\exists k_j \in K_j$ *tale che "*k_j *e* $\{A_1, A_2, \ldots, A_K\}$ *sono semanticamente omogenei"* $\Rightarrow a_G(k_j) \in AJ$*;*
  - ∗ *in tutti gli altri casi :*`<intervento del progettista>`.
- – **`case`** *er* **`of`** $null$ *:* `<intervento del progettista>`.
- – **`case`** *er* **`of`** $DISJ_{Ext}$ *:* $AJ \equiv \emptyset$.

- *if* $K_h \not\equiv \emptyset \wedge K_j \equiv \emptyset$ *:*
  - – **`case`** *er* **`of`** SYN_{Ext} *:*
    - ∗ *if* $\exists\{B_1, B_2, \ldots, B_P\} \subseteq Y$ *:*$\exists k_h \in K_h$ *tale che "*k_h *e* $\{B_1, B_2, \ldots, B_P\}$ *sono semanticamente omogenei"* $\Rightarrow a_G(k_h) \in AJ$*;*
    - ∗ *in tutti gli altri casi :*`<intervento del progettista>`.
  - – **`case`** *er* **`of`** $NT_{Ext}, null$ *:* `<intervento del progettista>`.
  - – **`case`** *er* **`of`** $DISJ_{Ext}$ *:* $AJ \equiv \emptyset$.
- *if* $K_h \equiv \emptyset \wedge K_j \equiv \emptyset$ *:* `<intervento del progettista>`.

3.4 Strutture dati: Join Map e Join Table

Le *Regole di join*, ricavate sfruttando la conoscenza intensionale ed estensionale, e le informazioni esplicitate dal progettista, quali la classe intermedia di join e gli attributi globali su cui effettuare il join, racchiudono in sè tutto ciò che serve per permettere la fusione delle istanze di una medesima entità, in fase di ricostruzione della risposta globale.
Però, perchè tutto ciò sia effettivamente ed automaticamente utilizzabile dal Query Manager per realizzare la fusione (come questo avvenga verrà spiegato nella prossima sezione), è necessario che venga riorganizzato per dare vita ad una serie di strutture dati.
Come si è visto nella Sezione 3.3.2 ogni regola di join coinvolge due classi locali. Ebbene la struttura dati che viene generata è la **Join Map**, ovvero una "mappa di join" che indica per una coppia di classi locali, se vi è sovrapposizione di informazioni e quindi se vi è la necessità di effettuare il join, ed in caso affermativo quali attributi globali utilizzare in questa operazione. L'operazione di join può avvenire, a seconda delle relazioni estensionali ed intensionali che coinvolgono le classi locali, o in modo diretto o in modo indiretto, passando attraverso un join intermedio con una classe esplicitata dal progettista, che rende

la fusione corretta.

Formalmente, facendo riferimento al modello di rappresentazione dello schema globale in Sezione 2.2 ed al modello di rappresentazione delle *Regole di join* in Sezione 3.3.4, si ha:

Definizione 11 (Join Map) *Data una coppia di classi locali (L_h, L_j), con $h \neq j$, appartenenti alla medesima classe globale $G_i \in G$, una Join Map è una tupla $JM_{h,j}$, che può assumere le seguenti strutture:*

- **case** *direct-join:* $JM_{h,j} = (L_h, L_j, \{AJ_1, \ldots, AJ_M\})$
- **case** *indirect-join:* $JM_{h,j} = (L_h, L_j, AJ_h, AJ_j, L_y)$
- **case** *no-join*[4]*:* $JM_{h,j} = (L_h, L_j)$

Dove

- $L_y \in SG(G_y)$ *è una classe locale, che costituisce la classe intermedia di join;*
- AJ_x *è una tupla di attributi globali di join, che può assumere le seguenti strutture:*
  - **case** *direct-join:* $AJ_x = (Ag_{x1}, \ldots, Ag_{xN})$ *, con* $x \in \{1, \ldots, M\}$*;*
  - **case** *indirect-join:* $AJ_x = \{(Ag_{x1}, Ag_{y1}), \ldots, (Ag_{xN}, Ag_{yN})\}$ *, con* $x \in \{h, j\}$.

Siccome è necessario definire una Join Map per ogni coppia di classi locali di una data classe globale, è possibile raccogliere tutte queste mappe in una struttura dati di natura tabellare che prende il nome di **Join Table**. Ogni classe globale ha una propria Join Table.

Sfruttando l'esempio di riferimento, la Join Table che si ottiene per la classe globale University_Person (la più significativa per l'applicazione delle analisi effettuate in questa tesi), è quella rappresentata in Figura 3.3. Con l'intestazione Third Class si indica la classe locale intermedia utilizzata per il join indiretto.

Anche della Join Table è possibile dare una rappresentazione formale:

Definizione 12 (Join Table) *Data una classe globale G, a cui corrispondono le classi locali $\{L_1, \ldots, L_N\}$, sia $JM_{x,j}$ la generica Join Map corrispondente alla coppia di classi locali (L_x, L_y), allora una Join Table di G, è una tupla JT*

$$JT = (JM_{1,2}, \ldots, JM_{(N-1),N})$$

contenente tutte le possibili Join Map, ottenute dalla combinazione delle classi locali di G, tali che se $JM_{x,y} \in JT \Rightarrow JM_{y,x} \notin JT$.

[4]Non vi è sovrapposizione di istanze, e quindi, non vi è necessità di join.

First Class	Second Class	Join Attributes	Third Class
CS.CS_Person	CS.Professor	(name)	
CS.CS_Person	CS.Student	(name)	
CS.CS_Person	UNI.Research_Staff	(name)	
CS.CS_Person	UNI.School_Member	(name)	
CS.CS_Person	TP.Student	(name)	
CS.Professor	CS.Student		
CS.Professor	UNI.Research_Staff	(name)	
CS.Professor	UNI.School_Member		
CS.Professor	TP.Student		
CS.Student	UNI.Research_Staff		
CS.Student	UNI.School_Member	(name)	
CS.Student	TP.Student	(name,name)(studcode,studcode)	JS.Media
UNI.Research_Staff	UNI.School_Member		
UNI.Research_Staff	TP.Student		
UNI.School_Member	TP.Student	(name)	

Figura 3.3: La Join Table di University_Person

3.5 Il processo di Object Fusion

Fino ad ora si è parlato del fatto che il sistema mediatore MOMIS deve gestire informazioni di vario tipo, alcune delle quali fanno riferimento alla medesima entità del mondo reale. Si è detto che, in quest'ultimo caso, perchè si possa ottenere una risposta finale ad una generica query completa e corretta, si deve procedere ad una fusione delle informazioni, che avviene mediante una serie di join, che si basano sulle chiavi identificative individuate mediante le *Regole di join* e le direttive esplicitate dal progettista. Quello che non è ancora stato detto, è quando e in che modo MOMIS realizza il join fra le informazioni relative alla medesima entità.
Il modulo di MOMIS presposto all'esecuzione dell'Object Fusion è il Query Manager [27] .
Come si è visto nella Sezione 2.3, il Query Manager svolge la fase di Query Processing, che prevede le sottofasi di *Definizione del Query Plan* e di *Esecuzio-*

ne della Query. Nella prima sottofase il Query Manager individua come e a quali sorgenti accedere per reperire i dati richiesti dall'utente tramite la query. Nella seconda sottofase esegue la query, cioè accede alle sorgenti, recupera e integra i dati in modo da presentare all'utente una risposta globale. In particolare questo avviene passando attraverso tre livelli di esecuzione: esecuzione delle Local Query, esecuzione delle Basic Query, ricomposizione della Global Query.
In fase di *Esecuzione delle Local Query*, il Query Manager accede ai dati restituiti dai Wrapper in risposta alle query locali, e ne restituisce una rappresentazione globale, traducendo i valori locali e trasponendoli a livello globale. Il risultato parziale di questa operazione viene inserito in tabelle temporanee appartenenti ad un database interno a MOMIS.

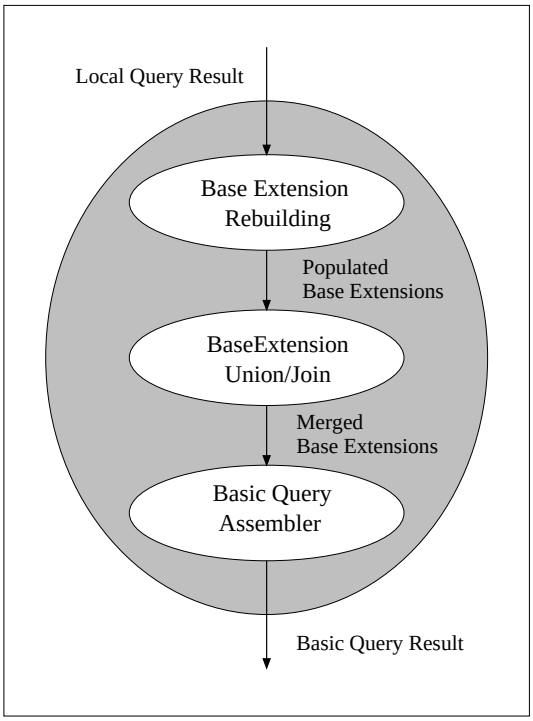

Figura 3.4: Esecuzione delle Basic Query

Il vero e proprio processo di Object Fusion avviene nella fase di *Esecuzione della Basic Query*, che prevede le sottofasi di *Ricostruzione di ogni base extension*, *Fusione delle base extension* e *Esecuzione della Basic Query Assembler*, come mostra la Figura 3.4.
In particolare la fusione viene realizzata passando attraverso le fasi di *Ricostru-*

zione della base extension e *Fusione delle base extension*.

Ricostruzione di ogni base extension

Gli attributi che una base extension presenta sono dati dall'unione degli attributi globali descritti nelle classi locali che la costituiscono. Tra gli attributi di una base extension selezionata saranno presenti tutti quelli di interesse, quelli cioè presenti nelle clausole di proiezione e di selezione della basic query; ogni classe locale invece avrà solo alcuni di questi attributi. Bisogna quindi unire gli oggetti, provenienti dalle classi locali, ma riferiti alla stessa entità del mondo reale, in modo da ottenere la struttura dati prevista per ogni base extension. Questa attività avviene mediante dei join tra i risultati provenienti dalle classi locali appartenenti alla base extension che si sta considerando. L'informazione su come vadano fuse le classi locali, cioè su quali attributi identificativi si debbano usare per effettuare il join, viene fornita dalla Join Table. Durante la fase di *Definizione del Query Plan*, le informazioni apportate dalle Join Map nella Join Table vengono inserite nel piano d'accesso e gli attributi necessari per la fusione di tutte le classi della base extension vengono eventualmente aggiunti alla clausola di selezione delle Local Query.
A livello pratico, il Query Manager genera, per ogni base extension da ricostruire, una query che riporta i join da effettuare tra le tabelle contenenti i risultati dell'esecuzione delle Local Query e pone i risultati ottenuti in altre tabelle temporanee.

Si supponga che l'insieme ottimo di base extension da interrogare sia $BE_{Opt} = \{BE1, BE2\}$, che $BE1$ coinvolga l'insieme di classi locali (classe A, classe B) e che $BE2$ coinvolga l'insieme di classi locali (classe A, classe C). Si supponga poi di ricavare dalla Join Table che interessa queste classi locali, le seguenti Join Map (ottenute mediante le *Regole di join* o l'intervento esplicito del progettista):

- $JM1$ = (classe A, classe B, $\{aj1\}$)
- $JM2$ = (classe A, classe C, $\{aj1\}$)

Quindi le query generate dal Query Manager per permettere la ricostruzione di $BE1$ e $BE2$ (interessandoci di selezionare solo gli attributi di join) sono:

```
BE1: select a.aj1
     from A as a,
          B as b
     where a.aj1=b.aj1
```

```
BE2: select a.aj1
     from A as a,
          C as c
     where a.aj1=c.aj1
```

Il Query Manager, dopo aver eseguito queste due query, crea le due tabelle corrispondenti in cui inserire i dati ottenuti.
Qualora la ricostruzione della base extension, supponiamo $BE1$, implicasse un join indiretto, la Join Map $JM1$ assumerebbe la struttura:

$JM1$ = (classe A, classe B, $\{(aj, dj1)\}$, $\{(bj, dj2)\}$, classe D)

Dove la classe D è la classe intermedia funzionale all'esecuzione del join.
Allora la query generata dal Query Manager per ricostruire $BE1$ sarebbe:

```
BE1:  select a.aj,b.bj
      from A as a,
           B as b
           D as d
      where a.aj=d.dj1
      and   d.dj2=b.bj
```

Fusione delle base extension

Una volta ricostruite le base extension, si dispone di base extension popolate, cioè contenenti oggetti distinti e dotati di uno schema uguale a quello della base extension stessa.
Se si è individuata una sola base extension di interesse questi oggetti costituiscono già il risultato di questa fase. Se invece il risultato della Basic Query proviene da più base extension, occorre quindi *fondere* gli oggetti precedenti in modo da ottenere un unico risultato. I criteri di fusione sono stati definiti durante la fase di *Definizione del Query Plan*: di default viene effettuata l'*unione*, ma nel caso in cui le due base extension ne dominino una terza, è necessario un *outer join* per evitare duplicazioni.

Si supponga di avere a che fare con l'insieme di base extension iniziale BE_{In} = $\{BE1, BE2, BE3\}$. Durante la fase di semplificazione, la base extension $BE3$ viene scartata in quanto dominata sia da $BE1$ che da $BE2$. Questo però non è sufficiente ad eliminare le duplicazioni, infatti la ricostruzione di $BE1$ e di $BE2$

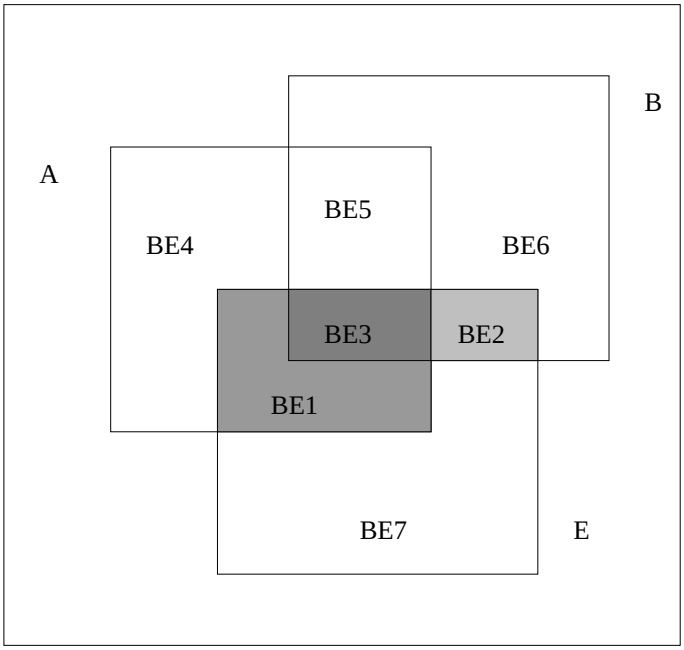

Figura 3.5: Fusione tramite outer join

genera insiemi di entità parzialmente sovrapposti e tale sovrapposizione è rappresentata proprio dall'estensione di $BE3$. In questo caso quindi è necessario un *outer join* fra i risultati ottenuti dalla ricostruzione di $BE1$ e di $BE2$. Nel fare questo vengono nuovamente in aiuto le *Regole di join*, perchè è possibile fondere due base extension se si determina un identificatore comune ad una classe locale di una base extension e ad una classe locale dell'altra, o eventualmente usando la chiave semantica di una classe comune ad entrambe.

