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

ODB-DSQO un server WWW per la validazione di schemi di dati ad oggetti e l'ottimizzazione di interrogazioni conforme allo standard ODMG-93

Tesi di Laurea di
Alberto Corni

Relatore
Chiar.ma Prof.ssa Sonia Bergamaschi

Correlatore
Dott. Ing. Maurizio Vincini

Anno Accademico 1995 - 96

Parole chiave:
Server WWW
Basi di dati ad oggetti
ODMG-93
Traduttori
Applet Java

Ai miei genitori

Indice

Capitolo 1

Introduzione

Premessa

Il lavoro della presente tesi è stato svolto all'interno del gruppo di ricerca sull'applicazione di tecniche di intelligenza artificiale alle basi di dati ad oggetti, presso il dipartimento di scienze dell'Ingegneria (DSI) dell'Università di Modena.

Tale gruppo ha in corso di sviluppo il progetto ODB-Tools .

Gli argomenti trattati riguardano basi di dati, modelli e linguaggi di programmazione ad oggetti ed anche l'utilizzo dei Browser WWW per la navigazione in Internet. Si presuppone che il lettore abbia una discreta preparazione su questi argomenti.

Scopo della tesi

Realizzare alcuni moduli del progetto ODB-Tools . In particolare:

1. Un traduttore che accetti come input la descrizione di uno schema di database ad oggetti descritto secondo il linguaggio descrittivo ODL dello *standard* ODMG-93 e che traduca tale schema nel linguaggio OCDL del progetto ODB-Tools al fine di poter utilizzare l'ottimizzatore di interrogazioni.

2. Interfaccia verso Internet di componenti prodotte nel progetto ODB-Tools : OCDL-designer per la validazione degli schemi e ODQOptimizer per l'ottimizzazione delle interrogazioni.

Contenuto della tesi

Questo documento contiene: una visione di insieme del progetto ODB-Tools , un riassunto dello *standard* ODMG-93 e la descrizione dei problemi affrontati e risolti durante la realizzazione del software.

Nei capitoli 2 e 3 si trovano le informazioni relative ai due modelli di basi di dati oggetto della tesi. Nel primo è riassunto il lavoro del gruppo ODMG. Nel secondo è riassunto il lavoro svolto dal gruppo di ricerca del DSI. In entrambi

i casi ho scelto di soffermarmi sui punti che riguardano più da vicino il lavoro operativo. Nel capitolo 4 si trova un breve confronto tra i due modelli.

Progetto e realizzazione del traduttore

Dati due linguaggi diversi per rappresentare lo schema di un database ad oggetti, occorreva trovare un metodo per tradurre le informazioni *importanti* da un sistema ad un altro.

In particolare, i linguaggi dello standard **ODMG-93 (Object Database Management Group)** sono ampi, orientati alla realizzazione di ODBMS funzionali e descrivono tutti gli aspetti che riguardano la gestione di un database ad oggetti, dalla definizione dei tipi oggetti alla gestione delle transazioni per il recupero dei dati in caso di errore. Questa proposta di standard, tenta di essere il più concreta possibile e cerca di conformarsi il più possibile agli standard esistenti, ad esempio la dichiarazione delle strutture dati avviene come in C, e prevede l'interfacciamento a linguaggi di programmazione ad oggetti quali il C++ e Smalltalk. Così **ODL** (Object Definition Language) dell'ODMG è stato sviluppato secondo questa filosofia e quindi non è solo un rigoroso linguaggio di descrizione dei tipi di oggetti contenuti in un database ma eredita dagli standard da cui deriva tutte quelle funzionalità che sono state giudicate utili e comode.

D'altro canto **OCDL** (Object and Costraint Definition Language) definito nel progetto ODB-Tools è nato per rendere più agevole il fornire dati all'algoritmo di ottimizzazione di uno schema. Così OCDL fonrisce potenti primitive per la gestione delle strutture dati dei sistemi ad oggetti ma è incapace di gestire molte delle funzionalità previste in ODL.

Viceversa alcune importanti caratteristiche a base a OCDL come le *regole di integrità* e le *viste* non esistono in ODL.

Vedremo entrambi i sistemi ed anche quali restrizioni ed estensioni sono state apportate all'ODL per poter tradurre efficacemente uno schema ODL in OCDL.

Progetto e realizzazione dell'interfaccia

Si è optato per realizzare un server **http** per rendere disponibile su Internet tutto il materiale disponibile sul progetto (documentazione e prototipo funzionante). Inoltre si voleva che chiunque, connesso ad Internet, potesse provare il prototipo sia su esempi forniti dal server, ma anche, su schemi forniti direttamente dall'utilizzatore. Occorreva dunque rendere interattivo il sito, fare in modo che la macchina su cui è installato il server HTTP potesse ricevere informazioni dal *cliente*, elaborarle e rispedire i risultati al cliente. Tutto ciò è stato risolto con l'uso delle funzionalità CGI (Common Gateway Interface).

La considerazione dell'efficacia di rappresentazione grafica delle informazioni ha portato alla scelta progettuale di realizzare un interfaccia grafica verso Internet. In particolare di è scelto di inserire un *applet Java* che riceve informazioni relative

alla struttura di uno schema e le visualizza sotto forma di grafo.
Attualmente è possibile interagire con il server WWW oggetto della presente tesi all'indirizzo Internet:
`http://sparc20.dsi.unimo.it`

Capitolo 2

il lavoro del gruppo ODMG-93

Importante : Il gruppo ODMG-93 (Object Database Management Group) non ha ancora pubblicato un vero e proprio standard, bensì un documento bozza che rende pubblico il lavoro svolto fino ad un certo istante. Il documento a cui si fà riferimento in questa tesi ha come data di scadenza *gennaio 1997* data in cui probabilmente verrà pubblicata una versione aggiornata.

Questa sezione è una sintesi dallo standard ODMG-93 [**?**].

2.1 Introduzione

ODMG (Object Database Mangement Group) è nato per arrivare ad una formulazione *veloce* di uno standard innovativo nel campo delle basi di dati ad oggetti. Un punto di partenza su cui possano lavorare gli organismi internazionali di standardizzazione.

Gli standard nel campo delle basi di dati sono importanti. Si stima che la maggior limitazione all'impiego massiccio dei DataBase ad oggetti è la carenza di standard.

L'obbiettivo è creare un insieme di standard che permettano agli utenti di ODBMS[1] di creare applicazioni portabili, applicazioni che possano girare cioè su ODBMS diversi. Per fare ciò occorre rendere portabile lo schema, l'interfaccia con i linguaggi di programmazione, i linguaggi di *manipolazione* ed interrogazione.

Definizione (ODBMS):

un ODBMS è un DBMS che integra in modo *trasparente* le funzionalità di un database con le funzionalità di linguaggi orientati agli oggetti.

[1] ODBMS: Object DataBase Management Systems

Un ODBMS permette di rendere oggetti persistenti e ne permette la condivisione tra programmi diversi. Inoltre mette a disposizione altre caratteristiche come il controllo di processi concorrenti, data recovery, query associative, ecc..

L'SQL è stato usato come base per il linguaggio di interrogazione. Il risultato è **OQL(Object Query Language)**, un linguaggio che non è più completamente compatibile con l'SQL standard ma è stato esteso per sfruttare i vantaggi degli oggetti.

I componenti principali dello standard ODMG-93 sono:

- il modello dei dati e degli oggetti ODM (sezione 2.2);
- Object Definition Language ODL (sezione 2.3);
- l'Object Query Language OQL. Nello standard sono descritte la sintassi e le caratteristiche del linguaggio di interrogazione proposto per le basi di dati ad oggetti;
- l'interfacciamento con C++ e Smalltalk. Sono descritte le API (Application Program Interface), strutture di dati, convenzioni che permettono l'interfacciamento di un programma in C++ o Smalltalk all'ODBMS;

2.2 Il modello dei dati ODM

Il modello ODM (Object Data Model) è orientato a fornire un valido strumento per la realizzazione di database ad oggetti. Per questo motivo affronta tutti i problemi ad esso associati.

2.2.1 Concetti generali

ODM specifica il tipo di semantica che può essere definita esplicitamente in un database ad oggetti.

I costrutti definiti sono i seguenti:

- le primitiva di base di modellazione sono gli *oggetti* ed i *letterali*. Ogni oggetto ha un identificatore unico. Un letterale non ha identificatore;
- Lo *stato* di un oggetto è definito dal valore che questo *assume* per un insieme di *proprietà*. Queste proprietà possono essere *attributi* o *relazioni* tra l'oggetto ed uno o più oggetti. Normalmente il valore di una proprietà cambia nel tempo;

- Il *comportamento* (behavior) di un oggetto è definito da un'insieme di *operazioni* che possono essere eseguite sull'oggetto;
- Oggetti e letterali sono suddivisi per *tipo*. Tutti gli elementi di un dato tipo hanno in comune lo stesso insieme degli stati possibili e comportamento. A volte ci si riferisce ad un oggetto come *istanza* del suo tipo;
- Un *database* memorizza oggetti e permette la condivisione di questi tra più utenti ed applicazioni. Un database è basato su uno *schema* che è definito utilizzando il linguaggio ODL (Object Definition Language) e contiene istanze dei tipi definiti dal suo schema;

Il modello degli oggetti ODM specifica cosa si intende per oggetti, letterali, tipi, operazioni, proprietà, attributi, relazioni e così via. Questi costrutti permettono a chi sviluppa applicazioni di descrivere lo schema logico del database, cioè di descrivere i costrutti utilizzabili da chi usa il database.

2.2.2 Tipi e classi: Interfaccia ed implementazione

In accordo con la modellazione Object Orientend, ci sono due aspetti nella definizione di un tipo. Per ogni tipo è specificata un *interfaccia* ed una o più *implementazioni*. L'interfaccia definisce le caratteristiche esterne di un tipo. Le caratteristiche esterne sono quelle visibili agli utilizzatori degli oggetti. L'implementazione definisce l'aspetto interno dei tipi oggetto.

L'implementazione di un tipo consiste in una *rappresentazione* e in un insieme di *metodi*. La rappresentazione è la struttura dei dati. I metodi sono procedure. C'è un metodo per ogni operazione definita nell'interfaccia. Possono esistere metodi che non hanno una controparte diretta nell'interfaccia. La realizzazione interna non è visibile agli utilizzatori degli oggetti.

La separazione tra interfaccia e implementazione riflette l'*incapsulazione*. L'ODL (Object Description Language) è il linguaggio utilizzato per specificare l'interfaccia dei tipi usati nelle applicazioni.

Sottotipi ed ereditarietà Anche il modello dei dati ODM include la relazione di tipo-sottotipo basata sul concetto di ereditarietà. Dal punto di vista grafico, questa relazione coinvolge due nodi, uno detto *supertipo* l'altro *sottotipo*. Il supertipo è il tipo più generale, il sottotipo è quello più specializzato. Il sottotipo porta tutte le informazioni del relativo supertipo, così pure per il comportamento, il sottotipo supporta tutte le operazioni dichiarate nell'interfaccia del supertipo. Un'istanza di sottotipo può essere usata come un'istanza del supertipo in quanto il sottotipo eredita dal supertipo.

Nell'interfaccia del sottotipo possono essere definite caratteristiche aggiuntive a quelle definite nel supertipo. L'interfaccia del sottotipo può anche essere ridefinita per specializzare lo stato ed il comportamento del sottotipo stesso.
È supportata l'ereditarietà multipla. Può capitare che un tipo erediti da due differenti supertipi caratteristiche con il medesimo nome ma diversa semantica, il modello NON specifica come risolvere questo problema (è dipendente dal linguaggio scelto per l'implementazione).