Facendo riferimento al set di base extension BE_{In}, rappresentato in Figura 3.5, il risultato della base extension $BE1$ è fornito dalla fusione delle classi A ed E, mentre la base extension $BE2$ è costituita da B e E.

Supponiamo che nella Join Table che interessa le classi citate, si trovino le seguenti Join Map:

- $JM1$ = (classe A, classe B,$\{(aj1), (aj2)\}$)
- $JM2$ = (classe A, classe E,$\{aj2\}$)
- $JM3$ = (classe B, classe E,$\{aj2\}$)

Siccome la classe E è comune a $BE1$ ed a $BE2$, si può usare $aj2$ come chiave per realizzare la fusione (inoltre come si evince da $JM1$, $aj2$ è attributo di join

per la coppia (A,B)). Quindi il Query Manager produce la seguente query:

```
BE1_BE2: select BE1.aj2
         from BE1 full join BE2 on (BE1.aj2=BE2.aj2)
```

L'ultimo passo che rimane per l'ottenumento del risultato della Basic Query è l'*Esecuzione della Basic Query Assembler*, contenente i predicati non verificati dalle Local Query e la `<select-clause>` della query originaria. Infatti durante la fase di *Definizione del Query Plan*, la query è stata aggiornata aggiungendo eventuali attributi necessari per la fusione delle base extension, per la valutazione dei predicati "esclusi" ed inoltre i risultati delle Local Query possono contenere campi aggiunti per effettuare la ricostruzione delle base extension. Con l'esecuzione della Basic Query Assembler si ottiene quindi il risultato finale della Basic Query, comprendente sia la valutazione completa dei predicati, sia il reperimento delle informazioni effettivamente richieste inizialmente.

In fase di *Esecuzione della Global Query* i risultati temporanei forniti dalle Basic Query vengono utilizzati per generare la relazione rappresentante il risultato finale. Viene sfruttato il piano di esecuzione che, a questo livello, contiene invocazioni a funzioni complesse; vengono quindi eseguiti tutti i join tra le classi globali, le aggregazioni e le altre operazioni non trattabili dalle Basic Query, ma necessarie per ottenere una risposta veramente corrispondente alla Global Query formulata dall'utente.

3.6 Sviluppi futuri

Un'imponente prospettiva per l'operazione di individuazione dei campi di join, su cui fondere oggetti che modellano la medesima entità del mondo reale, riguarda il ruolo ricoperto da questa operazione nella fase di integrazione degli schemi. Allo stato attuale l'operazione in questione viene svolta quando tutte le altre fasi del processo di integrazione degli schemi sono giunte a termine ed hanno prodotto i loro risultati.
Ciò è dimostrato anche dal fatto che le *Regole di join*, per dare i loro risultati automatici ricevono in input la conoscenza estensionale e la conoscenza intensionale, prodotte dalla fase di integrazione degli schemi. La conoscenza estensionale è rappresentata dagli assiomi estensionali e dalle classi globali determinate, mentre la conoscenza intensionale è rappresentata dalla Mapping Table e dalle relazioni semantiche contenute nel Common Thesaurus.

La sfida che ci si propone per il futuro immediato di MOMIS, è quella di sfruttare il concetto di *joinable* tra classi, non a posteriori, quando le classi globali sono già state definite, bensì ortogonalmente all'operazione di clustering. Infatti, durante la realizzazione di questa tesi ci si è resi conto del forte significato semantico che risiede nel concetto di *joinable* tra due classi locali, e del fatto che esso potrebbe concorrere alla determinazione degli *Affinity Coefficient* tra classi.
Come si è visto, il progettista ricopre un ruolo fondamentale nella determinazione delle relazioni di fusione tra istanze, quindi potrebbe essere a conoscenza a priori di una relazione di join tra due classi locali e quindi potrebbe esprimerla esplicitamente. A questo tipo di relazione potrebbe essere associato un peso, come avviene per le relazioni SYN_{Ext}, NT_{Ext} e BT_{Ext}. Infatti sappiamo che, ogni assioma estensionale definito, qualora sia validato (la validazione viene fatta valutando la compatibilità tra i domini degli attributi delle due classi), vincola le due classi che coinvolge ad avere anche un legame intensionale.
Il legame che viene generato, quindi, è molto forte, perchè implica sia un legame tra gli schemi che un legame tra le istanze. Quindi le relazioni intensionali logicamente implicate dagli assiomi estensionali vengono registrate nel Common Thesaurus e viene assegnato loro un peso di affinità pari a 1. Siccome la relazione di *join* ha un'implicazione intensionale ed estensionale molto forte anch'essa, ad essa potrebbe venire associato un peso analogo a quello dato alle relazioni estensionali, quindi $\sigma_{join} = 1$.
Perchè una relazione estensionale sia validata e finisca di conseguenza nel Common Thesaurus, è necessario che, come abbiamo detto sopra, tra le classi da essa interessate sussista un vincolo di compatibilità tra gli schemi. Quindi, si potrebbe verificare il caso sfortunato in cui, classi parzialmente sovrapposte finiscono in cluster diversi, perchè, non essendo la relazione estensionale che li lega validata, il loro Global Affinity Coefficient è basso e rimane al di sotto della soglia. Il fatto di associare un peso di affinità alla relazione di *joinable* tra classi, potrebbe aiutare a superare questo inconveniente. In questo modo classi, con istanze sovrapposte, che potrebbero finire, in un caso sfortunato, in classi globali diverse perchè sprovviste di relazioni estensionali e semantiche forti validate, verrebbero riconosciute come affini lo stesso ed assegnate al medesimo cluster (e di conseguenza alla medesima classe globale).

Capitolo 4

Stato dell'arte

MOMIS, nonostante l'integrazione di informazioni sia un campo di ricerca relativamente nuovo, non è l'unico sistema che cerca di realizzare un modulo integratore. Tra i sistemi mediatori più conosciuti si possono citare GARLIC [23, 24], SIMS [25, 26], HERMES [21], TSIMMIS [13],ecc.
In particolare quest'ultimo sviluppa in maniera approfondita ed interessante le problematiche, riguardo all'Object Fusion [15, 41], al centro della trattazione di questa tesi. Per questo motivo, in questo capitolo, si esporrà come TSIMMIS realizza il processo di fusione e lo si metterà a confronto con l'approccio adottato da MOMIS.
Inoltre, nel Capitolo 3, si è vista l'importanza del concetto di *omogeneità semantica* tra i campi chiave, nel processo che porta al riconoscimento delle istanze facenti riferimento alla medesima entità del mondo reale.
Le grandi organizzazioni necessitano di scambiare ed integrare informazioni, tra molti sistemi separati tra di loro. Perchè questa attività produca dei risultati utili e corretti, è fondamentale che vi sia accordo sui significati dei dati gestiti. Insomma, quello che bisogna garantire è che vi sia *interoperabilità semantica*. In questo capitolo verranno esposti alcuni studi riguardo questa problematica, e si cercherà di capire come questi possano influenzare gli sviluppi futuri del sistema MOMIS.

4.1 TSIMMIS

TSIMMIS (The Stanford_IBM Manager of Multiple Information Sources), 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 (ed a MOMIS in particolare): i *Wrapper* convertono i dati in un modello comune, mentre i *Mediator* combinano ed integrano i dati ricevuti dai

Wrapper.
La peculiarità di TSIMMIS si manifesta nel modello comune dei dati utilizzato per rappresentare le informazioni. OEM (Object Exchange Model) è un modello ad *etichette* basato sui concetti di identità di oggetto e di annidamento, particolarmente adatti a descrivere dati la cui struttura non è nota o è variabile nel tempo. Usando OEM, non è necessario che oggetti che si descrivono tramite la stessa etichetta abbiano pure lo stesso schema. In aggiunta a questo modello 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. In particolare quest'ultimo linguaggio viene sfruttato per definire in modo dichiarativo i mediatori: attraverso l'uso di *rules* si può specificare il punto di vista del mediatore ed in questo modo definire lo "schema globale" in modo manuale. Ogni rule è costituita da una *testa*, che definisce la struttura che l'oggetto dovrà avere una volta estratto e ricostruito; e da una *coda*, che descrive dove andare a recuperare l'oggetto che si vuole ricevere.
Per un'analisi più approfondita dell'architettura di TSIMMIS e dei linguaggi da esso utilizzati si rimanda alla bibliografia. Quello che interessa analizzare in questa sezione è come TSIMMIS affronti le problematiche inerenti al processo di Object Fusion.

4.1.1 Object Fusion basata su oid semantici

In TSIMMIS il processo di fusione delle istanze è basato sull'uso di *oid* semantici: il mediatore è specificato da una serie di rule logiche e non procedurali, che permettono di mappare oggetti relativi a determinate entità del mondo reale in determinate sorgenti, in un oggetto "virtuale" presso il mediatore. A questi oggetti virtuali viene assegnato un *oid* semanticamente significativo, dimodochè oggetti che presso il mediatore hanno lo stesso *oid* sono fusi assieme, perchè riferiti alla medesima entità del mondo reale. Ovviamente questa fusione avviene solo quando arriva una query al mediatore, le cui specifiche possono essere equiparate a delle viste.

Per mostrare come TSIMMIS realizzi l'Object Fusion, si supponga di voler integrare sorgenti che presentano delle informazioni bibliografiche. Mediante OEM un generico oggetto di queste sorgenti (esportato da un wrapper) può essere così rappresentato in un generico mediatore chiamato `simple`:

```
<&r1, report, set, {&r1n, &r1a, &r1t}>
      <&r1n, report_num, string, "AB-123-456">
      <&r1a, author, string, "John Patriot">
      <&r1t, title, string, "UN Conspirancies">
              ...
```

Per facilitare l'operazione di integrazione, quando nel mediatore si rappresentano gli oggetti OEM, si usa un identificatore semanticamente significativo: ad esempio se `report_num` è una *chiave*, può essere usata per riconoscere altri oggetti che modellano la medesima entità del mondo reale. Quindi `&AB-123-456` è l'*oid* semantico che si cercava.
Si supponga che `simple` esporti oggetti che presentano l'etichetta `techreport`: questi oggetti fondono informazioni di natura bibliografica (`report`) che presentano lo stesso `report_num`, e sono nelle generiche sorgenti `s1` e `s2`.
Allora sul mediatore possono essere effettuate tramite MSL le seguenti specifiche/rule, che definiscono la *vista* esportata:

(R1):

```
<trep(RN) thecreport {<title T>}>@simple :-
          <report {<report_num RN> <title T>}>@s1
```

(R2):

```
<trep(RN) thecreport {<postscript P>}>@simple :-
          <report {<report_num RN> <postscript P>}>@s2
```

La rule R1 specifica che se esiste una coppia di valori (*t*,*r*) per le variabili `T` e `RN`, tali che la sorgente `s1` contiene un oggetto `report` che prevede i sottoggetti `report_num` e `title` che assumono rispettivamente i valori *r* e *t*; allora il mediatore `simple` esporta un oggetto `techreport` con *oid* `trep(`*r*`)`, che ha un sottoggetto `title` con valore *t* ed un unico *oid* generato dal sistema.
Analogamente si possono leggere le specifiche presenti nella rule R2. La funzione `trep()` non è altro che una funzione di skolemizzazione che viene applicata alla chiave semantica comune.
Si può osservare che R1 non impedisce agli oggetti `techreport` con *oid* `trep(`*r*`)` di avere sottoggetti diversi da `title`, quindi permette alla rule R2 di aggiungere più sottoggetti (`postscript` in questo caso) allo stesso oggetto `techreport`. In generale è così che viene realizzata l'Object Fusion in TSIMMIS: MSL fornisce delle rules che permettono di inserire incrementalmente ed indipendentemente informazioni in un oggetto del mediatore, identificato tramite un *oid* semantico.
I passi eseguiti dal Mediatore, nel processo che va dall'acquisizione della query alla generazione degli oggetti che costituiscono la risposta finale, sono descritti nella prossima sezione.

4.1.2 Query Processing

Nella fase di Query Processing viene definito un *datamerge program*, sulla base della query posta dall'utente e delle rule sul mediatore, che descrivono la vista esportata su cui è posta la query. Questo datamerge program consiste in una collezione di rule, la cui coda si riferisce alla struttura degli oggetti presso le rispettive sorgenti, e la cui testa descrive la struttura degli oggetti costituenti la risposta.

Si consideri il mediatore `simple`, descritto nella sezione precedente, che integra informazioni provenienti dalle sorgenti `s1` e `s2`.
Si formuli su `simple` la seguente query, che richiede tutti i sottoggetti degli oggetti `techreport` che prevedono titolo `'abc'`.

(Q1):

```
<X techreport V> :- <X techreport:{<title 'abc'>}@simple
```

La prima cosa da fare è convertire la query Q1 e le rule R1 e R2 sul mediatore, in forma normale MSL:

(Q1a):

```
<X techreport {<Void Vl Vv>}> :-
<X report {<T2 title 'abc'>}>@simple
AND
<X report {<Void Vl Vv>}>@simple
```

(R1a):

```
<trep(RN1) techreport {<T1 title T>}>@simple :-
<Ro1 report {<RNo1 rn RN1> <T1 title T>}>@s1
```

(R2a):

```
<trep(RN2) techreport {<Poid postscript P>}>@simple :-
<Ro2 report {<RNo2 rn RN2> <Poid postscript P>}>@s2
```

In particolare, nella query Q1a è stata spezzata la coda, per far sì che ogni insieme { ... } contenga esattamente un solo oggetto < ... >.
Inoltre, per quanto riguarda le rule R1a e R2a, vengono rinominate le variabili, in modo che non ci siano due rule con variabili comuni. Questo viene fatto per evitare confusione quando le due rule sono fuse in una singola rule nel datamerge program.

Arrivati a questo punto, si procede all'individuazione del matching tra le condizioni espresse nella coda della query e la testa delle rule. In partcolare una condizione *c* realizza il matching con una rule *r*, se la rule può produrre oggetti che soddisfano la condizione. Ogni matching che ha successo, produce un *unifier*, che è una struttura dati che descrive il matching tra *c* e *r*. Per ogni unifier, la condizione *c* viene sostituita dalle condizioni sulle sorgenti specificate nella rule *r*.
Quindi, nel nostro caso, si producono i seguenti unifier.

θ_1 = [(R1a):`X`→`trep(RN1), T1`→`T2, T`→`'abc'`]

Siccome la rule R2a realizza due matching, è necessario introdurre una seconda istanza della rule in questione:

(R2a.bis):

```
<trep(RNb) techreport {<T1b title Tb>}>@simple :-
<Ro1b report {<RNo1b rn RNb> <T1b title Tb>}>@s1
```

e quindi gli unifier:

θ_2 = [(R2a):`RN2`→`RN1, Void`→`Poid, Vl`→`postscript, Vv`→`P`]

θ_3 = [(R2a.bis):`RNb`→`RN1, Void`→`T1b, Vl`→`title, Vv`→`T1b`]

Di conseguenza, per la query Q1, il datamerge program (DP) che si ottiene è:

(DP1):

```
<trep(RN1) techreport {<Poid postscript P>}:-
<Ro1 report {<RNo1 rn RN1> <T2 title 'abc'>}>@s1
AND
<Ro2 report {<RNo2 rn RN1> <Poid postscript P>}>@s2
```

(DP2):

```
<trep(RN1) techreport {<T1b title Tb>}:-
<Ro1 report {<RNo1 rn RN1> <T2 title 'abc'>}>@s1
AND
<Ro1b report {<RNo1b rn RN1> <T1b title Tb>}>@s1
```

Come è evidente, il datamerge program definito, permette di inserire incrementalmente ed indipendentemente informazioni, nel medesimo oggetto nel mediatore, identificato univocamente dall'*oid* semantico `trep(RN1)`.

4.1.3 Considerazioni

Il vantaggio maggiore offerto da questa soluzione risiede nel creare la possibilità di raggruppare/fondere oggetti a prescindere dalla sorgente in cui si trovano, sulla base dell'identificatore prescelto per la definizione dell'*oid* specifico. Si realizza un modello molto vicino ad un'approccio ad oggetti, realizzando una maggiore modularità della soluzione, in quanto l'introduzione di una nuova sorgente determina esclusivamente la scrittura di una nuova rule. Una condizione che può essere vincolante risiede nella necessità che esista una chiave semantica comune a tutte le sorgenti: la sua presenza è indispensabile per poter fondere gli oggetti. Qualora in una sorgente non fosse presente l'attributo che per le altre sorgenti è stato identificato come significativo, si è costretti ad esportare oggetti senza poterli fondere, perchè gli *oid* generati per oggetti provenienti da sorgenti diverse non sono compatibili.
La soluzione al problema dell'Object Fusion che è stata proposta per MOMIS, vuole mantenere sia la capacità fondere gli oggetti che di creare oggetti complessi. Per assicurare la fusione delle istanze facenti riferimento alla medesima entità del mondo reale si definiscono, per ogni coppia di classi locali di una data classe globale, i campi semanticamente identificativi (le *chiavi* quando dichiarate) sulla base dei quali effettuare il join in fase di *Esecuzione della Basic Query*. Perchè il join sia effettivamente realizzabile è necessario che questi campi risultino semanticamente omogenei, come illustrato in Sezione 3.3.2; oppure qualora questo non si possa ottenere, il *matching* tra i valori assunti dai campi chiave viene esplicitato dal progettista tramite una tabella, che richiede il passaggio attraverso un join intermedio per realizzare una corretta fusione.
Il vantaggio di questa soluzione consiste nel superare la forte restrizione che in TSIMMIS richiede che tutte le classi aventi estensione parzialmente sovrapposta presentino la medesima chiave semantica.
Il nostro sistema, come dimostrato, è invece in grado di realizzare la fusione in ogni caso, purchè le classi coinvolte presentino a due a due chiavi semanticamente omogenee, o comunque una tabella comune di *matching*.

4.2 Interoperabilità semantica

Un importante ostacolo al raggiungimento di una corretta e completa integrazione delle informazioni è l'*eterogeneità semantica*.
Ad esempio, si consideri il caso di dati sull'altitudine. Il concetto di "altitudine", in un contesto orbitale viene considerato come distanza dal centro della terra, mentre, nel contesto dell'aviazione, definisce la distanza verticale dalla superficie terrestre.

La risoluzione di questi problemi non si può limitare ad un approccio sintattico o strutturale. L'eterogeneità sintattica riguarda differenze nella rappresentazione dei dati (ad esempio, l'altitudine può essere espressa in metri o kilometri). L'eterogeneità strutturale riguarda differenze nella struttura dei dati (ad esempio i siti web possono strutturare contenuti similari in modi diversi).
È evidente che vi è la necessità di una *riconciliazione semantica*, per ottenere il vero significato di questi dati e per evitare errori in fase di integrazione.
In questa sezione verranno esposti alcuni approcci, finalizzati alla risoluzione del problema della riconciliazione semantica.

4.2.1 Riconciliazione semantica

Come primo approccio alla riconciliazione semantica, può essere interessante fare riferimento, come esposto in [42], al concetto di *semantic value*. Alla base di questi studi, vi è la convinzione che le informazioni sul contesto di una dato, devono essere un componente attivo di un dato sistema di informazioni. Per *semantic value* si intende una porzione di dato con associato il corrispondente *contesto*. Con *contesto* di un dato si fa riferimento al metadato che raccoglie le informazioni riguardo il suo significato, le sue proprietà e la sua struttura. La conseguenza immediata di utilizzare esplicitamente il contesto di un dato, è la possibilità di rendersi conto se due dati sintatticamente diversi hanno il medesimo significato. Un esempio di semantic value può essere:

Prezzo = 1.25(Periodicità = 'quadrimestre', Valuta = 'dollaro USA')

In questo caso, il termine *Prezzo*, che assume il valore 1.25, viene definito nel suo significato dalle due proprietà *Periodicità* e *Valuta*, che ne individuano il contesto.
A questo punto, per realizzare l'operazione di riconciliazione semantica, è però necessario avere a disposizione delle *funzioni di conversione*, che permettano di ricondurre due dati al medesimo contesto, e quindi di confrontarli tra di loro (col fine di capire se rappresentano la stessa cosa).
Data una proprietà P, che individua il contesto di un termine, si definisce funzione di conversione per P una funzione $f()$, che converte il valore di P in un contesto, in quello assunto (sempre da P) in un altro contesto. Queste funzioni di conversione (che possono essere definite o dal sistema mediatore o dalle applicazioni che fanno parte del sistema mediatore) necessitano di essere raccolte in librerie, che possono essere o del sistema oppure anche trovarsi on-line. È chiaro quindi, che l'utilizzo di ogni funzione di conversione ha un costo associato: conversioni che implicano la consultazione di librerie on-line avranno un costo maggiore di conversioni che necessitano solo di un calcolo matematico.

Un problema che rimane irrisolto però, è il fatto che non sempre è possibile riconciliare (tramite funzioni di conversione) due termini al medesimo contesto. Ad esempio, non sempre è possibile trovare funzioni di conversione che riportino al medesimo contesto due valute, presenti in due semantic value rappresentanti un prezzo.

Analoghi studi sulla riconciliazione semantica (che sfruttano nuovamente il concetto di contesto) sono stati sviluppati dalla MITRE Corporation per conto del Dipartimento della Difesa degli Stati Uniti, col fine di realizzare un *Integrated Information Space (IIS)* [43].
In questi studi, si è visto come il fatto di associare ad un termine, il contesto che ne definisce il significato, possa risultare, col tempo e con l'aumentare del numero di informazioni da gestire, molto dispendioso. Questo per tre ordini di motivi:

- alcuni contesti sono molto complessi da rappresentare;
- in genere si ha a che fare con grandi numeri di attributi, a cui associare un contesto;
- anche le informazioni sul contesto necessitano di essere mantenute ed aggiornate (si hanno di conseguenza elevati costi di amministrazione).

Si rende evidente la necessità di individuare informazioni semanticamente efficienti, che permettano di descrivere correttamente il contesto di grosse classi di attributi, e di conseguenza di realizzare il processo di riconciliazione semantica. La proposta fatta dalla MITRE Corporation prevede la definizione di tre tipi di informazione, che permettono di inferire il contesto degli attributi (costituenti le informazioni da integrare), rendendo espliciti conflitti semantici che altrimenti rimarrebbero nascosti, e di conseguenza, rendendo possibile la riconciliazione semantica.
Queste informazioni sono:

1. **Source-descriptors**: ad ogni sorgente viene associato un source-descriptor, ovvero un record formato da nove campi, che permettono di delineare il contesto in cui sono inserite le sorgenti da cui provengono le informazioni.
2. **Usage-descriptors**: uno usage-descriptor è un record di sei campi, che permette di individuare il contesto in cui si inseriscono gli obiettivi di una determinata comunità di utenti, che consulta le informazioni integrate.
3. **Canonical attributes**: un canonical attribute, rappresenta una classe di attributi affini e contiene informazioni su come il significato di questi attributi cambi, al cambiare del contesto. Tra i campi che costituiscono un canonical attribute, vi è una libreria di funzioni di trasformazione semantica.

4.2.2 Considerazioni

Come si è visto nelle sezioni precedenti il processo che porta alla formulazione della Join Table di una data classe globale, non è un processo completamente automatico, bensì necessita di un significativo intervento da parte del progettista. Facendo riferimento a quanto detto in Sezione 3.3.3, tale intervento può limitarsi ad una validazione dei campi di join, oppure può essere più imponente e consistere nell'iserimento di una tabella contenente il *matching* tra i valori dei campi di join. Questa tabella realizza il concetto di *vista materializzata*.
Siccome i database che MOMIS gestisce sono in continua evoluzione e possono subire notevoli modifiche, è necessario che queste tabelle di esplicitazione del matching vengano tenute continuamente aggiornate, per garantire la correttezza dei risultati delle operazioni di fusione.
La possibilità di stabilire il contesto dei termini da integrare può essere molto importante. È immediato rendersi conto che ciò verrebbe in grosso aiuto al sistema mediatore, qualora questo si trovasse a dover gestire oggetti identificati da chiavi disomogenee o che, comunque, presentano contesti diversi.
Ad esempio, si supponga che due classi presentino istanze sovrapposte identificate da un codice alfanumerico, calcolato secondo criteri diversi. Allo stato attuale per realizzare la fusione viene usata una tabella intermedia. In futuro però, la presenza di una funzione di conversione da un codice ad all'altro renderebbe immediata l'operazione di individuazione delle istanze facenti riferimento alla medesima entità del mondo reale e, di conseguenza, faciliterebbe l'operazione di fusione.

Capitolo 5

Il modulo joinMap: progetto e realizzazione del software

In questa tesi, oltre allo studio delle metodologie che hanno portato alla definizione delle Join Map e della Join Table, è stato progettato e realizzato il modulo **joinMap** del sistema SI-Designer.
Il modulo software joinMap implementa sia le *Regole di join*, che permettono il calcolo delle Join Map, nel caso in cui ciò sia realizzabile automaticamente, sia un'interfaccia grafica che rende possibile la dichiarazione esplicita da parte del progettista delle Join Map quando questo si rende necessario. L'interfaccia grafica del modulo joinMap rende inoltre possibile la visione della Join Table definitiva, comprensiva sia delle Join Map calcolate automaticamente, che di quelle esplicitate dal progettista.
Per la fase di implementazione del modulo è stato utilizzato il linguaggio Java, che essendo completamente orientato agli oggetti, favorisce la modularità e la componibilità del software. Questo si è reso necessario in quanto MOMIS è un progetto di ricerca a lungo termine che si avvale nel tempo del contributo di molte persone.
In questo capitolo si provvederà a presentare le procedure più interessanti utilizzate dal modulo joinMap, includendo inoltre la descrizione delle strutture dati e degli algoritmi più significativi realizzati.

5.1 Organizzazione del software

Il software del modulo joinMap è organizzato in due package che raccolgono le classi java implementate per la modellazione, la definizione automatica e la dichiarazione esplicita delle strutture dati gestite dal sistema per rendere possibile il processo di Object Fusion, cioè le Join Map e la Join Table.
Verranno di seguito decritte ed analizzate le funzionalità e la struttura di que-

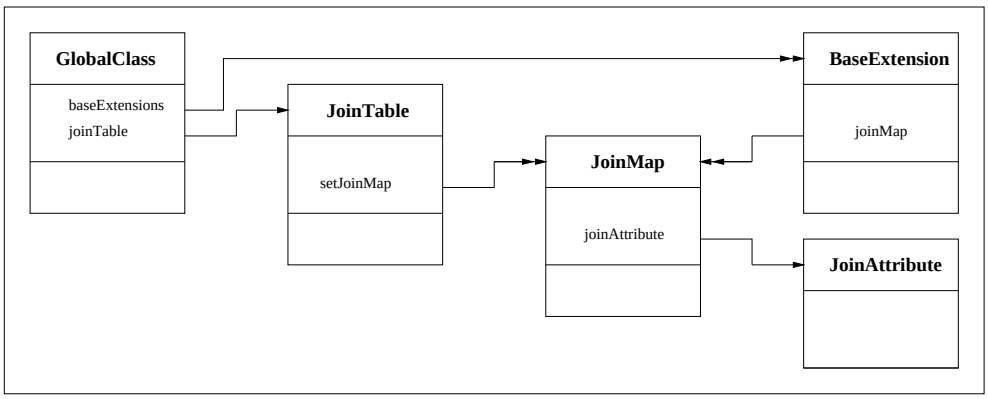

Figura 5.1: Modello ad oggetti del package globalschema interessato

sti package, tenendo presente che, nel linguaggio Java, i package permettono di raccogliere classi per scopo e relazioni di ereditarietà, fornendo una maggiore modularità ed una maggiore protezione.

5.1.1 Il package globalschema

Questo package raccoglie tutte le classi che descrivono lo schema globale e fa parte dei package che modellano i server di MOMIS.
Siccome le strutture dati fondamentali al processo di Object Fusion, le Join Map e la Join Table, fanno riferimento alle classi locali di una medesima classe globale, si è pensato di raggruppare le classi che modellano queste strutture nel package in questione. Inoltre si è ritenuto opportuno modellare anche gli attributi di join, per permetterne una più accurata gestione a seconda che il join sia diretto o indiretto.