Estensioni L'*estensione* di un tipo è l'insieme di tutte le istanze del tipo all'interno di un dato database. Se il tipo **A** è sottotipo di **B** allora l'estensione di **A** è sottoinsieme dell'estensione di **B**.
Nelle basi di dati ad oggetti è possibile decidere quali sono i tipi di cui l'ODBMS deve mantenere l'estensione. Mantenere l'estensione significa tener aggiornato l'insieme *estensione* ogni qual volta viene creato un nuovo oggetto o ne viene cancellato uno. Sulle estensioni possono essere definiti degli indici, che vanno mantenuti coerenti.
Alcuni tipi possono essere istanziati direttamente e sono detti *tipi concreti*. Altri non possono essere istanziati direttamente e sono detti *tipi astratti*.

Chiavi (Keys) In alcuni casi le singole istanze di un tipo possono essere identificate univocamente dal valore di alcune *proprietà* o da insiemi di queste. Queste proprietà sono dette chiave. Una *chiave semplice* (*simple key*) è costituita da una sola proprietà. Una *chiave composta* (*compound key*) consiste in un insieme di proprietà. L'unicità di una chiave è definita rispetto all'estensione di un tipo, così un tipo deve avere un'estensione per avere una chiave.

2.2.3 Gli oggetti

In ODMG vengono rappresentati i seguenti aspetti degli oggetti:

- *Identificatori*. Usati da un ODBMS per distinguere un oggetto da un altro e per trovare gli oggetti;
- *Nomi*. Scelti dal programmatore o dall'utente finale come maniera conveniente per riferirsi ad un particolare oggetto;
- *Creazione*. Determina il modo in cui gli oggetti sono creati dal programmatore;
- *Tempo di vita*. Determina il modo in cui dev'essere gestita la memoria allocata ad un oggetto;

- *Struttura*. Può essere atomica oppure no, se non è atomica, l'oggetto contiene altri oggetti.

Identificatore di oggetto Poiché tutti gli oggetti hanno un identificatore, un oggetto può sempre essere distinto dagli altri all'interno del suo *dominio di memorizzazione*. In questa versione dell'ODMG Object Model, un dominio di memorizzazione è un database. L'unicità di un oggetto è assicurata grazie all'*object identifier* (OID). Un oggetto mantiene il proprio *object identifier* per tutta la propria *vita*. Il valore degli *object identifier* non cambia, mentre lo stato, cioè l'insieme dei valori dei suoi attributi, cambia.

I letterali non hanno un proprio identificatore, esistono negli oggetti e non possono essere referenziati individualmente. I valori letterali sono spesso descritti come delle costanti.

Nome di oggetto Ad un oggetto viene assegnato un identificatore dall'ODBMS ma possono essergli assegnati anche dei nomi che siano più significativi per il programmatore e l'utente finale. L'ODBMS fornisce una funzione che mappa i nomi in *object identifier*. ODMG indica che i nomi usati nelle applicazioni si riferiscano ad oggetti *radice*, questi forniscono il punto di accesso al database.

I nomi degli oggetti sono come le variabili globali dei linguaggi di programmazione e sono ben diversi dalle chiavi.

Il dominio di unicità di un nome è il database. Il modello degli oggetti ODM non indica nessuna nozione sulla gerarchia dello spazio dei nomi di un database e non prevede norme sulla condivisione di nomi tra database diversi.

Creazione di oggetti Gli oggetti sono creati invocando l'operazione di creazione definite nell'interfaccia *factory* dell'oggetto *factory*. Ogni interfaccia "fabbrica" crea oggetti di uno o più tipi. L'operazione `new` causa la creazione di un oggetto di un dato tipo.

Tempo di vita degli oggetti Cioè come dev'essere gestita la memoria e lo spazio di memorizzazione allocato all'oggetto gestito. Il tempo di vita dell'oggetto dev'essere specificato quando l'oggetto è creato.

Sono supportati due tipi di tempi di vita:

- transiente
- persistente

Un oggetto definito come *transiente* è memorizzato in memoria centrale e la memoria è gestita dal sistema run-time associato al linguaggio di programmazione. Un esempio sono gli oggetti allocati da una procedura, questi vengono creati dalla procedura e distrutti quando la procedura termina.

Un oggetto definito come *persistente* è memorizzato in memoria secondaria e la sua memorizzazione è gestita dal sistema run-time del ODBMS. Questi oggetti continuano ad esistere anche dopo la fine della procedura che li ha creati. Gli oggetti *persistenti* sono anche detti *oggetti del database*.

Il tempo di vita è indipendente dal tipo. Lo stesso tipo può avere oggetti persistenti e oggetti transienti. Questo fatto (che è innovativo rispetto ai database relazionali) permette di manipolare oggetti persistenti e non persistenti, usando le medesime operazioni.

Oggetti Atomici Un *tipo* di oggetto atomico è definito dall'utente. Non esistono oggetti atomici definiti nel ODBMS. Nel modello degli oggetti ODM i tipi di oggetto che non sono atomici sono collezioni.

Oggetti Collezione Le istanze dei tipi collezione sono composte da diversi elementi ogni uno dei quali può essere un'istanza di un tipo atomico, di una collezione o di un letterale. Tutti gli elementi di una collezione devono essere dello stesso tipo. Tipi collezione supportati dal modello degli oggetti ODM :

- **set** <**t**> collezione non ordinata di elementi in cui non sono permessi duplicati
- **bag** <**t**> collezione non ordinata di elementi che può contenere duplicati
- **list** <**t**> collezione ordinata di elementi
- **array** <**t**> collezione ordinata di un numero fissato di elementi che possono essere referenziati per posizione
- **dictionary** <**t,v**> è una sequenza disordinata di coppie chiave-valore ove non sono permesse chiavi duplicate.

t indica il tipo degli elementi della collezione.

Un meccanismo per accedere agli elementi della collezione è fornito dagli *iteratori* . Un iteratore può essere *stabile* o meno. Un iteratore *stabile* assicura coerenza di elaborazione di una collezione anche se questa viene modificata durante l'iterazione.

2.2.4 Letterali

I letterali NON hanno identificatore. Il modello ad oggetti supporta tre tipi di letterali:

- letterali atomici
- letterali collezione
- letterali strutturati

Letterali Atomici Esempi di Letterali Atomici sono i numeri ed i caratteri. Istanze di questi tipi non sono esplicitamente create dall'applicazione ma, semplicemente, esistono. Il modello degli oggetti ODM supporta i seguenti tipi di letterali atomici:

- long
- short
- unsigned long
- unsigned short
- float
- double
- boolean
- char (carattere)
- string
- enum (enumerazione)

Se il linguaggio di programmazione scelto per l'implementazione non supporta uno di questi tipi, allora, le librerie di interfaccia con il ODBMS dovranno contenere l'implementazione del tipo mancante.
Il costrutto `enum` è un *generatore di tipo*.

Letterali Collezione Il modello degli oggetti ODM supporta i seguenti tipi di letterali collezione:

- set<t>
- bag<t>
- list<t>
- array<t>
- dictionary<t>

Questi tipi sono analoghi agli oggetti collezione ma non hanno l'identificatore. Tuttavia i loro elementi possono essere letterali o oggetti.

Letterali Strutturati I letterali strutturati (*structure*), sono composti da un numero fissato di elementi ad ogni uno dei quali è associato un nome e può essere un letterale o un oggetto.
Il modello degli oggetti ODM supporta almeno i seguenti tipi di letterali strutturati:

- **date**
- **interval** rappresenta una durata di tempo, è usato per alcune operazioni su **Time** e **Timestamp**
- **time** Identifica un tempo a livello mondiale, tiene conto del meridiano in cui ci si trova.
- **timestamp** consiste nell'incapsulazione **date** e **time**.

Strutture definite dall'utente Poiché il modello degli oggetti ODM è estensibile gli sviluppatori possono definirsi le strutture di cui hanno bisogno. I costrutti *struct* e *typedef* permettono di generare nuove strutture.

2.2.5 Modellazione dello stato - proprietà

Ogni oggetto definisce un insieme di proprietà attraverso le quali l'utente può accedere e manipolare direttamente lo stato di un'istanza di un tipo oggetto. Vi sono due tipi di proprietà definite nel modello degli oggetti ODM : *attributi* e *relazioni*. L'attributo è relativo ad certo tipo-oggetto. La relazione è definita tra due tipi-oggetto, i due tipi devono avere istanze referenziabili da *object identifier*.

Attributi La dichiarazione di attributi in un'interfaccia definisce lo stato astratto di un tipo. Il valore di un attributo può essere un letterale oppure un identificatore di un oggetto. Gli attributi NON sono delle strutture. Possono essere implementati con *metodi*. Nell'attuale versione del modello degli oggetti ODM gli attributi NON sono considerati oggetti.

Relazioni Sono definite tra tipi. Il modello degli oggetti ODM supporta solo relazioni *binarie*, ad esempio relazioni tra due tipi. Non supporta relazioni n-arie. La relazione binaria puo essere uno-a-uno, uno-a-molti o molti-a-molti. Le relazioni nel modello degli oggetti ODM sono simili alle relazioni del modello entity-relationship (ma binarie).

Una relazione NON è un oggetto. Una relazione è definita dichiarando il *percorso trasversale* che abilita le applicazioni ad usare le connessioni logiche tra gli oggetti che partecipano alla relazione. All'interno della definizione di un tipo si attribuisce un nome alla proprietà relazione, il percorso trasversale è il nome che indica il nome della proprietà controparte della relazione nel tipo legato. I percorsi trasversali sono dichiarati a coppie, uno per ogni direzione della relazione binaria. Esempio:

```
interface Professor {
   ...
   relationship set<Course> teaches
      inverse Course::is_taught_by;
   ...
}

interface Course {
   ...
   relationship Professor is_taught_by
      inverse Professor::teaches
   ...
}
```

La cardinalità di una relazione è data nella dichiarazione dei percorsi trasversali, dipende dal fatto che si usi una *collezione* oppure un singolo identificatore di oggetto. Analogamente la relazione può essere ordinata (se si usa una *collezione* ordinata) o non ordinata.

L'ODBMS è responsabile di mantenere l'integrità referenziale delle relazioni. Il metodo consigliato per mantenere tale integrità consiste nel gestire le relazio-

ni tramite le operazioni *form*, creazione di una istanza della relazione, e *drop*, cancellazione di un'istanza della relazione.

Un percorso trasversale è più di un puntatore di un linguaggio di programmazione ad oggetti.

2.2.6 Modellazione del comportamento - operazioni

Accanto [2] ad attributi e relazioni, l'altra caratteristica di un tipo è il suo comportamento, questo è specificato come insieme di *"prototipi" di operazioni*. Ogni "prototipo" definisce il nome dell'operazione, il nome ed il tipo di ogni argomento, il tipo del valore ritornato ed il nome di ogni *eccezione* (condizione di errore) ritornata. La specifica del modello degli oggetti ODM è identica a quella dell' OMG CORBA [**?**].

Il nome di un'operazione deve essere unico all'interno di un tipo. Più tipi possono avere operazioni diverse con il medesimo nome. Si dice che il nome di queste operazioni è *overloaded*. Quando viene invocata una operazione il cui nome è *overloaded* occorre *scegliere* l'operazione adeguata. L'azione di scelta dell'operazione adeguata è detta *operation name resolution* oppure *operation dispatching*, viene scelta l'operazione definita sul tipo più specifico dell'oggetto fornito come primo argomento della chiamata.

Il modello degli oggetti ODM assume che le operazioni siano eseguite sequenzialmente. Non richiede supporto per operazioni parallele o concorrenti ma non preclude l'ODBMS dal trarre vantaggi dai supporti multiprocessore.

Modello delle eccezioni Le eccezioni sono oggetti speciali riservati alla gestione degli errori.

Le operazioni possono rilasciare eccezioni e le eccezioni possono contengono le informazioni relative all'errore generato durante l'operazione. Nel modello degli oggetti ODM le eccezioni sono oggetti ed hanno un'interfaccia che permette di mettersi in relazione con altre eccezioni in modo da ordinare le eccezioni in una gerarchia di generalizzazione-specializzazione. L'eccezione radice è fornita dall'ODBMS e può eseguire operazioni quali la terminazione del processo attuale.