Di seguito si passeranno in rassegna tutte le classi rappresentate nel modello ad oggetti in Figura 5.1, che sono state implementate per rendere possibile la realizzazione del processo di Object Fusion, o comunque che sono direttamente interessate dalla *fusione*. Di queste classi verranno descritti le proprietà ed i metodi di maggior interesse.

Classe JoinMap

La classe **JoinMap**, come dice il nome stesso, modella la struttura dati definita dalla Join Map.

PROPRIETÀ

- *firstElement*: è un oggetto di tipo **SourceClass** (classe del package globalschema) e rappresenta una delle due classi locali costituenti la Join Map, e delle quali tramite questa struttura dati si modella la possibilità o meno di effettuare il join.
- *secondElement*: è un oggetto di tipo **SourceClass** del tutto analogo a *firstElement* e rappresenta la seconda classe coinvolta nel join.
- *mustJoin*: è un attributo booleano e rappresenta la necessità o meno di effetuare il join. Infatti se il suo valore è settato a *true* significa che tra *firstElement* e *secondElement* vi è sovrapposizione di istanze e quindi vi è necessità di realizzarne la fusione. Se il suo valore è settato a *false* le due classi *firstElement* e *secondElement* sono disgiunte.
- *joineable*: è un attributo booleano e rappresenta la possibilità o meno di realizzare tra *firstElement* e *secondElement* un join diretto. Se il suo valore è settato a *true* allora il join risulta diretto, invece se il suo valore è settato a *false* allora il join è indiretto (oppure le due classi locali sono disgiunte).
- *joinAttribute*: è un oggetto di tipo **JoinAttribute** e rappresenta l'insieme degli attributi globali, sulla base dei quali realizzare il join tra *firstElement* e *secondElement*. La natura di questa proprietà risulterà più chiara in seguito, quando si descriverà l'interfaccia della classe JoinAttribute.
- *thirdElement*: è un oggetto di tipo **SourceClass** e rappresenta la classe intermedia di join, inserita esplicitamente dal progettista, quando si verifica la necessità di realizzare la fusione passando attraverso un join intermedio.

METODI

- *setJoineable()*: permette, ricevendo come parametro un valore booleano, di settare la proprietà *joineable*.
- *setMustJoin()*: permette, ricevendo come parametro un valore booleano, di settare la proprietà *mustJoin*.
- *setJoinA()*: associa alla proprietà *joinAttribute* l'oggetto di tipo JoinAttribute passato per parametro. Mediante questo metodo è possibile definire quali siano gli attributi globali su cui eseguire il join.
- *getJoinA()*: restituisce un oggetto di tipo JoinAttribute, che contiene le informazioni sugli attributi globali da utilizzare in fase di join.

- *setThird()*: questo metodo riceve come parametro un oggetto di tipo SourceClass e permette di associare alla proprietà *thirdElement* la classe locale da utilizzare in caso di join indiretto.

Classe JoinTable

La classe **JoinTable**, come dice il nome stesso, modella la struttura dati definita dalla JoinTable.

PROPRIETÀ

- *gClassName*: è una stringa contenente il nome della classe globale a cui è associata la Join Table.
- *globalSchema*: è una stringa contenente il nome dello schema globale a cui fa riferimento la classe globale di appartenenza della Join Table.
- *setJoinMap*: è un oggetto di tipo **Vector**. In esso sono collezionati tutti gli oggetti di tipo JoinMap che modellano le Join Map che costituiscono la Join Table. Si ricorda infatti che una Join Table è una tupla contenente tutte le possibili Join Map che si ottengono combinando a due a due le classi locali di una classe globale.

METODI

- *setJoinTable()*: il metodo in questione riceve come parametro una collezione (Vector) di oggetti di tipo JoinMap, che modella l'insieme delle JoinMap che è possibile calcolare automaticamente. Questa collezione viene associata alla proprietà *setJoinMap*, applicando per ogni oggetto JoinMap dell'insieme il metodo *addJoinMap()*. Questo metodo viene anche utilizzato per aggiungere uno alla volta alla proprietà *setJoinMap* gli oggetti JoinMap che modellano le Join Map esplicitate dal progettista, e che quindi non è stato possibile calcolare automaticamente.
- *getJoinMap()*: il metodo riceve come parametri due oggetti di tipo SourceClass e restituisce un oggetto di tipo JoinMap. Il suo compito è di individuare nella Join Table la Join Map corrispondente alle due classi locali passate per parametro. Se tale Join Map non esiste viene restituito un valore *null*.

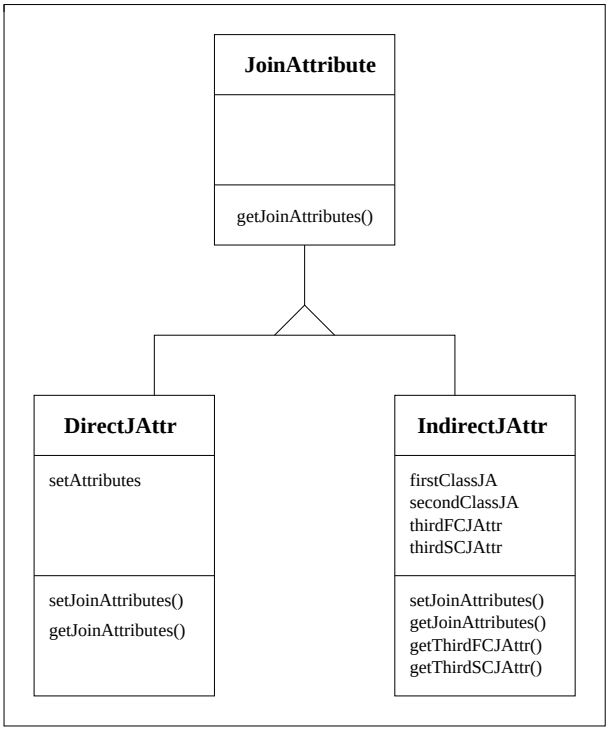

Figura 5.2: Modello ad oggetti degli attributi di join

La classe è provvista anche dei metodi *removeJoinMap()*, che rimuove da *setJoinMap* l'oggetto JoinMap passato come parametro, e *removeAll-JM()*, che svuota la proprietà *setJoinMap* da tutti gli oggetti JoinMap che la costituiscono.

Classe JoinAttribute

La classe **JoinAttribute** generalizza il concetto di attributo di join presente in una Join Map. La proprietà *joinAttribute* della classe JoinMap prevede un oggetto di tipo JoinAttribute come entry. Siccome poi la struttura dati costituita dagli attributi di join si differenzia a seconda che il join sia diretto o indiretto, si è creduto opportuno organizzare le classi che la modellano secondo la gerarchia rappresentata nella modellazione ad oggetti in Figura 5.2. La classe JoinAttribute è di tipo *abstract* e prevede nella sua interfaccia solo i metodi comuni ai due tipi di join, diretto ed

indiretto. Per vedere la loro implementazione è necessario però fare riferimento alle classi **DirectJAttr** e **IndirectJAttr**, che modellano gli attributi di join nel caso diretto ed indiretto rispettivamente. L'implementazione di questi metodi è specifica per le due classi, che in più prevedono proprietà e metodi propri. Tutto questo si è reso necessario perchè gli attributi di join (diretto o indiretto) assumono connotazioni molto diverse.

PROPRIETÀ

La classe non prevede proprietà nella sua interfaccia, questo perchè le classi specializzate **DirectJAttr** e **IndirectJAttr** sono molto diverse tra di loro e non condividono proprietà.

METODI

- *getJoinAttribute()*: è un metodo di tipo *abstract* che viene implementato opportunamente nelle classi specializzate. Restituisce una collezione di attributi di join.

Classe DirectJAttr

La classe **DirectJAttr** modella gli attributi di join nel caso di join diretto.

PROPRIETÀ

- *setAttributes*: è una collezione di attributi globali di join. In particolare ogni elemento del **Vector** *setAttributes*, è a sua volta un **Vector of GlobalAttribute**. Ogni elemento rappresenta un possibile insieme di attributi globali (che nel nostro sistema sono modellati mediante la classe GlobalAttribute) su cui effettuare il join. Vi è un'unica collezione per entrambe le classi coinvolte nel join, perchè nel caso di join diretto gli attributi globali di join sono gli stessi per le classi coinvolte, e quindi presentano anche il medesimo nome.

METODI

- *getJoinAttributes()*: questo metodo permette di reperire gli attributi globali su cui effettuare il join. Restituisce un oggetto di tipo Vector

of Vector of GlobalAttribute. Ciascun elemento di questa collezione (Vector) è a sua volta una collezione di attributi sulla base della quale è possibile realizzare un join diretto.

- *setJoinAttributes()*: questo metodo riceve come parametro un oggetto di tipo Vector of Vector of GlobalAttribute che permette di settare opportunamente la proprietà *setAttributes*.

Classe IndirectJAttr

La classe **IndirectJAttr** modella gli attributi di join nel caso di join indiretto.

PROPRIETÀ

- *firstClassJA*: è un oggetto di tipo **Vector of GlobalAttribute** e modella l'insieme degli attributi di join della prima classe (*firstElement* dell'oggetto JoinMap corrispondente) coinvolta nel join indiretto.
- *secondClassJA*: è un oggetto di tipo **Vector of GlobalAttribute** e modella l'insieme degli attributi di join della seconda classe (*secondElement* dell'oggetto JoinMap corrispondente) coinvolta nel join indiretto.
- *thirdFCJAttr*: rappresenta determinati attributi di join nella classe intermedia che permette di realizzare il join indiretto. In particolare prevede come entry un oggetto **TreeMap** che permette di esprimere il matching tra gli attributi di join della prima classe della Join Map e quelli della classe intermedia. Infatti, ogni elemento della collezione rappresentata nella proprietà in questione prevede come *key*, il nome di un attributo di join della prima classe e come *value*, l'oggetto GlobalAttribute che modella l'attributo globale, che realizza il matching, presente nella classe intermedia.
- *thirdSCJAttr*: contiene il riferimento a determinati attributi di join nella classe intermedia che permette di realizzare il join indiretto. In particolare prevede come entry un oggetto **TreeMap** che permette di esprimere, come nella proprietà *thirdFCJAttr* il matching tra gli attributi di join della seconda classe della Join Map e quelli della classe intermedia.

METODI

- *getJoinAttributes()*: permette di reperire gli attributi su cui realizzare il join indiretto. Restituisce un oggetto di tipo Vector con soli due elementi. Il primo è un oggetto di tipo Vector of GlobalAttribute e rappresenta l'insieme degli attributi globali di join della prima classe. Analogamente il secondo elemento è un oggetto di tipo Vector of GlobalAttribute e rappresenta l'insieme degli attributi globali di join della seconda classe.
- *setJoinAttributes()*: permette di settare opportunamente le proprietà *firstClassJA* e *secondClassJA*. Riceve come parametri due oggetti di tipo Vector of GlobalAttribute, il primo costituisce la entry per *firstClassJA*, il secondo costituisce la entry per *secondClassJA*.
- *getThirdFCJAttr()*: ritorna un oggetto TreeMap che permette di esprimere il matching tra gli attributi di join della prima classe e quelli della classe intermedia. Ciò viene realizzato come è stato spiegato per *thirdFCJAttr*.
- *getThirdSCJAttr()*: analogamente ritorna un oggetto TreeMap che permette di esprimere il matching tra gli attributi di join della seconda classe e quelli della classe intermedia.

Le classi **GlobalClass** e **BaseExtension** non sono state implementate per rendere possible il processo di fusione, ma ne risultano direttamente coinvolte: per questo motivo si è deciso di darne una breve descrizione. Di queste due classi verranno analizzati solamente le proprietà ed i metodi funzionali al processo di Object Fusion.

Classe GlobalClass

La classe **GlobalClass** descrive una classe globale.

PROPRIETÀ

- *joinTable*: è un oggetto di tipo **JoinTable** e rappresenta la JoinTable associata alla classe globale in questione.

METODI

- *getJoinTable()*: restituisce un oggetto di tipo JoinTable, ovvero restituisce la Join Table associata alla classe globale.

Classe BaseExtension

La classe **BaseExtension** descrive una base extension.

PROPRIETÀ

- *joinMap*: contiene un oggetto di tipo **Vector of JoinMap**. Gli elementi di questa collezione sono tutte le Join Map che si possono ottenere combinando le classi locali che costituiscono la base extension. Questa proprietà, come è facile intuire, è fondamentale nel processo di *Ricostruzione di una base extension*.

METODI

- *getJoinMap()*: a questo metodo vengono passati per parametro due oggetti di tipo SourceClass, che rappresentano due classi locali. Restituisce un oggetto di tipo JoinMap, che rappresenta la Join Map individuata dalle classi locali passate come parametro. Se almeno una delle due classi locali passate come parametro non appartiene alla base extension in questione, allora il metodo restituisce un valore *null*.

5.1.2 Il package joinMap

Il package joinMap contiene tutti i metodi e le procedure finalizzati alla definizione automatica o alla dichiarazione esplicita delle Join Map, e di conseguenza delle Join Table. Questo package fa parte dei package che modellano i moduli di MOMIS.
La classe più importante contenuta nel package è la classe **JoinMapPanel**, la quale si occupa del calcolo automatico delle Join Map e quindi del calcolo automatico delle Join Table. Inoltre la classe prevede anche alcuni metodi finalizzati alla realizzazione dell'interfaccia grafica del modulo joinMap. I metodi della classe JoinMapPanel che si occupano della realizzazione dell'interfaccia e le

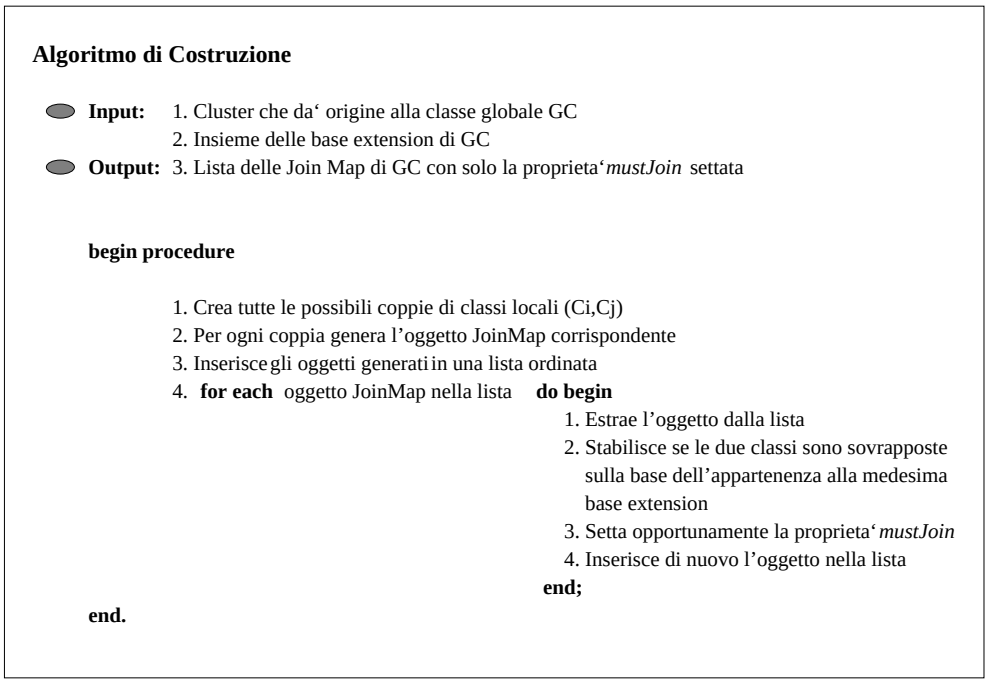

Figura 5.3: Algoritmo di Costruzione

altre classi del package (anch'esse funzionali alla realizzazione dell'interfaccia) verranno presi in rassegna nella prossima sezione.