La gestione delle eccezioni avviene tenendo conto delle transazioni attive, tutte le transazioni attive devono essere *abortite*.

[2] questo, è forse la parte che meno riguarda la mia tesi in quanto nel prototipo ODB-Tools non sono prese in considerazioni le operazioni. Tuttavia mi sembra bene avere una visione di insieme di un ODBMS.

2.2.7 Metadati

Un metadato è un'informazione descrittiva che riguarda gli oggetti che definiscono lo schema di un database. È usato dall'ODBMS per definire strutture di un database ed a run-time per guidarne gli accessi alla base di dati. Il metadato è memorizzato in un *ODL schema Repository*.

2.2.8 Modello delle Transazioni

I programmi che usano dati persistenti sono organizzati in transazioni. La gestione delle transazioni è una funzionalità fondamentale per un ODBMS per garantire l'integrità del database, condivisibilità e recupero dei dati in caso di errore.
La transazione è una unità logica per la quale un ODBMS garantisce *atomicità, consistenza, isolamento, durata, serializzabilità*.

2.2.9 Operazioni del Database

Un ODBMS può gestire uno o più database logici, ogni uno dei quali può essere memorizzato in uno o più database fisici. Ogni database logico è istanza del tipo Database che è fornito dall'ODBMS. Il tipo Database supporta le seguenti operazioni:

- *open* Il modello degli oggetti ODM richiede che sia aperto un solo database per volta
- *close* Questa operazione dev'essere invocata quando un programma ha completato tutti gli accessi ad un database.
- *bind* Serve per associare un nome ad un oggetto.
- *lookup* Dato il nome di un oggetto ne trova l'identificatore.

2.3 Il linguaggio ODMG-ODL

2.3.1 Introduzione

Il Linguaggio di Definizione degli Oggetti (ODL) è un linguaggio di specifica usato per definire le interfacce verso i tipi di oggetti in conformità con il modello degli oggetti ODM . L'obbiettivo principale dell'ODL è facilitare la portabilità di schemi di database tra ODBMS. Inoltre ODL è un passo verso l'interoperabilità di ODBMS creati da diversi produttori.

Principi che hanno guidato lo sviluppo dell'ODL.

- ODL deve supportare tutti i costrutti semantici del modello degli oggetti ODM
- ODL non deve essere un linguaggio di programmazione, bensì un linguaggio di specifica per il "prototipo" delle interfacce
- ODL dev'essere indipendente dal linguaggio di programmazione
- ODL dev'essere compatibile con il Linguaggio di Definizione delle Interfacce (IDL) dell'OMG
- ODL dev'essere estendibile, non solo per ulteriori funzionalità ma anche verso ottimizzazioni fisiche
- ODL dev'essere pratico, per gli sviluppatori di applicazioni, e dev'essere "supportabile" dai fornitori di ODBMS in un tempo relativamente breve.

Con riferimento ai database relazionali, si può pensare ad ODL come al DDL (Data Description Language) di un database ad oggetti. ODL Definisce le caratteristiche dei tipi oggetti, in particolare le proprietà e le operazioni. ODMG-93 non fornisce un OML (Object Manipulation Language) standard ma descrive le API (Application Program Interface) per l'utilizzo di C++ e Smalltalk.
La sintassi ODL estende IDL – il Linguaggio di Definizione delle Interfacce sviluppato dall'OMG come parte di CORBA (Common Object Request Broker Archiecture) [**?**]. IDL fù a sua volta influenzato dal C++, così ODL è simile al C++.
ODL vuol essere una sorta di *lingua franca*, secondo ODMG è possibile tradurre in ODL schemi di database che si adeguano ai seguenti standard: STEP/PDES (EXPRESS), ANSI X3H2 (SQL), ANSI XX3H7 (Object Information Management), CFI (CAD Framework Initiative) ed altri ancora.

2.3.2 Specifiche

Un tipo è definito specificando l'interfaccia in ODL. Nel libro dello standard a questo punto è descritta la sintassi dell'ODL per la descrizione di:

- caratteristiche dei tipi oggetto
- proprietà dei tipi (attributi e relazioni)
- operazioni

Ritengo sia inutile includere in questa tesi la sintassi standard dell'ODL per due motivi:

1. La sintassi standard è descritta molto bene nel libro dello standard;
2. Sul documento da cui è stata presa la sintassi ODL è scritto chiaramente che a Gennaio 1997 uscirà una nuova versione dello standard, per cui la sintassi che a mia disposizione è da ritenersi già obsoleta.

In appendice D è riportato un sottoinsieme della sintassi ODL, che è la sintassi riconosciuta dal *traduttore* sviluppato nella tesi. Questa sintassi deriva dalla sintassi ODL standard ma è stata Ristretta/Ampliata (vedere sezione 6.1.1 a pagina 56).

2.3.3 Un esempio in ODL

L'esempio illustra l'uso di ODL per definire lo schema di un database per la gestione di alcune funzioni di segreteria di un'università[3].
La rappresentazione grafica di questo schema è riportata in figura 6.2 a pagina 60.

```
interface Course ( extent courses keys name, number)
   {
   attribute string name;
   attribute string number;
   relationship list<Section> has_sections
      inverse Section::is_section_of
      {order_by Section::number};
   relationship set<Course> has_prerequisites
      inverse Course::is_prerequisite_for;
   relationship set<Course> is_prerequisite_for
      inverse Course::has_prerequisites;
   boolean offer (in unsigned short semester) raises (already_offered);
   boolean drop (in unsigned short semester) raises (not_offered);
   };
interface Section ( extent sections key (is_section_of, number))
   {
   attribute string number;
   relationship Professor is_taught_by
      inverse Professor::teaches;
   relationship TA has_TA
      inverse TA::assists;
```

[3]Questo è un esempio di ODL standard, è privo delle estensioni introdotte per portare in ODL caratteristiche tipiche di OCDL.

```
   relationship Course is_section_of
      inverse Course::has_sections;
   relationship set<Student> is_taken_by
      inverse Student::takes;
   };
interface Employee  ( extent employees key (name, id))
   {
   attribute string name;
   attribute short id;
   attribute unsigned short annual_salary;
   void fire () raises (no_such_employee);
   void hire ();
   };
interface Professor: Employee  ( extent professors)
   {
   attribute enum Rank { full, associate, assistant} rank;
   // attribute string rank;
   relationship set<Section> teaches inverse Section::is_taught_by;
   short grant_tenure () raises (ineligible_for_tenure);
   };
interface TA: Employee, Student()
   {
   relationship Section assists
      inverse Section::has_TA;
   };
interface Student ( extent students keys name, student_id)
   {
   attribute string name;
   attribute string student_id;
   attribute
      struct Address
         {
         string college;
         string room_number;
         } dorm_address;
   relationship set<Section> takes
      inverse Section::is_taken_by;
   boolean register_for_course
      (in unsigned short course, in unsigned short Section)
      raises
      (unsatisfied_prerequisites, section_full, course_full);
   void drop_course (in unsigned short Course)
      raises (not_registered_for_that_course);
```

```
void assign_major (in unsigned short Department);
short transfer(
   in unsigned short old_section,
   in unsigned short new_section
   )
   raises (section_full, not_registered_in_section);
};
```

Capitolo 3

Il progetto ODB-Tools

Il progetto ODB-Tools ha come obbiettivo lo sviluppo di strumenti per la progettazione assistita di basi di dati ad oggetti e l'ottimizzazione semantica di interrogazioni. Si basa su algoritmi che derivano da tecniche dell'intelligenza artificiale.
Il risultato della ricerca svolta nell'ambito di questo progetto è un prototipo che realizza l'ottimizzazione di schemi e l'ottimizzazione semantica delle interrogazioni.
In figura 3.1 sono rappresentati i vari moduli che compongono tale prototipo.
Nel presente capitolo viene presentato lo stato di ODB-Tools prima della tesi.

3.1 Il modello degli oggetti ODB-Tools

La seguente trattazione è una sintesi della teoria del progetto da cui mi sono riservato di trarre solo i concetti più importanti per la mia tesi. Rimando all' appendice A a pagina 117 per una presentazione più rigorosa dei risultati della teoria.
Si tratta di un modello per basi di dati orientate agli oggetti [**?**] che permette la modellazione di valori complessi e dell'ereditarietà multipla. Questo modello degli oggetti è fortemente orientato alla gestione delle strutture dati tralasciando altri aspetti dei modelli ad oggetti come il modo di definire le operazioni degli oggetti o il modo in cui più istanze di oggetti possono interagire tra loro.
Concetti principali:

- **Tipi atomici**.
- **Tipi Base**: tuple, sequenze e insiemi di tipi base.
- **Tipi Oggetto**, deriva dalla promozione di un tipo qualunque ad oggetto.

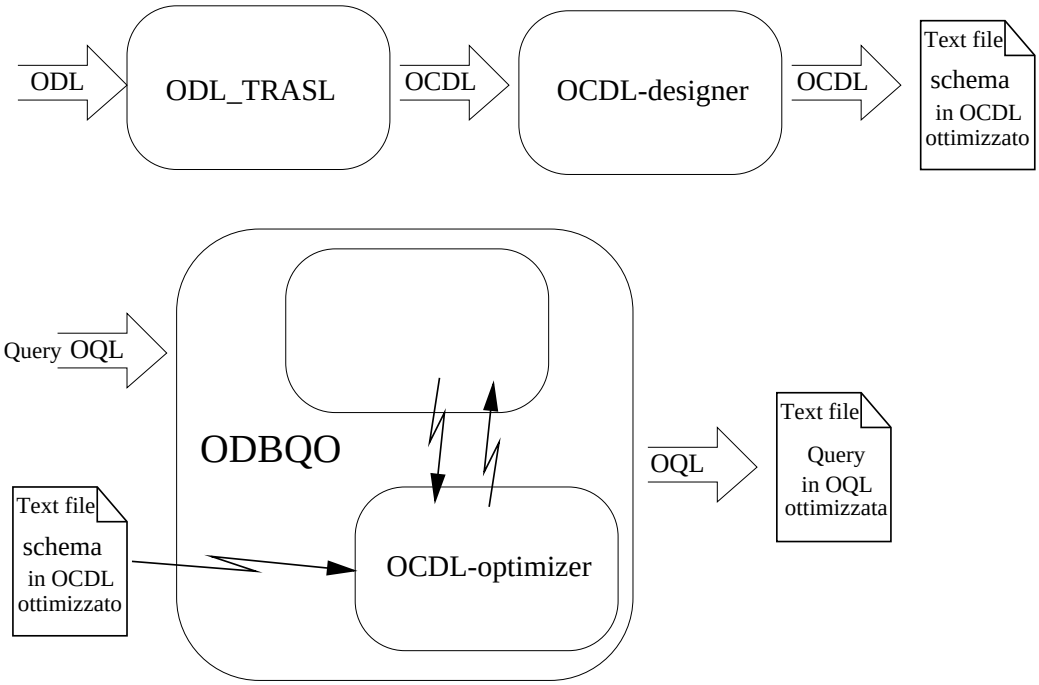


Figura 3.1: Componenti ODB-Tools

- **Ereditarietà** semplice e multipla espressa tramite l'operatore di intersezione.
- **Nomi di tipi** Si può assegnare nomi ai tipi. I vari tipi si dividono in:
  - tipo-valore
  - tipo-classe base
  - tipo-classe virtuale

3.1.1 Sistema di tipi atomici

I *tipi atomici*, detti anche *tipi base* sono:

- **integer**
- **string**
- **boolean**
- **real**

- **intervalli di interi**
- **tipi monovalore** questi corrispondono alle *costanti* dei tipi sopra elencati.

Una entità di tipo atomico può dunque essere ad esempio un *integer*, oppure il valore *5*, o il valore *"pippero"*, o una entità *boolean*, o un intervallo *4-154*, e così via.

3.1.2 Oggetti complessi, tipi e classi

Concetto di tipo Un *tipo* è un concetto ampio che permette di distinguere, classificare e definire le entità che fanno parte di una base di dati ad oggetti. Il tipo di una generica entità di una base di dati ad oggetti indica le caratteristiche di tale entità e non l'entità stessa. Una fondamentale distinzione esiste tra i *tipo oggetto* e i *tipo non oggetto* (*literal* in ODMG).

Concetto di oggetto Un oggetto è un'istanza di un particolare tipo detto *tipo oggetto* caratterizzata da un identificatore che identifica univocamente l'istanza e da un insieme di valori associati agli *attributi* dell'oggetto. I tipi oggetti sono istanziabili e ogni istanza porta il valore per gli attributi del tipo. I tipi non oggetto non sono istanziabili ma possono essere usati per creare tipi oggetto complessi o, messi in relazione con tipi oggetto, per porre delle costrizioni al dominio del valore degli attributi dei tipi oggetto.

Concetto di attributo Un attributo è un campo di una coppia: *label*, *tipo*, dove il tipo è fra quelli forniti nella definizione che segue.

Definizione di tipi La seguente regola sintattica definisce il sistema di tipi ammessi in OCDL.

$$
\begin{array}{lll}
S \rightarrow & B & tipo\ atomico \\
 & N & nome\ tipo \\
 & \{S\} & tipo\ insieme \\
 & < S > & tipo\ sequenza \\
 & \Delta S & tipo\ oggetto \\
 & [a_1 : S_1, ..., a_k : S_k] & tipo\ tupla \\
 & S_1 \sqcap S_2 & intersezione
\end{array}
$$

Tramite lo *schema* è possibile assegnare un nome ad un tipo. Tale nome dev'essere univoco.

La sintassi riportata significa che un **tipo** può essere:

- un semplice **tipo atomico**
- il tipo già associato ad un nome tipo
- un **insieme di tipi**
- una **sequenza di tipi** è un inisieme ordinato
- un **tipo oggetto**
- un **tipo tupla**
- un **tipo intersezione** questo tipo indica *ereditarietà* nel caso di intersezione tra tipi oggetto, ma serve anche a definire *viste* se avviene tra un tipo oggetto ed un tipo con struttura compatibile ma più *restrittiva* (ad esempio un "range" al posto di un "integer").

3.1.3 Schema di base di dati

Uno schema associa ai nomi di *tipi-classe* e *tipi-valore* la loro descrizione.

I **tipi-valore** sono "strutture" di tipi base.

I **tipi-classe-base** corrispondono alle tradizionali classi di oggetti.

I **tipi-classe-virtuale** corrispondono a viste. Cioè la loro descrizione rappresenta un insieme di condizioni necessarie e sufficienti.

I tipi classe base sono direttamente istanziabili in oggetti mentre quelli virtual sono popolati automaticamente dal sistema.

Consideriamo il concetto di *descrizione ciclica*, una *descrizione è ciclica* se nella descrizione di un nome si fà riferimento, direttamente o indirettamente (tramite altri nomi) al nome stesso.

In uno *schema ben formato* sono ammesse descrizioni cicliche purché nel *ciclo* sia coinvolto almeno un costruttore di tipo oggetto.

In uno *schema ben formato* le relazioni *isa* di ereditarietà non contengono cicli.

In questo modello degli oggetti si suppone che tutti gli schemi siano *schemi ben formati*.

3.1.4 Istanza di uno schema

In questa sezione si associa una semantica al sistema dei tipi attraverso il concetto di *funzione di interpretazione*. La funzione di interpretazione o *interpretazio-*

ne associa ad ogni tipo un insieme di valori e viceversa, associa a tutti i valori ammissibili e/o legali per un database, un tipo.
Il concetto di *istanza possibile* permette di affermare che:

- L'interpretazione di una classe base è un dato sottoinsieme (dato dagli utenti del database) dell'interpretazione della descrizione della classe stessa.
- Per le classi virtuali e per i tipi-valore, l'interpretazione è data proprio dall'interpretazione della descrizione.

3.1.5 Schemi e regole di integrità

I vincoli di integrità sono asserzioni che devono essere vere durante tutta la vita di una base di dati ed hanno lo scopo di imporre la consistenza di una base di dati. I seguenti vincoli di integrità sono già esprimibili nello schema e vengono imposti agli oggetti tramite la nozione di istanza legale:

- *vincoli di dominio*: per ogni attributo è specificato l'insieme dei valori ammissibili
- *vincoli di integrità referenziale*: devono esistere tutti gli oggetti in qualche modo referenziati
- *vincoli di ereditarietà*: un oggetto che appartiene ad una certa classe C deve appartenere anche a tutte le superclassi di C.

Per poter inserire nel modello *regole di integrità* ODL (del progetto ODB-Tools) è stato esteso nel modo specificato in seguito.
Le regole di integrità hanno dalla forma:

$$nome\_regola: \ tipo^a \rightarrow tipo^c$$

ed il seguente significato: per ogni valore v di tipo $tipo^a$ allora v dev'essere anche di tipo $tipo^c$, in altri termini, ogni valore dell'interpretazione di $tipo^a$ dev'essere contenuto nell'interpretazione di $tipo^c$. I tipi $tipo^a$ e $tipo^c$ vengono detti rispettivamente *antecedente* e *conseguente* della regola.
L'estensione con regole ha dato origine al linguaggio OCDL.

3.2 Il linguaggio OCDL

OCDL (Object and Costraint Definition Language) estensione di ODL (del progetto ODB-Tools) con regole, è il linguaggio attraverso il quale è possibile de-

scrivere uno schema di dati. OCDL è supportato da `OCDL-Designer` che *valida* ed ottimizza schemi di basi di dati.

3.2.1 Sintassi

La descrizione di uno schema consiste in una sequenza di definizioni di tipi, classi e regole[1]. Per distinguere i vari costrutti è stato introdotto un prefisso che specifica il tipo del termine che si sta definendo. Tale prefisso può essere:

`prim:` classe primitiva;

`virt:` classe virtuale;

`type:` tipo valore;

`btype :` tipo base;

La definizione di costrutto ha la seguente sintassi:
`prefisso Nome_tipo = descrizione_del_tipo`
dove la `descrizione_del_tipo` può essere descritta dalla seguente grammatica (indico con S la descrizione di un tipo)):

$$
\begin{array}{lll}
S \rightarrow & B & tipo\ atomico \\
 & N & nome\ tipo \\
 & \{S\} & tipo\ insieme \\
 & < S > & tipo\ sequenza \\
 & \hat{}S & tipo\ oggetto \\
 & [a_1 : S_1, ..., a_k : S_k] & tipo\ tupla \\
 & S_1 \& S_2 & intersezione,\ indica\ ereditarieta'
\end{array}
$$

Regole di integrità Una regola di integrità ha la forma

$$nome\_regola:\ tipo^a \rightarrow tipo^c$$

I tipi $tipo^a$ e $tipo^c$ vengono descritti con due definizioni separate, essi possono essere solo classi virtuali e tipi-valore.
Introduciamo cosí quattro nuove tipologie che descrivono le regole di integrità:

[1]Per la sintassi formale dell'OCDL vedere appendice sezione E a pagina 161 .

`antev` : antecedente di una regola di tipo classe virtuale.
`antet` : antecedente di una regola di tipo valore.
`consv` : conseguente di una regola di tipo classe virtuale.
`const` : conseguente di una regola di tipo valore.

Il validatore interpreta i tipi che descrivono una regola come classi virtuali quando la tipologia è `antev` o `consv` mentre li interpreta come tipi-valore quando la tipologia è `antet` o `const`.

Per mantenere la relazione tra antecedente e conseguente di una stessa regola occorre dare un particolare nome al tipo. Il nome della parte antecedente è dato dal nome della regola seguito da una a. Il nome della parte conseguente è dato dal nome della regola seguito da una c.

esempio: data la regola "*Tutti i materiali con rischio maggiore di dieci sono materiali speciali SMATERIAL*":

$$R_2: \quad material \sqcap (\Delta \texttt{risk}: 10 \div \infty) \rightarrow \texttt{smaterial}$$

in OCDL è espressa dalle seguenti definizioni:

```
antev r2a = material & ^ [ risk :  10 0 ]
consv r2c = material & smaterial
```

Per la sintassi formale dell'OCDL si veda l'appendice E.

3.2.2 Esempi di OCDL

Gestione Università Questo esempio è la traduzione in OCDL dell'esempio relativo alla gestione di alcune funzioni di segreteria di un'università definito in precedenza secondo ODMG-93.

```
prim Student =  ^ [ name : string ,
                    student_id : string ,
                    dorm_address : [ college : string ,
                                     room_number : string ],
                    takes : { Section }  ] ;
prim TA      =    Employee &
                  Student  &
                  ^ [  assists : Section  ] ;
prim Professor =  Employee &
                  ^ [ rank : string ,
```

```
                        teaches : { Section }  ] ;
prim Employee =    ^ [ name : string ,
                        id : integer ,
                        annual_salary : integer ] ;
prim Section =     ^ [ number : string ,
                        is_taught_by : Professor  ,
                        has_TA : TA  ,
                        is_section_of : Course  ,
                        is_taken_by : { Student }  ] ;
prim Course =      ^ [ name : string ,
                        number : string ,
                        has_sections : { Section }  ,
                        has_prerequisites : { Course }  ,
                        is_prerequisite_for : { Course }  ] .
```

OCDL permette la definizione di tipi valori direttamente nella dichiarazione delle classi, ad esempio l'attributo `dorm_address` nella dichiarazione della classe `Student`.

Questo stesso esempio, completo di regole di integrità, è riportato nel capitolo del traduttore in 6.2.

Differenze dallo schema descritto in ODL (pagina 19)
(1) L'attributo `Rank` di `Professor` in ODL è di tipo `enum` ed è stato tradotto come `string` in quanto il tipo `enum` in OCDL non esiste.
(2) Le relazioni `relationship` non sono rappresentate in OCDL. Non è quindi possibile esprimere la semantica di ruolo inverso ma solo esprimere ad esempio il ciclo tra `Professor` e `Section`.

appaiono in OCDL come puntatori ad oggetti e perdono parte del loro contenuto informativo.

3.3 Validatore di schemi

Il programma che svolge l'operazione di validazione e ottimizzazione di uno schema è stato chiamato `OCDL-designer`. Tale programma acquisisce schemi di basi di dati ad oggetti complessi espressi nel linguaggio OCDL, opera la trasformazione in forma canonica al fine di controllare la consistenza dello schema e calcola le relazioni `isa` eventualmente implicite nelle descrizioni.

Il programma prende in ingresso un file di testo `nomefile.sc` contenente lo schema iniziale, durante le fasi successive comunica a video eventuali messaggi di errori e incoerenze rilevate, se l'esecuzione ha termine correttamente i risultati

dell'elaborazione vengono scritti in due file: `nomefile.fc` e `nomefile.sb`. Il primo contiene i risultati della trasformazione canonica, il secondo le relazioni di sussunzione e le relazioni `isa` minimali computate. La fase di acquisizione consiste nella lettura del file contenente lo schema e la creazione delle relative strutture dinamiche rappresentanti le definizioni dei tipi (classi). Durante questa fase non viene realizzato un controllo sintattico e semantico sistematico, ma solo alcuni controlli di comodo per rilevare eventuali errori nella scrittura del file di input. Si assume infatti che lo schema preso in input sia corretto dal punto di vista sintattico e semantico, e privo di cicli rispetto alle relazioni `isa` e alle definizioni dei tipi valori (*schema ben formato*).

Una volta acquisito lo schema ha inizio la fase di generazione dello schema canonico. Durante tale fase, il programma individua i tipi (classi) incoerenti, quelli, cioè, la cui descrizione è inconsistente quindi con estensione vuota. Una volta determinata la forma canonica si passa all'esecuzione dell'algoritmo di sussunzione che permette di ricalcolare tutte le relazioni `isa`. Il programma rileva anche le relazioni implicite nella descrizione originale di tipi e classi e determina i tipi e le classi equivalenti. Inoltre calcola le relazioni di sussunzione esistenti tra parte antecedente e conseguente di regole.