I passi logici svolti dalla classe JoinMapPanel per eseguire il calcolo automatico delle JoinMap e della JoinTable, seguono quelli descritti negli algoritmi espressi in pseudo codice riportati qui di seguito.

La prima operazione che viene fatta è quella di organizzare le classi locali sulle quali si lavora in una lista ordinata di coppie in cui sia già evidente dove sia necessario effettuare il join o meno. Facendo riferimento all'*Algoritmo di Costruzione* in Figura 5.3, si vede che come prima cosa le classi locali del cluster che si sta analizzando vengono combinate in modo da creare tutte le coppie possibili, ed il risultato viene posto in una lista ordinata. Questo viene svolto dalla classe JoinMapPanel tramite il metodo *makeJoinMap()*. Questo metodo prevede come parametro il nome di una classe globale e si occupa di: ricavare le classi locali che costituiscono il cluster che genera la classe globale con il nome dato; combinare le classi locali a coppie facendo in modo che non esistano coppie uguali (il caso (C_i,C_j) e (C_j,C_i) non si deve verificare); generare gli oggetti di tipo JoinMap corrispondenti alle coppie create; inserire gli oggetti JoinMap in un oggetto di tipo Vector che viene restituito. In seguito la classe JoinMapPanel, tramite il meto-

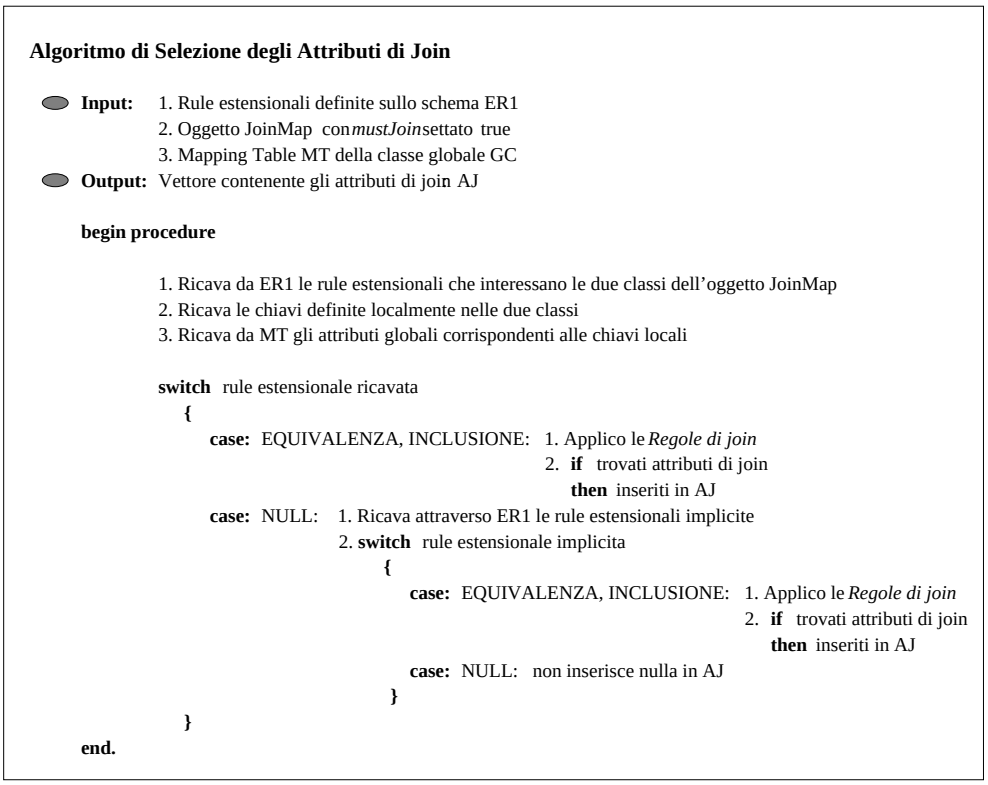

Figura 5.4: Algoritmo di Selezione degli Attributi di Join

do *costruisci()*, setta opportunamente la proprietà *mustJoin* degli oggetti JoinMap presenti nella lista generata in precedenza. Questo viene realizzato applicando ad ogni oggetto JoinMap il metodo *findCommon()* che testa se le due classi locali coinvolte appartengono o meno alla medesima base extension. Il risultato di queste operazioni è nuovamente un oggetto di tipo Vector i cui elementi sono oggetti JoinMap con la proprietà *mustJoin* settata.

I passi successivi sono quelli più complessi perchè consistono nell'implementazione delle *Regole di join*, come è evidenziato nell'*Algoritmo di Selezione degli Attributi di Join* in Figura 5.4.

Il metodo *getExtR()*, ricevendo come parametro un oggetto JoinMap restituisce un oggetto di tipo Vector i cui elementi sono oggetti di tipo **ExtRule**, che rappresentano gli assiomi estensionali espressi sulle due classi locali che costituiscono la Join Map. Il risultato di questa ricerca viene poi integrato tramite il metodo *superCl()* che permette di stabilire se tra le due classi passate per parametro sussiste una relazione di specializzazione, che dà vita ad una relazione estensionale di inclusione.

Il metodo *findAttr()*, ricevendo come parametro un oggetto di tipo SourceClass corrispondente ad una delle due classi locali della Join Map, restituisce un oggetto di tipo Vector i cui elementi sono a loro volta oggetti Vector of GlobalAttribute, che rappresentano le chiavi della classe locale espresse in termini globali. In particolare tra queste chiavi vi sono anche quelle ereditate, e questo risultato viene raggiunto grazie all'applicazione del metodo ricorsivo *superGAK()*. Come si vede si ha già a che fare con attributi globali (gli attributi di join devono essere di natura globale): è sempre il metodo *findAttr()* ad occuparsi di ricercare gli attributi globali che mappano gli attributi locali che costituiscono le chiavi. Ovviamente questi metodi vengono sfruttati per entrambe le classi locali che costituiscono la Join Map.
Eventuali relazioni estensionali implicite tra le due classi locali sono ricavate tramite i metodi *findIncluSx()* e *findEquiv()*. Questi metodi, esplorando ricorsivamente tutte le relazioni estensionali presenti nello schema globale, ricercano se vi sono relazioni di inclusione e di equivalenza rispettivamente, che coinvolgono indirettamente le classi della Join Map in esame.
L'implementazione vera e propria delle *Regole di join* viene realizzata tramite il metodo *qualiAtt()*[1], che prevede come parametro l'oggetto JoinMap coinvolto e la stringa che indica il tipo di relazione estensionale che lega le due classi locali della Join Map, e restituisce un oggetto Vector of Vector of GlobalAttribute che raccoglie gli attributi di join ricavati. In particolare tale implementazione viene attuata dal metodo in questione, mediante l'applicazione dei seguenti metodi:

- *theSame()*: permette di stabilire se le due classi hanno chiavi semanticamente omogenee tra di loro.
- *findKey()*: permette di stabilire se tra gli attributi locali non chiave di una classe locale coinvolta, ve ne è un insieme che risulta essere semanticamente omogeneo ad una chiave dell'altra classe coinvolta. Se questo insieme viene trovato viene restituito l'insieme di attributi globali che lo mappa.
- *inglobe()*: permette di stabilire se l'insieme di attributi globali che mappa la chiave di una classe contiene l'insieme degli attributi globali che mappa la chiave dell'altra classe. Viene restituito il vettore di attributi globali contenuti. Questo metodo viene utilizzato nel caso di chiavi composte, per stabilire se una chiave è contenuta nell'altra.

L'ultimo passo che rimane da fare è quello di raccogliere e riorganizzare opportunamente i risultati ottenuti dall'applicazione degli algoritmi descritti in precedenza, e ciò viene fatto seguendo quanto descritto nell'*Algoritmo di Creazione*

[1]Il caso di chiavi mappate con uno *union mapping* non è stato gestito, in quanto questo tipo di mapping è ancora in fase di studio ed i metodi della classe UnionMapping sono ancora in fase di progettazione.

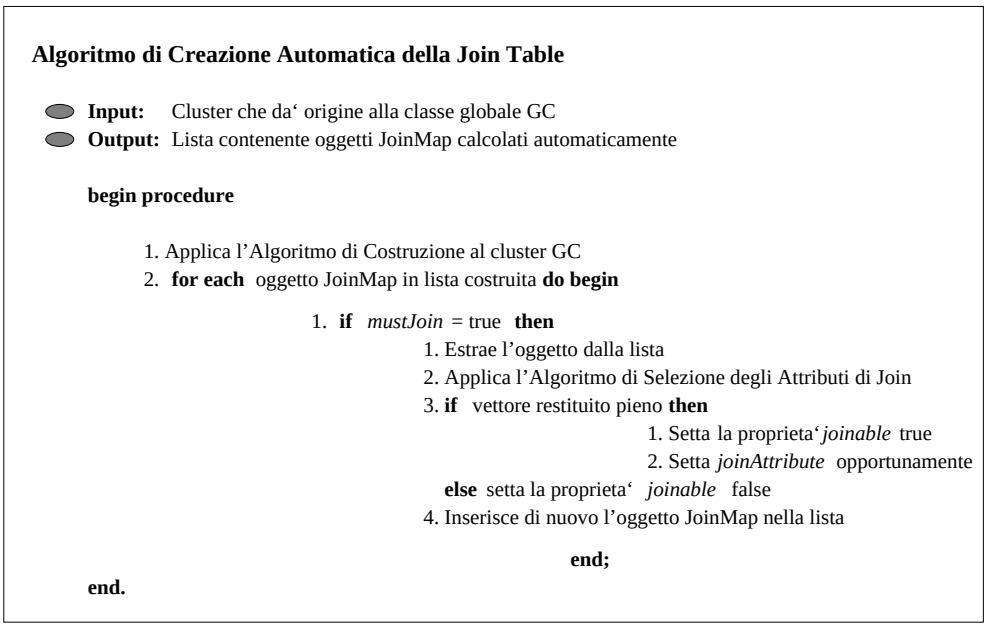

Figura 5.5: Algoritmo di Creazione Automatica della Join Table

Automatica della Join Table rappresentato in Figura 5.5.
Il metodo preposto all'esecuzione dei passi logici descritti nell'algoritmo è il metodo *autoJT()*. Questo metodo, riceve come parametro il nome della classe globale di cui si vuole calcolare la Join Table e restituisce un oggetto di tipo Vector i cui elementi sono oggetti di tipo JoinMap con le proprietà opportunamente settate. Ovviamente questo settaggio riguarda solo il caso automatico: per proprietà come *thirdElement* è necessario l'intervento del progettista.
Il metodo in questione realizza quanto richiesto nell'*Algoritmo di Costruzione* tramite il metodo *costruisci()*, già descritto in precedenza. Per ogni elemento del vettore di oggetti JoinMap ottenuto viene applicato il metodo *riempi()* che realizza il settaggio delle proprietà descritto prima, sulla base dei risultati restituiti dal metodo *qualiAtt()*, realizzando quindi quanto richiesto dall'*Algoritmo di Selezione degli Attributi di Join*.

5.2 L'interfaccia grafica

L'interfaccia grafica di MOMIS è realizzata da SI-Designer (Source Integrator Designer), che è un tool di supporto al progettista per l'integrazione semi-automatica degli schemi. Questa interfaccia è costituita da una serie di pannelli in sequen-

za, ognuno dei quali relativo ad un modulo diverso di SI-Designer, preposto all'esecuzione di una delle fasi dell'integrazione.

5.2.1 SI-Designer

L'interazione tra il tool SI-Designer ed i wrapper viene realizzata tramite l'architettura CORBA: i wrapper sono dei *servant-object* che rispondono ai messaggi inviati da SI-Designer, che è il client. Analogamente è stata implementata l'interazione di SI-Designer con i tool esterni WordNet e ODB-Tools, che sono stati "inglobati" in oggetti CORBA.
Per rendere il sistema più aperto, il progettista ha la possibilità di creare oggetti CORBA contenenti tutte le informazioni relative ad uno schema globale. Questi oggetti sono istanze della classe **GlobalSchema**. SI-Designer è in grado di accedere agli oggetti GlobalSchema attraverso un oggetto istanza della classe **GlobalSchemaProxy** che funge da *proxy*. In questo modo, è possibile mettere in rete gli schemi globali di MOMIS e renderli consultabili da uno o più Query Manager client, o da più applicazioni.
L'architettura di SI-Designer, rappresentata in Figura 5.6, prevede sette moduli preposti alla realizzazione delle fasi di integrazione degli schemi:

- SAM, *Sources Acquisition Module*: acquisisce gli schemi delle sorgenti da integrare espressi in ODL_{I^3}.

- SIM, *Sources Integrator Module*: estrae relazioni intra-schema, valida tramite ODB-Tools le relazioni inter-schema e quelle aggiunte dal progettista e inferisce nuove relazioni attraverso ODB-Tools. Il tutto porta alla generazione del Common Thesaurus.

- SLIM, *Schemata Lessical Integrator Module*: grazie al sistema lessicale WordNet estrae relazioni terminologiche inter-schema. Anche questa attività partecipa alla generazione del Common Thesaurus.

- ARTEMIS: calcola l'affinità tra le classi locali e crea di conseguenza i cluster che porteranno alla costituzione delle classi globali dello schema.

- TUNIM, *TUNIng of mapping-table Module*: crea le classi globali (e di conseguenza la corrispondente Mapping Table) a partire dai cluster e dal Common Thesaurus.

- EXTM: gestisce la conoscenza estensionale agevolando l'inserimento degli assiomi estensionali e calcolando le base extension e la gerarchia estensionale.

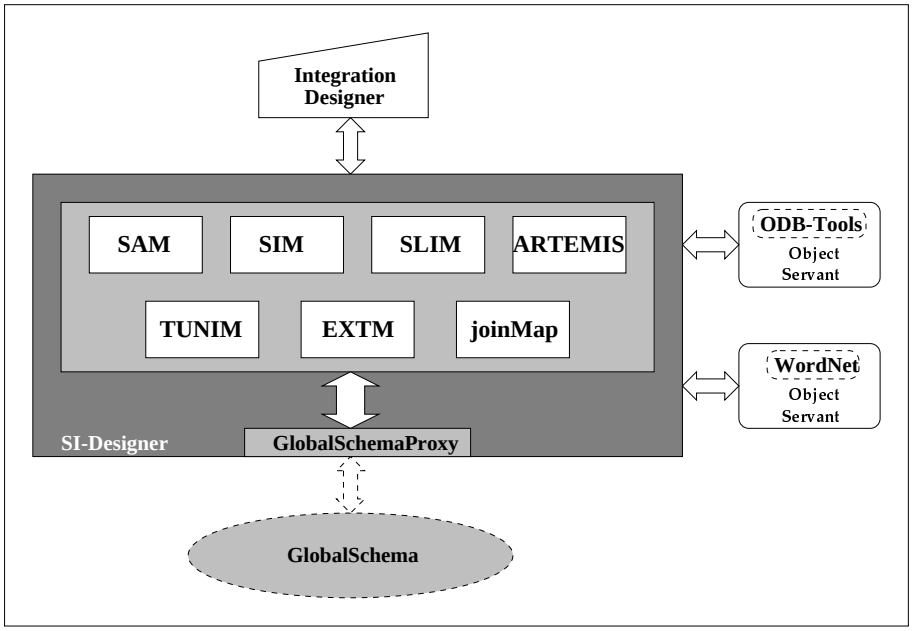

Figura 5.6: Architettura di SI-Designer

- joinMap: provvede alla definizione automatica delle Join Map e gestisce la dichiarazione esplicita di queste.

5.2.2 Il pannello "Join Map"

L'interfaccia grafica di SI-Designer è costituita da un pannello principale in cui sono inseriti in sequenza una serie di pannelli, ognuno dei quali selezionabile clickando su di un'etichetta che lo identifica. Selezionando l'ultima di queste etichette (che presenta il testo "Join Map") si seleziona il pannello del modulo **joinMap**. Ciò che appare è visibile in Figura 5.7. Il pannello è suddiviso in tre parti fondamentali:

- sulla sinistra vi è una struttura ad albero (realizzata mediante la classe del package joinMap **GlobalClassTree**) che rappresenta la classe globale attualmente selezionata e le classi locali che la costituiscono.

- in basso a destra vi è una tabella inizialmente vuota (ottenuta grazie alla classe **JoinTable**), realizzata per visualizzare la Join Table.

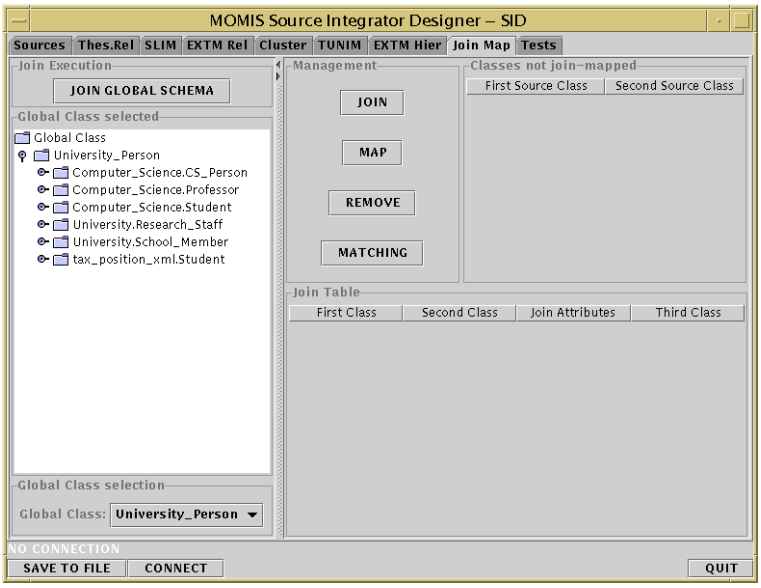

Figura 5.7: Il pannello “Join Map”

- in alto a destra vi è una tabella vuota (ottenuta mediante la classe **NotJoinTable**) finalizzata a contenere le coppie di classi locali sulle quali non è possibile individuare gli attributi di join automaticamente.

La struttura ad albero sulla sinistra può essere espansa clickando sulle icone a fianco delle classi. Quello che si ottiene è la visualizzazione degli attributi globali di ciascuna classe locale. Si è fatta la scelta di visualizzare gli attributi globali e non quelli locali, perchè gli attributi di join che bisogna individuare (attività a cui è preposto il modulo joinMap) hanno natura globale. Però, per rendere questa visualizzazione più chiara e lontana da fraintendimenti, per alcuni di questi attributi, sono mostrate informazioni aggiuntive. Infatti gli attributi globali che mappano attributi locali costituenti una chiave per la classe, oltre al loro nome visualizzato prevedono: l’insieme di attributi locali che mappano, la scritta “(K)” indicante che si tratta di una chiave, la scritta“(I)” qualora l’attributo sia ereditato.
In alto a sinistra è visibile il pulsante “JOIN GLOBAL SCHEMA”. L’evento generato, premendo questo pulsante, è quello di calcolare le Join Map automatiche per tutte le classi locali, che costituiscono lo schema globale. Il risultato di questa operazione finisce nello spazio dedicato a contenere la Join Table. Le coppie di classi che non generano le corrispondenti Join Map automaticamente finiscono nella tabella in alto a destra. Lo stesso evento viene generato clickando il pulsante “JOIN”, con la sola differenza che il calcolo delle Join Map viene svolto

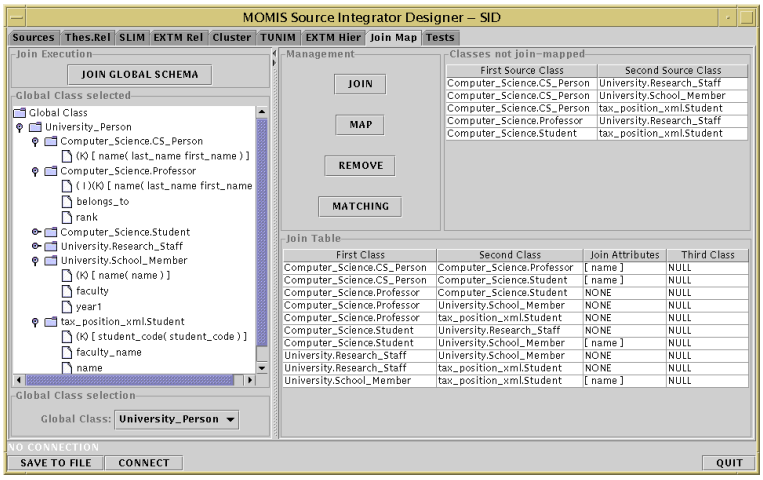

Figura 5.8: Il pannello "Join Map": Join Table automatica

solo sulle classi locali della classe globale selezionata. È evidente che queste operazioni coincidono con l'esecuzione del metodo della classe JoinMapPanel *autoJT()* descritto nella Sezione 5.1.2.
Il risultato delle attività appena descritte è visibile in Figura 5.8.

Per definire esplicitamente una Join Map, bisogna selezionare la coppia di classi locali che interessa dalla tabella in alto a destra e premere il pulsante "MAP". Questa operazione provoca l'apertura di una finestra (Figura5.9) munita di tutte le funzionalità finalizzate all'esplicitazione degli attributi di join. Se viene selezionata la sezione in alto è possibile provvedere ad esplicitare gli attributi di join nel caso di join diretto. Infatti la sezione presenta due caselle combinate che permettono di selezionare gli attributi globali.
Se viene selezionata la sezione in basso è possibile provvedere ad esplicitare gli attributi di join e la classe intermedia, nel caso di join indiretto.
Nella parte superiore della sezione vi è una casella combinata, grazie alla quale è possibile selezionare la classe locale intermedia, scegliendo sull'intero panorama delle classi locali che costituiscono lo schema globale.
Gli attributi di join vanno selezionati nelle caselle combinate sottostanti, facendo attenzione ad esprimere per ogni attributo globale di una classe locale selezionato (premendo il pulsante "OK" corrispondente) il matching con il corrispondente attributo globale della classe intermedia (anch'esso selezionato dalla casella combinata premendo poi il pulsante "OK"). Se di volta in volta questo matching non viene esplicitato, appaiono delle finestre informative che avvertono il progettista di questa dimenticanza.
Ai piedi di ogni sezione vi sono: il pulsante "APPLY", che, una volta premuto,

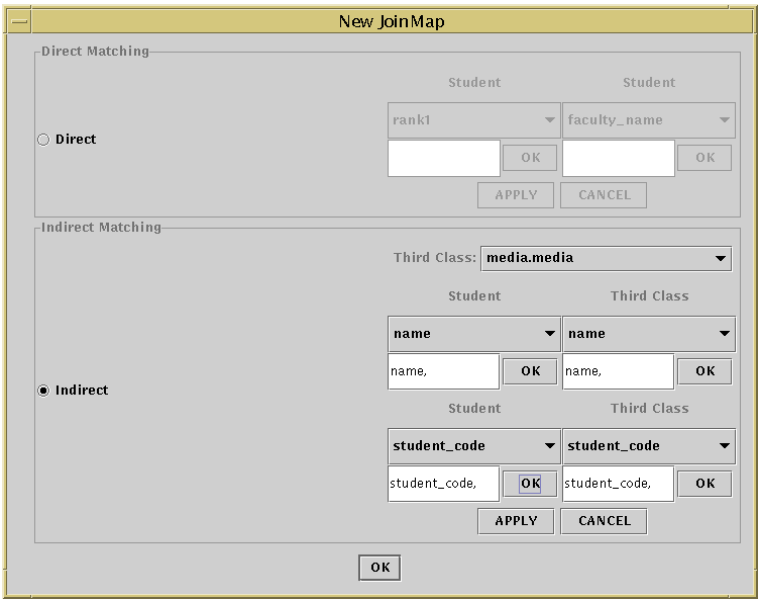

Figura 5.9: Il pannello “Join Map”: dichiarazione esplicita delle Join Map

provvede ad inserire la Join Map nello schema ed a visualizzare la nuova riga corrispondente nella Join Table dell’interfaccia; il pulsante “CANCEL”, che una volta clickato cancella le esplicitazioni fatte fino a quel momento.
Il matching definito esplicitamente dal progettista tra gli attributi globali delle classi locali e gli attributi della classe intemedia può essere meglio visualizzato selezionando la riga interessata nella Join Table e premendo il pulsante “MATCHING”.
Questa operazione provoca l’apertura della finestra in Figura 5.10, che visualizza il matching tramite una tabella (realizzata mediante la classe **MatchingTable**).
Il pannello “Join Map” è dotato inoltre del pulsante “REMOVE”, che, selezionando prima una riga dalla Join Table, permette di rimuovere la Join Map corrispondente, qualora non la si ritenga soddisfacente.

Può risultare interessante, arrivati a questo punto, prestare attenzione al flusso dei dati.
Come è già stato detto, premendo il pulsante “JOIN GLOBAL SCHEMA” o il pulsante “JOIN”, si fa eseguire il metodo della classe JoinMapPanel *autoJT()*, che si occupa del calcolo automatico delle Join Map. Questo metodo restituisce una collezione di oggetti JoinMap. Per associare questa collezione alla classe globale corrispondente, viene istanziato un oggetto di tipo JoinTable la cui proprietà *setJoinMap* viene popolata proprio dalla collezione restituita. Ciò avviene tramite

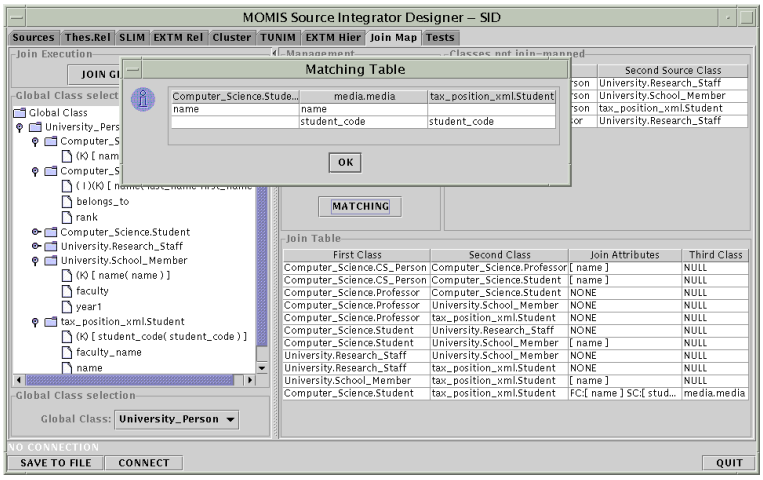

Figura 5.10: Il pannello "Join Map": il matching tra gli attributi di join

il metodo della classe JoinTable *setJoinTable()*. L'oggetto JoinTable così ottenuto costituisce la entry per la proprietà *joinTable* dell'oggetto GlobalClass che modella la classe globale che si sta considerando. Inoltre è necessario associare gli oggetti JoinMap ottenuti alle corrispondenti base extension, ovvero è necessario andare a popolare la proprietà *joinMap* degli oggetti BaseExtension coinvolti dalle classi locali delle Join Map nella collezione. Ciò viene realizzato tramite il metodo della classe JoinMapPanel *fillBE()*. Analogamente ogni volta che viene esplicitata una nuova Join Map, si provvede ad aggiungerla alla Join Table ed alla base extension corrispondenti tramite i metodi *addJoinMap()* della classe JoinTable, e *addInBE()* della classe JoinMapPanel. *addJoinMap()* prevede come parametro l'oggetto JoinMap corripondente alla Join Map definita e provvede ad aggiungerlo alla collezione contenuta nella proprietà *setJoinMap* dell'oggetto JoinTable interessato. *addInBE()* allo stesso modo aggiunge l'oggetto JoinMap passato come parametro alla collezione contenuta nella proprietà *joinMap* dell'oggetto BaseExtension interessato.
Analogamente, quando si provvede alla rimozione di una Join Map (pulsante "REMOVE"), si provvede a togliere l'oggetto JoinMap corrispondente dagli oggetti JoinTable e BaseExtension interessati. Questo avviene tramite i metodi *removeJoinMap()* della classe JoinTable e *removeFromBe()* della classe JoinMapPanel.