I concetti di espansione semantica, sussunzione ed ottimizzazione semantica sono descritti *formalmente* nella breve trattazione teorica riportata in appendice.

Per ulteriori informazioni sul *validatore* vedere la tesi [**?**].

3.4 Ottimizzatore di query

L'obiettivo dell'ottimizzazione semantica delle interrogazioni è quello di trasformare un'interrogazione in una *equivalente* con un minor costo di esecuzione. L'ottimizzazione è detta semantica in quanto le trasformazioni avvengono utilizzando la conoscenza semantica relativa alla base di dati. In particolare si considera il fatto che i vincoli di integrità, espressi per forzare la consistenza di una base di dati, possono essere utilizzati anche per ottimizzare le interrogazioni fatte dall'utente.

L'ottimizzatore realizzato nell'ambito del progetto ODB-Tools ed utilizzato nel prototipo da interfacciare con Internet, ha nome `odbqo`. Questo è composto da due componenti realizzate in due tempi diversi. Un *cuore* in cui avviene l'ottimizzazione semantica tramite l'applicazione delle tecniche descritte in teoria, ed una parte di interfacciamento tra tale *cuore* ed il linguaggio di interrogazione OQL. Il tutto è compilato in un unico programma eseguibile.

L'ottimizzatore realizzato nell'ambito del progetto ODB-Tools è basato sulla relazione di sussunzione, ampiamente descritta in [**?**].

L'approccio proposto è caratterizzato dai seguenti punti:

- *Base di dati e i vincoli di integrità considerati*:

  L'ottimizzazione è riferita al modello ad oggetti complessi OCDL che include regole di integrità esprimibili come implicazioni logiche tra i tipi del sistema. Con questa restrizione, le regole di integrità possono essere trattate efficientemente tramite gli algoritmi di sussunzione.

- *Correttezza ed equivalenza semantica*:

  La precisa semantica sia del modello dei dati che della relazione di sussunzione permettono di garantire formalmente la correttezza delle trasformazioni semantiche.

- *Processo di generazione di interrogazioni equivalenti*:

  La fase principale del processo di generazione, chiamata *espansione semantica*, permette di incorporare ogni possibile restrizione che non è presente nell'interrogazione originale ma che è *logicamente implicata* dall'interrogazione e dallo schema con le regole di integrità. In questo modo viene determinata l'*interrogazione più specializzata* tra tutte quelle semanticamente equivalenti.

- *integrazione con ottimizzazioni convenzionali*:

  La generalità e l'indipendenza da ogni specifico modello di costo e da ogni dettaglio di memorizzazione fisica rendono il metodo adatto per l'integrazione con un ottimizzatore di interrogazione tradizionale.

3.4.1 Interfaccia OQL ODMG-93

È un modulo software che realizza un'interfaccia OQL alle funzionalità del sistema ottimizzatore delle interrogazioni, realizzato da Paolo Apparuti [**?**]. In questa sezione intendo solo citare le caratteristiche principali di tale interfaccia, per approfondimenti si veda [**?**].
L'interfaccia è in grado di:

- Determinare la correttezza sintattica e semantica di una interrogazione OQL;
- Rilevare le parti dell'interrogazione che possono essere tradotte nel formalismo OCDL;
- Generare la corrispondente struttura di memoria centrale che rappresenta la query da fornire in input al modulo ottimizzatore;
- Tradurre la struttura di memoria centrale che costituisce l'output del modulo ottimizzatore in un insieme di espressioni e predicati in linguaggio OQL;

- Sostituire tali espressioni nella query originale, evidenziando le parti ottimizzate.

Il linguaggio OQL presenta un'elevata potenza espressiva e non tutti i suoi costrutti sono rappresentabili nel formalismo usato in ODB-Tools . Questa interfaccia è in grado di trattare una qualunque espressione del linguaggio OQL, identificando al suo interno eventuali parti traducibili da sottoporre la modulo ottimizzatore.

Il passaggio dei dati verso il modulo ottimizzatore avviene attraverso le strutture dati in memoria principale, questo per riuscire a realizzare controlli semantici quali il *type-checking* possibili solo se si dispone di una descrizione completa dello schema della base di dati da interrogare.

Capitolo 4

Confronto tra ODL e OCDL

In questa sezione si confrontano le caratteristiche essenziali di ODL di ODMG-93 e OCDL. ODL e OCDL sono entrambi linguaggi per la descrizione di schemi di oggetti, tuttavia sono nati in ambienti diversi con ambizioni diverse.

OCDL deriva direttamente da un linguaggio matematico sviluppato nell'ambito della teoria della Sussunzione [**?**], è un formalismo con una precisa semantica per fare il controllo di consistenza ed il calcolo della sussunzione. È orientato alla rappresentazione compatta delle classi e delle relazioni tra le classi di oggetti.

ODL, invece, nasce dalla necessità di uno standard nel campo della rappresentazione degli schemi di basi di dati ad oggetti. Come scopo ha quello di essere facilmente utilizzabile. Per rendere questo possibile la sintassi di ODL è stata fortemente influenzata dalla sintassi di quello strumento ad oggetti, molto diffuso, che è il C++. Direi che ODL nasce strizzando l'occhio ai programmatori che si ritrovano tutti i costrutti dichiarativi che sono soliti usare, immersi in un contesto mirato alla descrizione di schemi. Così in ODL ritroviamo costrutti famigliari ai programmatori che conoscono il C.

compattezza Sintatticamente i due linguaggi sono completamente diversi. Possiamo osservare che ODL (in confronto all'OCDL) è più prolisso.

Infatti il modo in cui si dichiarano i dati, nei linguaggi di alto livello è tale da costringere il programmatore ad essere il più chiaro possibile. Questo contribuisce alla riduzione degli errori e facilita il debug delle applicazioni.

Al contrario OCDL, per propria natura (è nato in ambienti di ricerca), è molto compatto, in poche righe è possibile descrivere schemi anche molto complessi. Purtroppo questo rende più facile commettere errori in fase di scrittura di uno schema (basta dimenticarsi un '$\hat{}$').

Nei due paragrafi che seguono sono descritte le caratteristiche tipiche dei due

linguaggi (che rispecchiano le caratteristiche dei modelli degli oggetti adottati) che non hanno una controparte.

Caratteristiche tipiche di OCDL *ranges di valori interi*: tra i tipi base esiste il tipo `range` "compatibile" con il tipo `integer`. Questo tipo può risultare molto utile nel discriminare tra le classi in base al valore assunto da una variabile intera.
regole di integrità del sistema: OCDL è nato supportando le regole di integrità (vedere 3.2.1). Il fatto che ODL non le supporti può essere visto come una grave carenza.
Classi virtuali: Un altro importante concetto supportato dai DBMS relazionali ma che ODL non prevede sono le *viste*. Le classi virtuali di OCDL permettono di descrivere delle viste e sono usate anche nella dichiarazione delle regole di integrità.

Caratteristiche tipiche di ODL Vediamo quali sono le caratteristiche del modello degli oggetti ODM che si trovano in ODL ma non hanno una controparte in OCDL.

1. Dichiarazione delle operazioni. In OCDL non sono contemplate.

2. Gestione errori. In ODMG è previsto che gli oggetti "comunichino" anche mediante le *eccezioni* lanciate dalle operazioni in caso di errore.

3. Gestione operativa delle istanze degli oggetti, si parla si operazioni di database.

4. Il concetto di relazione. In ODL la relazione è una particolare proprietà di una *interface*. In OCDL il modo di mettere in relazione due classi è introdurre tra gli attributi di una classe oggetti dell'altra classe. Le relazioni possono essere ordinate su un dato campo.

5. in ODL c'è distinzione tra l'*interface* e l'estensione della classe che è opzionale. L'estensione di una interface è dichiarata mediante l'uso della parola chiave `extents`. L'estensione di una *interface* può essere associata ad una o più chiavi.

6. ODL supporta i tipi `enum` e `union` che non hanno controparte in OCDL. In generale i tipi di dati utilizzabili sono molto più ricchi[1], si pensi ad esempio agli oggetti collezione oppure agli interi che in ODL sono divisi in `long`, `short`, `unsigned long`, `unsigned short`.

[1]Vedere tabella delle corrispondenze tra tipi sezione 6.1 a pagina 57

7. ODL permette di suddividere uno schema in *moduli*. Ogni modulo può contenere sottomoduli. Si ha così una suddivisione gerarchica dello schema che OCDL non contempla. In OCDL uno schema è *piatto*.

Capitolo 5

Interfaccia verso Internet

In questo capitolo sono descritti i problemi affrontati nel realizzare il sito Web che funge da interfaccia del prototipo verso Internet.

5.1 Cosa si intende per World-Wide Web

Il *World-Wide Web*[1] (WWW, W3, The Web) è un sistema client-server di Internet, ipertestuale e distribuito per la trasmissione delle informazioni. È nato al CERN High-Energy Physics laboratories di Geneva, Svizzera.

Fin dalla nascita (è diventato servizio di pubblico dominio nel 1991) un gran numero di utenti ha lavorato sul Web. Fin dalla nascita gli sviluppatori del CERN hanno messo a disposizione le risorse del Web alle comunità scientifiche. Per dare un'idea della diffusione del Web, si pensi che nel settembre 1993 il traffico attraverso la rete NSFNET (National Science Foundation Network, statunitense) ha raggiunto i 75 gigabytes al mese, mentre a luglio del 1994 il traffico era di un terabyte al mese.

Il formato standard utilizzato nel WWW per la rappresentazione dei documenti (documenti ma anche menu e indici) l'HTML. I documenti sono identificati da una propria URL (Uniform Resource Locator) e questa permette *Links* ipertestuali ad altri documenti. Tali documenti possono essere locali o remoti e accessibili secondo vari protocolli come FTP, Gopher, Telnet, News, o ancora disponibili via il protocollo HTTP (Hypertext Transfer Protocol) usato per trasferire documenti ipertestuali.

Il programma *client* (detto browser) come *Netscape Navigator*, viene eseguito sulla macchina dell'utente e fornisce due operazioni di base per la navigazione. Permette di seguire un link oppure di spedire una richiesta al *server*.

[1] questa definizione è tratta da FOLDOC [**?**]

La maggior parte dei clients e dei servers supporta le *forms* che permettono agli utenti di passare informazioni al server come testo o come selezioni da menu personalizzabili o pulsanti on/off. Dal 1995 i browser supportano anche le "Applet Java", in particolare i browser supportano la "java virtual machine" che è un interprete in grado di eseguire programmi scritti in Java (un linguaggio di programmazione ad oggetti).
Seguendo la grande diffusione e disponibilità di browsers e servers, a partire dal 1995 molte compagnie hanno usato tale software e protocolli per le proprie reti interne TCP/IP dando origine al termine "intranet".
Il W3C[2] (World Wide Web Consortium) è il consorzio che detta gli standard sui protocolli Web. Il W3C è nato il 25 Ottobre 1994 dal MIT/LCS (Massachusetts Institute of Technology Laboratory for Computer Science) e raggruppa anche INRIA (Institut National de Recherche en Informatique et Automatique), il CERN e Keio University Shonan Fujisawa Campus in Asia. Il manuale dell'HTML che è stato usato per realizzare il sito Web è stato copiato dal sito W3C.

5.2 Conoscenze necessarie

Per installare un sito Web occorre:

- un elaboratore connesso ad Internet che funga da server
- un software HTTPd (Hypertext transfer protocol daemon) che giri sul server e risponda alle richieste dei client
- la conoscenza necessaria a scrivere le pagine HTML e a interfacciare altri programmi all'HTTPd tramite le CGI (Common Gateway Interface).

5.2.1 Note sul linguaggio HTML

HTML è un formato per documenti ipertestuali[3] usato nel World-Wide-Web, costruito sul SGML (Standard Generalized Markup Language, standard per la rappresentazione dei documenti).
Caratteristica di questo linguaggio sono i "Tags" . I Tags sono presenti nel testo, sono composti dal carattere "<" una "direttiva", zero o più parentesi ed il carattere ">". Direttive *accoppiate* come "<title>" e "</title>" sono usate per delimitare

[2] attuale indirizzo Internet: `http://www.w3.org/pub/WWW/`

[3] **ipertesto**: da *hypertext*, termine coniato da Ted Nelson intorno al 1965 per una collezione di documenti (detti "nodi") contenenti riferimenti incrociati o "links" che, con l'aiuto di un *browser* interattivo, permette all'utente di muoversi agevolmente tra un documento e l'altro.

testo che deve apparire in posto o in uno stile particolare. Links ad altri documenti hanno la seguente forma:

```
<a name="baz" href="http://machine.edu/subdir/file.html">foo</a>
```

Dove “a” e “/a” delimitano una “anchor” chiamata *baz*, “href” introduce un riferimento ipertestuale che in questo caso è un’URL (nell’esempio è la cosa tra parentesi). Il testo “foo” è ciò che indica il link sul browser.

Altri tags comuni sono:
<p> per un nuovo paragrafo,
, per testo in grassetto,
 per una lista non numerata,
<pre> per testo preformattato,
<form>, </form> per predisporre un modulo per l’input di dati,
<applet>, </applet> per inserire un applet in Java (vedere sezione 7.1.3 a pagina 92),
<h1> , <h2> .. <h6> per le intestazioni.

Per maggiori informazioni sull’HTML rimando a testi e manuali specifici tenendo conto che su Internet vi sono molti manuali “on-line” sempre aggiornati e gratuiti.

Attualmente un ottimo documento redatto da Dave Raggett `<dsr@w3.org>` si trova all’indirizzo:

`http://www.w3.org/pub/WWW/TR/PR-html32-961105.html`

Per ulteriori informazioni sull’HTML fare riferimento al sito W3C[4].

Le forms

Ritengo utile soffermarmi su questo particolare Tag in quanto elemento fondamentale per agganciare il prototipo ad internet. Il Tag FORM all’interno di una pagina HTML indica al browser di produrre una scheda per l’inserimento di dati. Su di uno stesso documento possono esserci più forms ma queste non possono essere innestate.

Formato:

```
<FORM ACTION="url"> ...  </FORM>
```

Attributi dell’intera form sono:

`ACTION` : indica l’URL del programma CGI (Common Gateway Interface) sul server a cui inviare la form che si preoccupa di elaborare i dati della form.

`METHOD` : indica il metodo secondo il quale il browser deve inviare i dati al query server. I metodi supportati sono:

[4]attuale indirizzo internet: `http://www.w3.org`

`GET`: è il metodo di default, si assume che il contenuto della form sia *appeso* all'URL
`POST`: si assume che il contenuto della form sia inviato al server in un particolare pacchetto-dati. Questo è il metodo che ho scelto di usare nella realizzazione del sito.

`ENCTYPE` : è possibile specificare il modo in cui *criptare* il pacchetto-dati del metodo post.

All'interno della form possono apparire tutti tag HTML ed altri che indicano i tipi di campi di input della form stessa.

importante: Ad ogni campo di input è associato un nome. Questo nome è utile al programma CGI che interpreta la form. Permette al programma CGI di discriminare i vari campi.

Tag di input informazioni:

`INPUT` : per specificare semplici campi di input come: una riga di *testo*, una *password*, *checkbox* (un bottone che può essere on o off), *radio* (bottone on/off che può essere messo in mutua esclusione con altri), *submit* (bottone che causa l'invio della form), *reset* (bottone che causa l'azzeramento dei compi), *file* (permette di inviare un file locale), *hidden* (informazioni non visibili all'utente ma che possono essere utili al programma che elabora la form),

`SELECT` : permette di creare menu per la scelta tra un gruppo di opzioni

`TEXTAREA` : mette a disposizione un campo in cui è possibile editare un testo su più righe.

5.2.2 APACHE: il software HTTPd

La comunicazione nel mondo dei Web è basata sul protocollo HTTP. Tale protocollo è gestito da programmi detti HTTPd (HTTP daemon) che sono installati sulle macchine che fungono da server. Il programma scelto come HTTPd è *Apache*.

Il protocollo HTTP L'HTTP (Hypertext Transfer Protocol) è un protocollo a livello di *applicazione* (ISO-OSI) per sistemi distribuiti per lo scambio di informazioni "hipermediali[5]" distribuite. In pratica permette lo scambio di documenti

[5]Estensione dell'ipertesto che incude grafica, suoni, video ed altro.

scritti in HTML . Una caratteristica dell'HTTP è la negoziazione e la tipizzazione della rappresentazione dei dati che permette ai sistemi di gestione dell'HTTP di essere costruiti indipendentemente dai dati che devono essere trasferiti.

Per la realizzazione di questa tesi NON è stato necessario approfondire la conoscenza dell'HTTP.

Apache Apache[6] è un server HTTP di pubblico dominio per Unix. Il progetto Apache è nato nel 1995 basato sul codice e l'idea del più diffuso server HTTP dell'epoca: NCSA httpd 1.3. In seguito è stato ulteriormente sviluppato, sono state aggiunte nuove funzionalità ed è diventato più veloce. Oggi è il più popolare dei server WWW, a novembre del 1996 Apache girava su circa 200000 servers (circa il 36% dei servers WWW).

Di Apache ho gestito:

installazione : Reperimento dei sorgenti e compilazione. Questa parte non è gravosa, ho scaricato i sorgenti dal sito di Apache, e ho seguito le istruzioni indicate nei file "README". Non ha dato problemi di compilazione.

avvio : Come mettere *on-line* la prima pagina HTML: Generato l'eseguibile di Apache è necessario configurare il server manipolando i files di configurazione. Apache ha tre files di configurazione. `httpd.conf` che descrive gli attributi del server quali, la porta da utilizzare, con che nome-utente il server deve girare, ed altro ancora. `srm.conf` in cui è indicata la directory da utilizzare come radice per l'albero dei documenti, contiene anche altre informazioni che riguardano l'utilizzo di funzioni particolari di Apache. `access.conf` che permette di configurare diritti di accesso esecuzione per ogni directory dell'albero dei documenti. Il tutto è risultato semplice, ho trovato questo "prodotto" ben fatto.

mettere i documenti on-line Finalmente, una volta scritte le pagine HTML, basta metterle nelle directory già citate nei files di configurazione, lanciare l'eseguibile, ed il server funziona.

protezione delle pagine del prototipo Per effettuare un controllo degli accessi, sapere chi è interessato al prototipo, e' stato introdotto un meccanismo di protezione delle pagine HTML relative al prototipo. Chiunque desideri provare il prototipo deve prima *registrarsi*. Il meccanismo di protezione, basato su *logname* e *password* è una funzionalità di Apache. Per attivare tale funzionalità occorre predisporre un particolare file delle password ed indicare nel file `Access.conf` quali sono le directory da "proteggere"

[6]Queste informazioni sono state prese da uno dei mirror del sito Apache (indirizzo: `http://www.nonsoloweb.it/apache/`).

Manutenzione dell'HTTPd La manutenzione di server che utilizzano *Apache* non risulta complicata. Quando si modificano le pagine HTML o si aggiornano gli script CGI, non è necessario comunicarlo al server. Solo quando si modificano i files di configurazione è necessario comunicare ad Apache che la configurazione è cambiata. Questo si fà lanciando il segnale UNIX `SIGHUP` al processo httpd. In pratica si usa il comando:

```
kill -1 ‘cat log/httpd.pid‘
```

Che forza Apache a rileggere i files di configurazione. Il file log/httpd.pid è il file in cui Apache scrive il proprio `pid` (Process IDentifier).

5.2.3 Specifiche CGI

Introduzione Le CGI (Common Gateway Interface) sono uno standard per interfacciare applicazioni esterne con server per le comunicazioni come i server HTTP o Web. Possono essere programmi CGI tutti quelli che accettano argomenti da linea di comando o da standard input.

Le CGI sono nate per risolvere il problema di collegare una database al sistema Web. Per fare ciò occorre un programma CGI che possa essere eseguito dal Web daemon per trasmettere informazioni al DBMS e ricevere da questo le informazioni da mostrare al client.

Non ci sono limiti alle cose che si possono collegare al Web tramite le CGI. Un programma CGI può essere scritto in qualunque linguaggio di programmazione. L'unica cosa da ricordare è che il programma lanciato deve durare poco tempo, in quanto l'utente attende che il programma termini.

CGI specifica come passare gli argomenti al programma da eseguire come parte della richiesta HTTP. Definisce inoltre un insieme di utili variabili di ambiente (del sistema operativo) aggiornate dal Web server utilizzabili dal programma chiamato.

Normalmente il programma chiamato genera in output un pagina HTML che viene ripassata al browser. La pagina HTML *ritornata* può essere creata in modo arbitrario e dipendere in modo arbitrario dai dati in ingresso. Una scelta comune è utilizzare il linguaggio `Perl` per scrivere scripts CGI.

Note sulla sicurezza Un programma CGI è un'eseguibile. Fondamentalmente usare programmi CGI significa permettere al mondo intero di eseguire programmi sul proprio sistema, che non è una cosa molto sicura.

Una precauzione da prendere quando si usano le CGI è predisporre un direttorio particolare in cui mettere i programmi CGI, questa dir può essere configurata in modo da essere l'unica dir in cui il Web server può caricare programmi da esegui-

re. Per ottenere un controllo ancora maggiore si può configurare la directory in modo che possa essere gestita dal solo WebMaster.

Utilizzo Nella tesi si è scelto di utilizzare per i programmi CGI degli scripts in bourne shell. Questi scripts vengono invocati dal server WWW ogni qual volta un client esegue il *submit* di una *form*. A loro volta gli script richiamano i programmi del prototipo che sono scritti in ANSI C. I programmi che si preoccupano della formattazione HTML dell'output sono gli scripts.

5.3 Organizzazione del sito

Il sito Web è un documento ipertestuale e mal si presta ad una descrizione a parole, per capirne la struttura ed il contenuto è molto più veloce, facile ed intuitivo visitare il sito con un browser.
L'obbiettivo di questa tesi è realizzare la struttura base del sito ed in particolare realizzare un'interfaccia funzionale dei programmi del prototipo verso internet. Questo è il motivo per cui ho ho scelto di limitare la spiegazione del contenuto del sito.
Il sito è nato come sito del prototipo ODB-Tools e quindi contiene una presentazione del progetto, la relativa documentazione, riferimenti a varie pubblicazioni ed una demo del prototipo.
La pagina indice e le altre del sito sono scritte direttamente in HTML. La documentazione è scritta in HTML ottenuto dalla traduzione di documenti scritti in LaTex. Il prototipo è interfacciato ad internet utilizzando lo standard CGI che *Apache* supporta.

5.3.1 Il prototipo

In figura 5.1 è mostrata l'organizzazione della parte di `sito` che riguarda la *demo* del prototipo. Il prototipo permette di effettuare fondamentalmente due operazioni:

- Input, validazione ed ottimizzazione dello schema di una di base di dati ad oggetti;

- Ottimizzazione semantica delle query sullo schema validato.