5.3 Il software

Il software prodotto in questa tesi è stato sviluppato utilizzando la versione 1.3 del Java Development Kit della Sun (**jdk1.3**). Approssimativamente sono state prodotte 4000 righe di codice commentato. I commenti sono stati realizzati usando il formalismo richiesto dal componente *Javadoc*, che produce in modo automatico una documentazione in formato HTML con collegamenti ipertestuali delle classi create. Circa il 60% del software prodotto è finalizzato alla realizzazione dell'interfaccia grafica del modulo joinMap, mentre il restante 40% serve per la definizione automatica delle Join Map. L'impatto dell'interfaccia sulle dimensioni del codice è considerevole: ciò è reso indispensabile dal fatto che l'intervento del progettista nella definizione delle Join Map è fondamentale, perciò si è ritenuto opportuno realizzare un'interfaccia il più possibile user-friendly e provvista di tutti i controlli di coerenza richiesti.
In Appendice E è possibile trovare il codice realizzato per l'implementazione delle classi JoinMap e JoinTable.

Tutto il codice prodotto e la relativa documentazione possono essere trovati nel direttorio */export/home/progetti.comuni/tesi/micol/sw* del server "*Sparc20*" del Dipartimento di Scienze dell'Ingegneria.

Conclusioni

Come è stato illustrato nei primi due capitoli di questa tesi, l'obiettivo del sistema MOMIS è quello di realizzare un mediatore in grado di attuare l'integrazione di un insieme di sorgenti eterogenee ed autonome, generando una vista globale che l'utente può interrogare senza possedere un'effettiva conoscenza delle diverse sorgenti.

Gli argomenti trattati in questa tesi sono focalizzati sulla gestione della conoscenza intensionale ed estensionale (in possesso del sistema MOMIS), col fine di reperire, in modo automatico, le informazioni semanticamente significative che permettono di identificare istanze facenti riferimento alla medesima entità del mondo reale. Il risultato di questi studi è funzionale al Query Manager che, per realizzare il processo di Object Fusion (tramite una serie di operazioni di join) in fase di esecuzione della query, necessita di criteri che gli permettano di riconoscere istanze della medesima entità del mondo reale. In particolare sono state definite e formalizzate una serie di regole, le *Regole di join*, che, date due classi locali da integrare, permettono di individuare i campi semanticamente identificativi su cui il Query Manager può effettuare il join, qualora sia richiesta una fusione delle istanze. Per implementare le *Regole di join* è stato realizzato il modulo software joinMap. Siccome, non sempre le *Regole di join* sono in grado di portare ad un'individuazione automatica dei campi identificativi su cui effettuare il join, l'intervento del progettista è indispensabile (rendendo, quindi, il reperimento delle informazioni significative semi-automatico). Quindi, si è resa necessaria la realizzazione, per il modulo joinMap, di un'interfaccia grafica, che permettesse l'inserimento esplicito (quando opportuno), da parte del progettista, di tutte le informazioni funzionali al processo di Object Fusion.
Gli studi effettuati durante la realizzazione hanno anche portato alla definizione di due importanti strutture dati: le Join Map e la Join Table. Data una coppia di classi locali, la corrispondente Join Map ne rappresenta gli attributi identificativi su cui effettuare il join, se questo è necessario. Data una classe globale, la corrispondente Join Table ne raccoglie tutte le Join Map ottenute combinando a due a

due le classi locali appartenenti al cluster che l'ha generata.
Sulla base dell'ipotesi che, classi appartenenti a cluster diversi, sono disgiunte tra loro, le *Regole di join* sono state definite su una classe globale, quindi, quando la fase di integrazione degli schemi è già terminata ed ha prodotto i suoi risultati.
In fase di realizzazione di questa tesi, ci si è resi conto del forte significato estensionale ed intensionale racchiuso nella relazione di *joinable* tra classi. Questo ha portato a pensare che questa conoscenza possa essere utilizzata in modo attivo in fase di integrazione degli schemi, nel processo di creazione dei cluster. Quindi i prossimi sviluppi delle teorie elaborate e del modulo progettato in questa tesi, dovranno essere rivolti allo studio di tecniche che permettano alla relazione di *joinable* tra due classi di intervenire nel processo di creazione dei cluster.

Appendice A

Glossario I^3

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

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

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

A.1 Architettura

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

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

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

A.2 Servizi

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

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

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

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

A.3 Risorse

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

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

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

A.4 Ontologia

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

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

Appendice B

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

Si riporta la descrizione in BNF del linguaggio descrittivo ODL$_{I^3}$. Essendo questo una estensione del linguaggio standard ODL, si riportano in questa appendice solo le parti che differiscono dall'ODL originale, rimandando invece a quest'ultimo per le parti in comune.

⟨interface_dcl⟩ : : = ⟨interface_header⟩ {[⟨interface_body⟩]};
⟨interface_header⟩ : : = **interface** ⟨identifier⟩
[⟨inheritance_spec⟩]
[⟨type_property_list⟩]
⟨inheritance_spec⟩ : : = : ⟨scoped_name⟩ [,⟨inheritance_spec⟩]
⟨type_property_list⟩ : : = **(** [⟨source_spec⟩] [⟨extent_spec⟩]
[⟨key_spec⟩] [⟨f_key_spec⟩] **)**
⟨source_spec⟩ : : = **source** ⟨source_type⟩ ⟨source_name⟩
⟨source_type⟩ : : = **relational** | **nfrelational** | **object** | **file**
⟨source_name⟩ : : = ⟨identifier⟩
⟨extent_spec⟩ : : = **extent** ⟨extent_list⟩
⟨extent_list⟩ : : = ⟨string⟩ | ⟨string⟩ **,** ⟨extent_list⟩
⟨key_spec⟩ : : = **key**[**s**] ⟨key_list⟩
⟨f_key_spec⟩ : : = **foreign_key** ⟨f_key_list⟩
. . .

⟨attr_dcl⟩ ::= **[readonly] attribute**
⟨domain_type⟩ ⟨attribute_name⟩
[⟨fixed_array_size⟩] [⟨mapping_rule_dcl⟩]
⟨mapping_rule_dcl⟩ ::= **mapping_rule** ⟨rule_list⟩
⟨rule_list⟩ ::= ⟨rule⟩ | ⟨rule⟩**,**⟨rule_list⟩
⟨rule⟩ ::= ⟨local_attr_name⟩ | **'**⟨identifier⟩**'**
⟨and_expression⟩ | ⟨or_expression⟩
⟨and_expression⟩ ::= **(** ⟨local_attr_name⟩ **and** ⟨and_list⟩ **)**
⟨and_list⟩ ::= ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **and** ⟨and_list⟩
⟨or_expression⟩ ::= **(** ⟨local_attr_name⟩ **or** ⟨or_list⟩ **)**
⟨or_list⟩ ::= ⟨local_attr_name⟩ | ⟨local_attr_name⟩ **or** ⟨or_list⟩
⟨local_attr_name⟩ ::= ⟨source_name⟩**.**⟨class_name⟩**.**⟨attribute_name⟩
. . .
⟨relationships_list⟩ ::= ⟨relationship_dcl⟩**;** | ⟨relationship_dcl⟩**;** ⟨relationships_list⟩
⟨relationships_dcl⟩ ::= ⟨local_attr_name⟩ ⟨relationship_type⟩ ⟨local_attr_name⟩
⟨relationship_type⟩ ::= **syn** | **bt** | **nt** | **rt**
. . .
⟨rule_list⟩ ::= ⟨rule_dcl⟩**;** | ⟨rule_dcl⟩**;** ⟨rule_list⟩
⟨rule_dcl⟩ ::= **rule** ⟨identifier⟩ ⟨rule_pre⟩ **then** ⟨rule_post⟩
⟨rule_pre⟩ ::= ⟨forall⟩ ⟨identifier⟩ **in** ⟨identifier⟩ **:** ⟨rule_body_list⟩
⟨rule_post⟩ ::= ⟨rule_body_list⟩
⟨rule_body_list⟩ ::= **(** ⟨rule_body_list⟩ **)** | ⟨rule_body⟩ |
⟨rule_body_list⟩ **and** ⟨rule_body⟩ |
⟨rule_body_list⟩ **and (** ⟨rule_body_list⟩ **)**
⟨rule_body⟩ ::= ⟨dotted_name⟩ ⟨rule_const_op⟩ ⟨literal_value⟩ |
⟨dotted_name⟩ ⟨rule_const_op⟩ ⟨rule_cast⟩ ⟨literal_value⟩ |
⟨dotted_name⟩ **in** ⟨dotted_name⟩ |
⟨forall⟩ ⟨identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩ |
exists ⟨identifier⟩ **in** ⟨dotted_name⟩ **:** ⟨rule_body_list⟩
⟨rule_const_op⟩ ::= = | ≥ | ≤ | > | <
⟨rule_cast⟩ ::= **(**⟨simple_type_spec⟩**)**
⟨dotted_name⟩ ::= ⟨identifier⟩ | ⟨identifier⟩**.**⟨dotted_name⟩
⟨forall⟩ ::= **for all** | **forall**

Appendice C

Esempio di riferimento in ODL$_{I^3}$

Di seguito é riportata la descrizione, attraverso il linguaggio ODL$_{I^3}$, dell'esempio di riferimento citato nella trattazione della tesi.

Sorgente `University` (UNI):

```
interface Research_Staff
( source relational University
  extent Research_Staff
  key  (name)
  foreign_key (dept_code)
  references Depatment (dept_code);
  foreign_key (section_code)
  references Section (section_code)
{ attribute string name;
  attribute string e_mail;
  attribute integer dept_code;
  attribute integer section_code; };
```

```
interface School_Member
( source relational University
  extent School_Member
  keys (name))
 {attribute string 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;};
```

```
interface Section
( source relational University
  extent Section
  key section_code
  foreign_key (room_code)
  references Room (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; };
```

Sorgente `Computer_Science` (CS):

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

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

interface Location
( source object Computer_Science
  extent Locations
  keys city, street, county, number)
{ attribute string city;
  attribute string street;
  attribute string county;
  attribute signed long number; };

interface Professor : CS_Person
 ( source object Computer_Science
   extent Professors )
  {attribute Office belong_to;
  attribute string rank;};

interface Office
( source object Computer_Science
  extent Offices
  key description )
{ attribute string description;
  attribute Location address;};

interface Course
( source object Computer_Science
  extent Courses
  key course_name )
{ attribute string course_name;
  attribute Professor taught_by; };
```

Sorgente `Tax_Position_xml` (TP):

```
interface Student
( source semistructured tax_Position
  extent Students
  key student_code )
{ attribute string name;
  attribute integer student_code;
  attribute string faculty_name;
  attribute integer tax_fee; };

interface ListOfStudent
( source semistructured tax_position_xml
  extent ListOfStudent)
  attribute set <Student> Student;
```

Appendice D

L'architettura CORBA

CORBA [29] (*Common Object Request Broker Architecture*) è un'architettura standard, distribuita e ad oggetti sviluppata dall'Object Management Group (OMG). Dal 1989 l'obiettivo del gruppo OMG è stato la progettazione di una architettura per un *software bus* aperto, chiamato *Object Request Broker* (ORB), sul quale oggetti diversi potessero interagire via rete, indipendentemente dal sistema operativo in cui sono stati implementati. Questo standard permette a più oggetti di invocarne altri senza conoscerne l'esatta locazione o in quale linguaggio sono stati implementati.
Il linguaggio utilizzato per definire un oggetto CORBA è l'IDL (*Interface Definition Language*) mentre gli ORB comunicano attraverso il protocollo standard IIOP (*Internet InterORB Protocol*), definito sempre dall'OMG.

Gli oggetti CORBA si differenziano dagli oggetti creati con altri linguaggi per i seguenti aspetti:

- possono essere localizzati in qualsiasi punto della rete;
- possono interagire con oggetti implementati su piattaforme HW/SW diverse, purché ovviamente siano sempre oggetti CORBA;
- possono essere scritti in qualsiasi linguaggio di programmazione per il quale è stato definito il *mapping* con il linguaggio standard IDL (attualmente i linguaggi utilizzabili includono Java, C++, C, Smalltalk, COBOL e ADA).

Come funziona

Il diagramma di Figura D.1 mostra come un client manda un *messaggio* (inteso come esecuzione di un metodo di un altro oggetto) ad un oggetto CORBA implementato in un server, chiamato *servant-object.* Un client può essere un qualsiasi programma (anche un oggetto CORBA) che invoca un metodo di un servant-object.

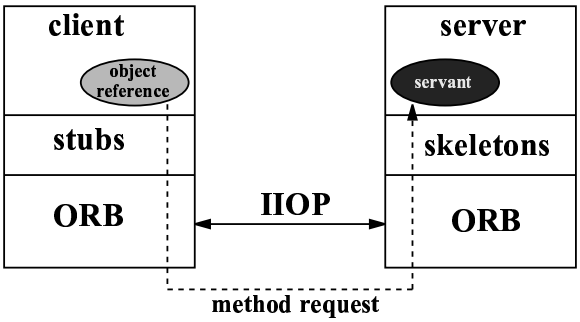

Figura D.1: Invocazione di un metodo di un oggetto CORBA remoto

Per invocare il metodo, il client utilizza un *object reference* dell'oggetto CORBA che vuole invocare. Se l'oggetto CORBA è locale, l'object reference è un puntatore ad un oggetto altrimenti, se l'oggetto CORBA è remoto, l'object reference punta ad una *stub function* senza che il client se ne accorga: per il client l'object reference è *sempre* un puntatore ad un oggetto. É l'apparato basato sugli ORB che rende possibile tutto questo.
L'invocazione di un metodo di un oggetto CORBA remoto da parte di un client avviene in questo modo:

1. il client invoca un metodo dell'oggetto CORBA utilizzando l'object reference;
2. la stub function puntata dall'object reference identifica, attraverso l'ORB locale, la macchina sulla quale si trova il servant-object CORBA, che è in attesa di ricevere messaggi;
3. l'ORB locale chiede all'ORB remoto di stabilire una connessione con l'oggetto CORBA;
4. ottenuta la connessione, l'ORB locale manda all'ORB remoto l'object reference della stub function e i parametri per il metodo da invocare;
5. l'ORB remoto passa la richiesta di esecuzione del metodo, assieme ai parametri, al servant-object che eseguirà il metodo invocato;
6. i risultati ed eventuali eccezioni vengono ritornate all'ORB locale lungo lo stesso percorso.

Il client non sa dove si trova il servant-object CORBA, non ne conosce i dettagli implementativi e nemmeno quale ORB è stato usato per stabilire la connessione.

Il Naming Service

Un client può invocare un oggetto CORBA remoto attraverso un object reference: ma come fa ad ottenere l'object reference? L'architettura CORBA mette a disposizione diversi modi per ottenere il reference. Uno di questi (semplice e flessibile) è il *Naming Service*, uno dei servizi standard implementato negli ORB. Il principio su cui si basa è semplice: assegnare un nome ad ogni oggetto CORBA creato e memorizzarlo in un *registro* di nomi.
In particolare, quello che occorre fare è attivare un *naming server* (un'applicazione fornita assieme alle librerie che permettono di creare oggetti CORBA in uno specifico linguaggio) sulla macchina in cui si vogliono creare oggetti CORBA accessibili in remoto: ogni oggetto CORBA creato dovrà poi registrarsi nel naming server, che gestisce il registro degli oggetti CORBA su quella macchina.
I nomi degli oggetti possono essere organizzati in una struttura ad albero proprio come i file sono organizzati in directory. Per accedere ad un determinato oggetto CORBA , il client esegue due sole operazioni:

1. chiedere all'ORB locale di connettersi ad un naming server (naturalmente, il naming server gira su una macchina remota collegata in rete e il client dovrà indicare all'ORB l'indirizzo e la porta per accedere al servizio);

2. ottenuta la connessione, attraverso l'ORB chiedere al naming server un object reference all'oggetto CORBA registrato sotto un certo nome.

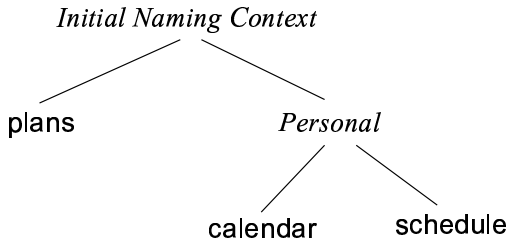

Figura D.2: Esempio di albero creato dal naming server

Per esempio, nella Figura D.2 è rappresentata la struttura ad albero memorizzata presso un naming server: si notano i *naming context* (equivalenti alle directory per i file system) *Initial Naming Context* (sempre presente) e *Personal*, mentre gli oggetti CORBA registrati sono `plans`, `calendar` e `schedule`. Per accedere all'oggetto CORBA `calendar` il client dovrà prima chiedere al naming server di accedere al naming context *Personal* e poi all'oggetto di nome `calendar`.

La creazione di oggetti CORBA in Java

Per creare un oggetto CORBA occorre definire quali sono le loro interfacce. Per fare questo si utilizza il linguaggio IDL, **I**nterface **D**efinition **L**anguage. Con un semplice tool messo a disposizione dalla Sun (`idltojava`) le definizioni delle interfacce IDL vengono tradotte nelle corrispondenti espresse in linguaggio Java, assieme ad una serie di classi che permetteranno l'implementazione dell'oggetto CORBA desiderato in modo semplice.
In Figura D.3 è mostrata la dichiarazione IDL di un oggetto CORBA **Wrapper** e la corrispondente traduzione in Java.
Definendo poi una classe Java che implementa l'interfaccia creata si può creare