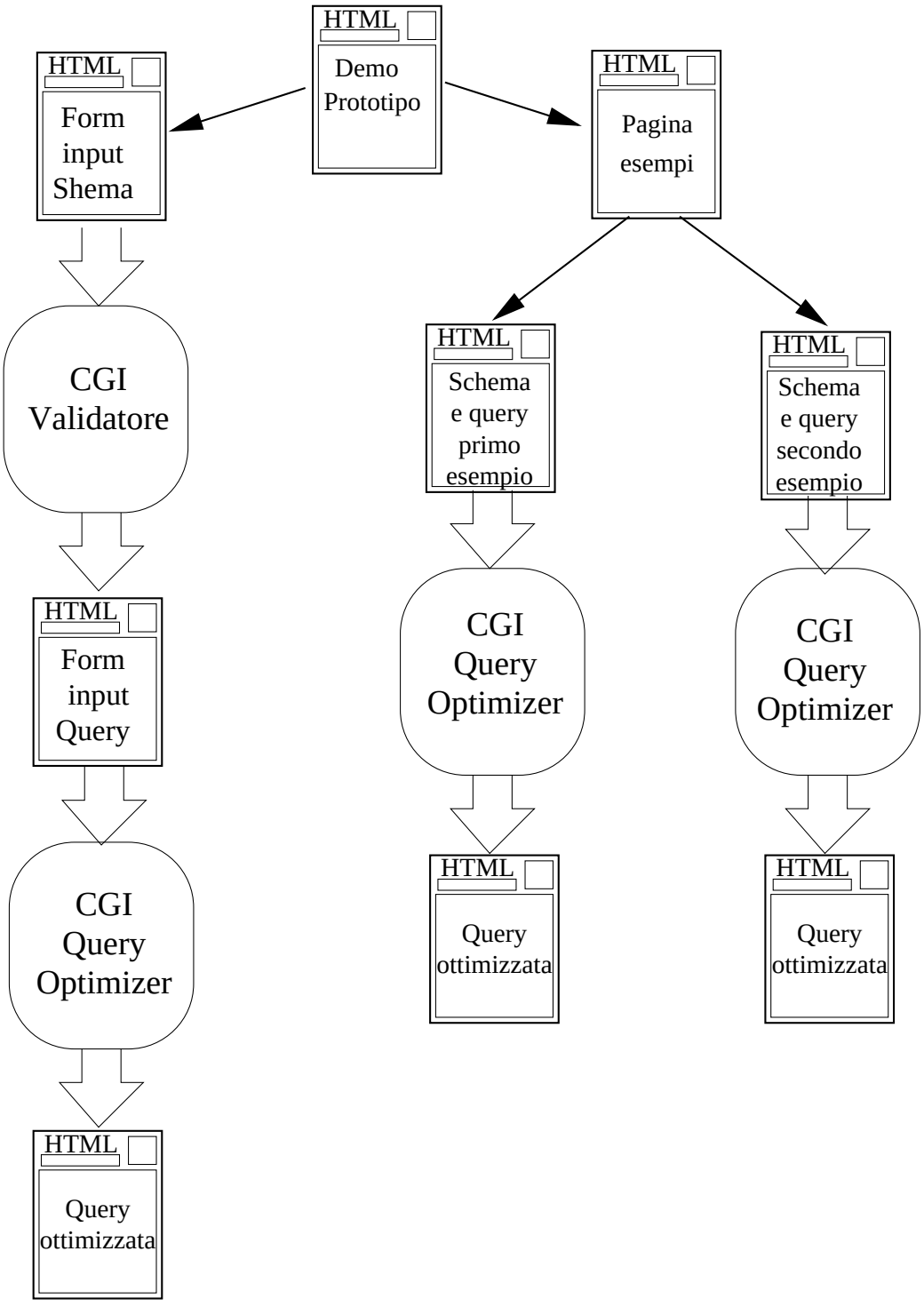


Figura 5.1: Organizzazione della demo del prototipo

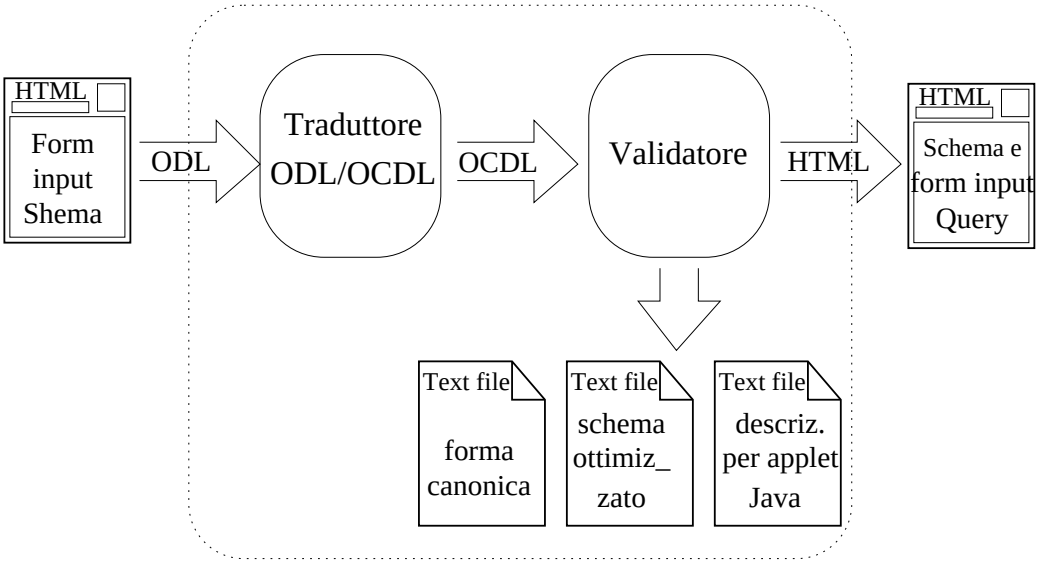


Figura 5.2: Valdiazione dello schema

Validazione

Lo schema da validare dev'essere descritto in ODL secondo la sintassi descritta in sezione 6.1.1. È possibile inviare lo schema da validare al validatore in due modi: scrivere l'intero schema in un campo testo, oppure inviare al validatore un file in cui è contenuta la descrizione dello schema.

Lo schema viene inviato al modulo traduttore che traduce la sintassi ODL in OCDL. Se il traduttore riesce a tradurre lo schema in OCDL allora lo schema OCDL viene passato al modulo di validazione-ottimizzazione.

Il validatore (vedere sezione 3.3) esegue le seguenti funzioni:

- Controlla la correttezza semantica dello schema;
- Verifica se è possibile ottimizzare in qualche modo lo schema;
- Genera lo schema OCDL ottimizzato;
- Genera un file che è interpretato dall'applet java che dà una rappresentazione grafica dello schema.

Se la validazione termina senza errori, viene mostrata una pagina HTML che contiene il tempo di esecuzione del validatore, la rappresentazione grafica dello schema generata dall'applet java `scvisual` e una `form` per l'inserimento delle query sullo schema dato da ottimizzare.

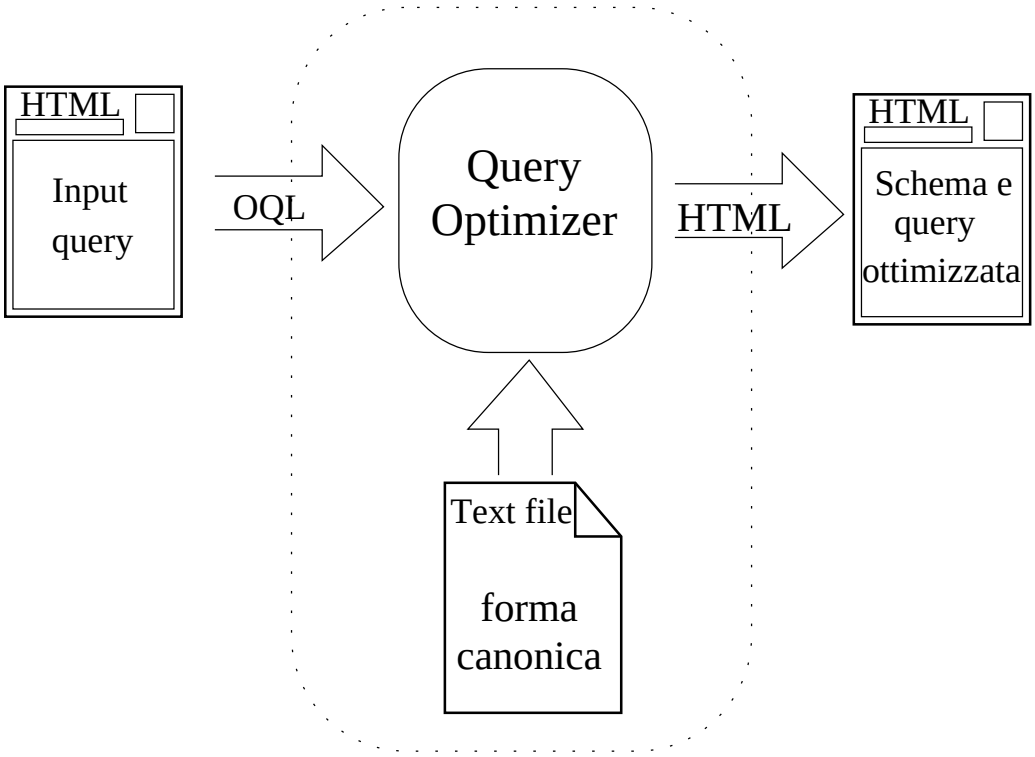


Figura 5.3: Ottimizzazione di una query

Ottimizzazione semantica delle query

Dalla pagina generata dalla fase di validazione è possibile inserire query in OQL secondo lo standard proposto dall'ODMG. Anche l'input delle query è possibile attraverso un campo testo oppure, per chi le avesse già preparate, inviando il contenuto di un file presente sulla macchina client.

Una volta inviata la query, questa viene elaborata dal modulo di ottimizzazione semantica della query[7]. Tale modulo svolge le seguenti funzioni:

- Controllo sintattico e semantico della query.
- Verifica se è possibile ottimizzare la query stessa.
- Genera la versione OQL ottimizzata della query.
- Genera un file che è interpretato dall'applet java che dà una rappresentazione grafica dello schema e della query.

[7]per una completa descrizione del modulo di ottimizzazione semantica delle query su uno schema validato si veda [**?**].

La fase di ottimizzazione della query termina senza errori, viene mostrata una pagina HTML che contiene: informazioni relative al tempo di esecuzione dell'ottimizzatore, la rappresentazione grafica dello schema che mostra come è trattata la query, la query ottimizzata.

5.4 Soluzioni adottate

Una volta installato Apache, il server è stato in grado di mettere in internet le pagine HTML.
Il problema era trovare un modo per permettere ad un qualunque utente di internet di spedire uno schema in ODL al nostro server e provare il prototipo. In pratica occorreva inserire alcune pagine che permettessero di trasmettere informazioni dai client verso il server. Il problema è stato risolto con l'utilizzo dell'interfaccia CGI.
Un altro problema era quello di permettere a più utenti di accedere contemporaneamente al prototipo. Questo problema è stato risolto adottando una politica di gestione di files temporanei.

5.4.1 Utilizzo dell'interfaccia CGI

L'input dei dati avviene tramite una FORM che permette l'input del testo direttamente in un campo testo o tramite l'invio di un file testo. Il metodo usato per l'invio dei dati è il metodo POST.

il metodo POST .
Questo metodo causa l'invio allo standard input del programma CGI (quello che interpreta la form) un pacchetto di dati che ha questo formato:
separatore
name="*nome variabile*"
riga vuota
righe con il contenuto della variabile

separatore
name="*nome variabile*"
riga vuota
righe con il contenuto della variabile
...

separatore è una data stringa che si trova nella variabile di ambiente `CONTENT_TYPE` dopo il tag "`boundary=`". *nome variabile* è il nome dato al campo all'interno della form.