```
// definizione dell'interfaccia IDL
module MomisApplic {
  interface Wrapper {
     string getType() raises (momisOqlException);
     string getDescription() raises (momisOqlException);
     MomisResultSet runQuery( in string oql ) raises (momisOqlException);
     string getSourceName() raises (momisOqlException);
  };
}

// la stessa interface tradotta in Java
package MomisApplic;
    public interface Wrapper
        extends org.omg.CORBA.Object,
                org.omg.CORBA.portable.IDLEntity {

        String getType()
            throws MomisApplic.momisOqlException;
        String getDescription()
            throws MomisApplic.momisOqlException;
        MomisApplic.MomisResultSet runQuery(String oql)
            throws MomisApplic.momisOqlException;
        String getSourceName()
            throws MomisApplic.momisOqlException;
}
```

Figura D.3: Traduzione in Java di una interfaccia IDL di un oggetto CORBA

l'oggetto CORBA: naturalmente, occorrerà scrivere il codice dei metodi dichiarati nell'interfaccia. Occorre sottolineare che gli oggetti CORBA non hanno proprietà *pubbliche* ma solo *private* ed accessibili solo tramite i metodi messi a disposizione dall'interfaccia.

Appendice E

Le classi java JoinMap e JoinTable

Come già accennato nel Capitolo 5, il software prodotto durante la realizzazione di questa tesi è disponibile nella directory */export/home/progetti.comuni/tesi/micol/sw* del server *"Sparc20"* presso il Dipartimento di Scienze dell'Ingegneria.
A titolo esemplificativo viene riportato di seguito il codice relativo a due classi java particolarmente significative tra quelle implementate: la classe *JoinMap*, che modella la struttura dati Join Map, e la classe *JoinTable*, che modella la struttura dati Join Table.

E.1 JoinMap.java

```
package globalschema;

import java.util.Vector;
import java.io.Serializable;

  /**
   * questa classe implementa la Mappa dei Join
   * di una classe globale
   */

  public class JoinMap extends odli3.MomisObject implements Serializable
  {
  //
  // PROPERTIES
  //

  /**
   * prima classe coinvolta
   */

  public SourceClass firstElement;

  /**
   * seconda classe coinvolta
   */
```

```
public SourceClass secondElement;

/**terza classe coinvolta. Viene usata
 * nel caso in cui non sia possibile
 * effettuare il join diretto tra le due classi
 */

public SourceClass thirdElement;

/**indica la presenza di informazioni
 * sulla stessa entita'
 * del mondo reale e quindi l'eventualita' del join
 */

public boolean mustJoin;


/**
 * indica se  possibile effettuare il join diretto
 * tra le due SourceClass
 */

public boolean joineable;


/**attributi da utilizzare per il join
 * e' un oggetto JoinAttribute
 */

public JoinAttribute joinAttribute;

//
// CONSTRUCTORS
//

/** genera un'istanza della classe
 *  inizializzandone i campi
 *  @param ele1 e' la prima classe locale
 *  @param ele2 e' la seconda classe locale
 */

public JoinMap(SourceClass ele1, SourceClass ele2)
  {
      firstElement = ele1;
      secondElement = ele2;
      thirdElement = null;
      mustJoin = false;
      joineable = false;
      joinAttribute=null;

   }
public JoinMap(SourceClass ele1, SourceClass ele2, boolean joinYN)
  {
   firstElement = ele1;
   secondElement = ele2;
   thirdElement = null;
   mustJoin = false;
   joinAttribute=null;
   joineable = joinYN;

  }
```

```
  //
  // METHODS
  //

  /** Associa alla propertie joinable il
   *  valore passato per parametro
   *  @param joinYN true se e' possibile il join diretto,
   *   altrimenti false
   */

  public void setJoineable(boolean joinYN)
  {
     joineable = joinYN;
  }

  /**
   * Associa alla propertie mustJoin il valore
   * passato per parametro
   * @param mustYN true se ci sono entita' in comune,
   * altrimenti false
   */

  public void setMustJoin(boolean mustYN)
  {
     mustJoin = mustYN;
  }

  /** Associa alla propertie thirdElement la
   *  SourceClass corrispondente
   *  @param tc Un oggetto SourceClass che
   *  rappresenta la classe intermedia
   *  di join
   */

  public void setThird(SourceClass tc)
  {
  thirdElement=tc;
  }

  /**Associa alla propertie joinAttribute
   * l'oggetto JoinAttribute
   * corrispondente
   * @param ja L'oggetto JoinAttribute coinvolto
   */

  public void setJoinA(JoinAttribute ja)
  {
  joinAttribute=ja;
  }

  /**Restituisce l'oggetto JoinAttribute della
   * JoinMap in questione
   */

  public JoinAttribute getJoinA()
  {
  return joinAttribute;
  }

}
```

E.2 JoinTable.java

```
package globalschema;

import java.util.Vector;
import java.io.Serializable;


  /** Costituisce la descrizione di una Join Table.
   *  E' un set di JoinMap, ovvero rappresenta l'insieme
   *  delle mappe di join di
   *  un'intera classe globale.
   */

  public class JoinTable extends odli3.MomisObject implements Serializable
  {
  //
  // PROPERTIES
  //

  /** indica il nome della classe globale cui
   *  fa riferimento
   */

  public String gClassName;

  /** indica il nome dello schema globale cui
   *  fa riferimento
   */

  public String globalSchema;

  /** questo campo riporta una collezione di
   *  tipo Vector contenente
   *  l'insieme delle JoinMap costituenti la Join Table
   */

  public Vector setJoinMap;

  //
  // CONSTRUCTORS
  //

  /** Genera un'istanza della classe inizializzandone i campi
   *  @param className Nome della classe globale cui
   *  fa riferimento
   *  la JoinTable
   */

  public JoinTable(String className)
  {
   gClassName = className;
   globalSchema = "";
   setJoinMap = new Vector(0);
  }


  //
  // METHODS
  //
```

```
/** Aggiunge una nuova JoinMap alla JoinTable
 *  @param JM Oggetto JoinMap
 */

public void addJoinMap(JoinMap JM) throws Exception
{
   if (setJoinMap.contains(JM)) throw new Exception ("Error");
       setJoinMap.add(JM);
}

/** Fornisce l'oggetto di tipo JoinMap
 *  riguardante le due classi passate
 *  come parametri
 *  @param cl1 prima classe coinvolta
 *  @param cl2 seconda classe coinvolta
 *  @return retJM oggetto di tipo JoinMap restituito
 */

public JoinMap getJoinMap(SourceClass cl1,SourceClass cl2)
{
 int i;
 JoinMap JM,retJM = null;

 for(i = 0;i < this.setJoinMap.size();i++)
 {
  JM = (JoinMap)this.setJoinMap.get(i);
  if((JM.firstElement == cl1)&&(JM.secondElement == cl2))
          {retJM = JM;
           break;
          }
  else if((JM.firstElement == cl2)&&(JM.secondElement == cl1))
         {retJM = JM;
          break;
          }
 }
 return retJM;
}

/** Associa alla propertie setJoinMap il
 *  Vector of JoinMap passato come
 *  parametro
 */

public void setJoinTable(Vector jt)
{
  for(int j=0;j<jt.size();j++){
    JoinMap jm=(JoinMap)jt.get(j);
    try{addJoinMap(jm);}
    catch (Exception e){System.out.println("Error");}
}
 }

/**Rimuove da setJoinMap la JoinMap specificata
 */

public void removeJoinMap(JoinMap jm)
{  boolean b;
    b=setJoinMap.remove(jm);
}

/**Svuota setJoinMap
 */
```

```
  public void removeAllJM()
  {

    setJoinMap.removeAllElements();
  }
}
```

Bibliografia

[1] A. Zaccaria. MOMIS: Il componente Query Manager. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[2] A. Rabitti. Architettura di un Mediatore per un Sistema di Sorgenti Distribuite ed Autonome. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[3] S. Montanari. Un approccio intelligente all'Integrazione di Sorgenti Eterogenee di Informazione. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1997-1998.

[4] G. P. Grifa. Analisi di Affinità Strutturali fra classi ODL_{I^3} nel Sistema MOMIS. Tesi di Laurea, Università di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1998-1999.

[5] A. 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, corso di laurea in Ingegneria Informatica, 1997-1998.

[6] M. Vincini. ODB-QOptimizer: un ottimizzatore semantico di interrogazioni. Tesi di Laurea, Univeristà di Modena, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1994.

[7] M. Franceschi. Il componente Query Manager di MOMIS: utilizzo della Conoscenza Estensionale. Tesi di Laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999-2000.

[8] S. Bergamaschi, S. Castano, S. De Capitani di Vimercati, S. Montanari, and M. Vincini. An Intelligent Approach to Information Integration. In *Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS'98)*, Trento, Italy, june 1998.

[9] S. Bergamaschi, S. Castano, S. De Capitani di Vimercati, S. Montanari, and M. Vincini. Exploiting schema knowledge for the integration of heterogeneous sources. In *Sesto Convegno Nazionale su Sistemi Evoluti per Basi di Dati - SEBD98, Ancona*, pages 103–122, 1998.

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

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

[12] Gio Wiederhold. Mediators in the architecture of Future Information Systems. *IEEE Computer*, 25:38–49, 1992.

[13] 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.

[14] 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.

[15] Y.Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object Fusion in Mediator Systems. In *VLDB Int. Conf.*, Bombay, India, September 1996.

[16] N.Guarino. Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration. Technical report, Summer School on Information Extraction, Frascati, Italy, July 1997.

[17] N.Guarino. Understanding, Building, and Using Ontologies. A commentary to 'Using Explicit Ontologies in KBS Development', by van Heijst, Schreiber, and Wielinga.

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

[19] 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.

[20] Oliver M.Duschka and Micheal R.Genesereth. Infomaster - An Information Integration Toolkit. Technical report, Department of Computer Science. Stanford University, 1996.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Y. Arens, C. A. Knoblock, and C. Hsu. Query Processing in the SIMS Information Mediator. *Advanced Planning Technology*, 1996.

[27] Silvia Zanni. Il componente Query Manager di MOMIS: esecuzione di interrogazioni. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999-2000.

[28] R. Hull. Managing Semantic Heterogeneity in Databases: A Theoretical Perspective. Technical report, Bell Laboratories, 1996.

[29] AA. VV. The common object request broker: Architecture and specification. Technical report, Object Request Broker Task Force, 1993. Revision 1.2, Draft 29, December.

[30] 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.

[31] 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.

[32] D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. A Description Logics Based Tool for Schema Validation and Semantic Query Optimization in Object Oriented Databases. Technical report, sesto convegno AIIA, 1997.

[33] A.G. Miller. WordNet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[34] 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.

[35] 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.

[36] I. Schmitt and G. Saake. Merging Inheritance Hierarchies for Database Integration. *IEEE Trans. on Knowledge and Data Engineering*.

[37] S. Castano and V. De Antonellis. Semantic Dictionary Design for Database Interoperability. In *Proc. of Int. Conf. on Data Engineering, ICDE'97*, Birmingham, UK, 1997.

[38] 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.

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

[40] Francesco Venuta. Trattamento della conoscenza estensionale nel sistema MOMIS. Tesi di laurea, Università di Modena e Reggio Emilia, Facoltà di Ingegneria, corso di laurea in Ingegneria Informatica, 1999-2000.

[41] Serge Abiteboul Y.Papakonstantinou, H. Garcia-Molina. Object Fusion in Mediator Systems. In *VLDB Int. Conf.*, 1996.

[42] Aron Rosenthal Edward Sciore, Michael Siegel. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM Transactions on Database System*, 19(2):254–290, June 1994.

[43] Ken Smith and Leo Orbst. Unpacking The Semantics of Source and Usage To Perform Semantic Reconciliation In Large-Scale Information Systems. *SIGMOD Records*, 28(1):26–31, 1999.