5.4.2 Gestione dei files temporanei

Un altro problema era quello di permettere a più utenti di accedere contemporaneamente al prototipo.
Vediamo di spiegare qual'è il problema. Gli algoritmi del prototipo sono codificati in programmi che lavorano su files con certe estensioni. Ad esempio traduttore ODL→OCDL viene eseguito da riga di comando con la seguente sintassi:

`odl-trasl nome_file`

legge dal file `nome_file.odl` e genera la traduzione OCDL nel file `nome_file.sc` e la rappresentazione in formato *visual* nel file `nome_file.odl.vf`. Analogamente il validatore legge il file `nome_file.sc` e genera files tra cui `nome_file.cf` (forma canonica) `nome_file.vf` (file interpretato dall'applet `scvisual`).
Ammettiamo che la descrizione ODL dello schema venga messa in un certo file di nome `nome_file.odl`. Se più di una persona accedono contemporaneamente al prototipo tale file viene sovrascritto più volte e può capitare che i files relativi alla forma ODL, a quella OCDL o alla forma canonica dello schema siano relativi a schemi diversi.
Con la gestione di files temporanei questo problema non si pone. Ogni volta che viene inviato uno schema, questo viene memorizzato in un file creato per l'occasione, poniamo per esempio di nome `tmp_name.odl`. Tutti i moduli del prototipo possono così essere lanciati sul file con prefisso `tmp_name` senza che le informazioni provenienti da utenti diversi si possano intrecciare.

Politica di gestione dei files temporanei Immerso tra il codice che gestisce l'interfaccia CGI c'è l'algoritmo di gestione dei files temporanei. Questo algoritmo è realizzato nel programma C `genera_tmp_file`. *Funzionamento dell'algoritmo*: gli elementi coinvolti dall'algoritmo sono:

- Un prefisso con cui i files temporanei devono iniziare.
- Una directory in cui memorizzare i files temporanei.
- Un file *contatore* contenete un *progressivo* di sette cifre. Tale file è usato anche come semaforo.
- Un demone (uno script lanciato in background) che cancella i files temporanei *vecchi*.

Ogni volta che c'è bisogno di un nuovo file temporaneo l'algoritmo di gestione dei files temporanei apre in modo esclusivo il file *contatore* (questo garantisce che venga creato un solo file temporaneo alla volta), legge il valore del contatore, cerca il file temporaneo che ha come estensione il numero letto, se tale file non esiste, crea il file temporaneo, incrementa il contatore, lo scrive nel file *contatore* e chiude il file. Se invece il file temporaneo che ha come estensione il numero letto esiste incremento il contatore fino a che non trovo un nome di un file che non esiste.

Utilizzo dei files temporanei In realtà, durante l'elaborazione di uno schema vengono creati più di un file. La routine di gestione dei file temporanei trova un nome di file unico a cui verranno aggiunte più estensioni. Le estensioni usate sono:

estensione	contenuto
	gesitione dell'input
.in	ciò che la FORM passa al programma CGI
.odl	schema in formato ODL
.err	errori generati dai vari programmi
.oql	query sullo schema in formato OQL
	generati dal traduttore
.sc	schema in formato OCDL
.odl.vf	descrizione schema in formato *visual*, e' il formato interpretato dall'applet java
	generati dal validatore
.fc	forma canonica
.ft	fattori dello schema
.sb	sussunzione
.vf	descrizione schema ottimizzato in formato *visual*
	generati dal ottimizzatore
.oql.fc	forma canonica della query
.oql.sb	sussunzione della query
.oql.vf	schema in formato *visual* mostra dove si trova la query

Tabella 5.1: Varie estensioni dei files temporanei

5.4.3 La gestione dell'input dello schema

La pagina HTML che è presentata all'utente contiene due FORM per l'invio dello schema. Le due form sono fondamentalmente uguali, la prima presenta un campo di *input testo* in cui è possibile inserire lo schema scrivendolo direttamente nella pagina. La seconda invece permette di inserire lo schema tramite un campo *input file*, questo permette di inviare un file locale. Questi due sono associati allo stesso nome: `testo_schema` in questo modo le due forms possono essere elaborate dallo stesso programma CGI.
Il programma CGI che elabora la form è `sc_form.script` , esegue le seguenti operazioni:

1. Cerca un nome-file temporaneo da usare per le elaborazioni. Per fare ciò lancia il programma `genera_tmp_file`.

2. Salva il contenuto dello standard input nel file temporaneo con estensione `.in`

3. Lancia il programma `estrai_post_var` che estrae il contenuto della variabile `testo_schema` e lo salva nel file temporaneo con estensione `.odl`.

4. Lancia il *traduttore* che genera il file di estensione `.sc`.

5. Lancia il *validatore*.

6. Crea una pagina HTML con il risultato della validazione, l'*applett java* che mostra graficamente struttura dello schema e la FORM per l'inserimento delle query sullo schema ottimizzato.

Il risultato della gestione della FORM di input dello schema è una pagina HTML che permette di provare l'*ottimizzatore semantico* delle query. Questa pagina contiene una FORM in cui è memorizzato un campo *hidden* di nome `temp_name` in cui è memorizzato il nome del file temporaneo che contiene lo schema. Questo campo permette alla procedura di gestione di input delle query di sapere su quale schema viene eseguita la query.

5.4.4 La gestione dell'input delle query

Il programma CGI che elabora la form è `sc_form.script` , esegue le seguenti operazioni:

1. Cerca un nome-file temporaneo da usare per le elaborazioni. Per fare ciò lancia il programma `genera_tmp_file`. Tale file temporaneo è usato per memorizzare il contenuto dello standard input e passare all'applet Java un nome file sempre diverso in modo che questa sia costretta a leggere i dati ad ogni nuova query.

2. Lancia il programma `estrai_post_var` che estrae il contenuto della variabile `temp_name` cioè il nome del file temporaneo in cui è memorizzato lo schema, e il contenuto della variabile `testo_query` che contiene il testo della query. Il testo della query viene memorizzato nel file temporaneo `temp_name` con estensione `.oql`.

3. Lancia l'*ottimizzatore delle query* sul file temporaneo.

4. Crea una pagina HTML con il risultato dell'ottimizzazione.

5.4.5 La gestione degli accessi

Una cosa che sembrava utile adottare è un sistema di controllo degli accessi al prototipo, a scopo puramente informativo.

Il sistema di controllo è stato realizzato sfruttando una particolare caratteristica di Apache e dei Browser HTTP.

Apache puo' essere configurato in modo da consentire l'accesso a documenti WWW solo agli utenti presenti in un particolare *file degli utenti* in cui sono memorizzati gli utenti riconosciuti dal server HTTPd e le relative password necessarie per certificarne l'identità. Login e password vengono richiesti all'utente quando si tenta di accedere ad un documento protetto. Questo controllo degli accessi però è *intelligente*, nel senso che, il *server* è in grado di riconoscere la connessione con il Browser e la procedura di identificazione avviene solo la prima volta che si accede ad una pagina protetta. Così, fino a che si utilizza l'attuale "sessione di lavoro" del browser si possono consultare, senza più dover comunicare logname password, tutte le pagine protette. Nel nostro sito, tutti gli utenti presenti nel file degli utenti hanno i medesimi diritti anche se Apache fornisce un controllo degli accessi più sofisticato, alla UNIX, cioè è possibile definire dei gruppi di utenti e specificare per ogni documento un gruppo in modo che tale documento sia accessibile solo dagli utenti del gruppo dato.

Chiunque può accedere al sito purchè si registri. La procedura di registrazione impone di fornire i propri dati e inserisce un nuovo utente nei file degli utenti autorizzati. Il logname e la password sono decisi da chi si effettua la registrazione.

La procedura che permette la registrazione è realizzata usando da una pagina HTML (`subscription_form.html`) che contiene la `FORM` per l'input dei

dati dell'utente e da uno script CGI (`register.script`) che estrae i dati dalla form, li inserisce in un archivio (`users.txt`) e aggiorna il file degli utenti riconosciuti dal sistema (`passwd.txt`).

Il programma (`pass.c`) che permette la gestione del file degli utenti riconosciuti dal sistema è fornito con i sorgenti di Apache. Purtroppo tale programma è iterativo, così, per poter realizzare la routine CGI di registrazione automatica ho dovuto modificarne il sorgente in modo che funzioni in modo batch. Passando come parametri al programma `mypass` il nome del file con le password, il logname e la password aggiunge l'utente tra gli utenti riconosciuti dal sistema.

Capitolo 6

Il Traduttore

Dato uno schema di base di dati ad oggetti descritta in ODL-ODMG, scopo del traduttore è generare la descrizione del medesimo schema in OCDL ed in formato `vf` (*Visual Form*) in modo che sia visualizzabile utilizzando l'applet Java `scvisual`. Il programma che esegue la traduzione è stato chiamato `odl_trasl`.

In questo capitolo è descritto come usare il programma, come funziona e sono approfonditi gli algoritmi più complessi.

6.1 Manuale utente

sintassi:

```
odl_trasl [nome_schema]
```

`nome_schema` è il nome del file che contiene schema da elaborare. Se si specifica questo parametro il testo ODL è letto dal file `nome_schema.odl`, la traduzione in OCDL è scritta nel file `nome_schema.sc`. la traduzione in visual form è scritta nel file `nome_schema.vf`. Se non è specificato alcun parametro il traduttore si aspetta la descrizione dello schema su `standard input` e scrive le traduzione vf e OCDL su `standard output`.

gestione degli errori:

In caso di *successo* il programma ritorna il codice di errore 0. In caso di *insuccesso* il programma ritorna un codice di errore diverso da 0 e scrive la causa dell'errore su `standard error`.

Il file di input dev'essere scritto in ODL (vedere ODL "standard" sezione 2.3 a pagina 17). L'ODL che il traduttore accetta è un sottoinsieme di quello proposto dall'ODMG. Per questo motivo l'ODL riconosciuto si differenzia per i tipi di dati

accettati (non accetta `enum` o `union` ma riconosce il tipo `range`) ed accetta costrutti che non sono presenti in ODL (le `rules`).
Vedere di seguito la sintassi dell'ODL accettata dal traduttore.
La sintassi della visual form e' descritta nella sezione 7.2.

6.1.1 Modifiche all'ODL

L'ODL che il *traduttore* riconosce presenta le caratteristiche sotto elencate.

Restrizioni rispetto ODL standard Per restrizioni intendo tutte quelle differenze che introducono incompatibilità verso l'ODL standard. A causa di queste restrizioni uno schema scritto in ODL standard può risultare non traducibile.

- Le relazioni perdono di significato. Poichè in OCDL non esiste il concetto di relazione, le `relation` sono mappate in OCDL come coppie di attributi. Gli attributi generati dalla traduzione di una `relationship` sono **indipendenti** tra loro;
- Non sono supportati i tipi `enum` e `union`;
- ODL permette una suddivisione gerarchica dello schema mediante l'uso dei *moduli*. Lo schema OCDL è invece piatto. Questo fa si che nell'ODL del traduttore i nomi de `struct`, `typedef`, `const` e `interfaces`, debbano essere unici per tutto lo schema;
- Non è possibile distinguere un'interface dalla sua estensione. Nella traduzione in OCDL viene considerato solo il nome dell'interface. In ODL questa suddivisione è chiara, si può dare il nome che si vuole all'estensione di un'interface, in OCDL, invece, la parte intensionale ed estensionale di una classe sono referenziate dallo stesso nome;
- Le costanti possono essere solo dei *letterali*. Non possono essere espressioni, tipi composti o tipi ridefiniti.

Per gli altri costrutti di ODL non supportati da OCDL come le *operazioni* e le *eccezioni*, viene effettuato il controllo sintattico. Se sono corretti sintatticamente non causano errore ma sono comunque ignorati dal traduttore.

Estensioni rispetto ODL standard

- Supporto delle regole di integrità (`rules`);

- Supporto delle classi virtuali (`virt`); La sintassi per dichiarare le virt è la stessa di quella delle interfaces;
- Introduzione dei tipi base `range` e `int`;
- È possibile dichiarare direttamente delle collection all'interno delle struct;
- Sono accettati come parametri da passare ad una operazione anche i tipi collezione.

6.1.2 Corrispondenze tra tipi

La tabella 6.1 riassume le scelte fatte per mappare i tipi di dati presenti in ODL in OCDL.

ODL	OCDL
`int`	`integer`
`long`	`integer`
`unsigend short`	`integer`
`unsigend long`	`integer`
`short`	`integer`
`float`	`real`
`double`	`real`
`char`	`string`
`string`	`string`
`boolean`	`boolean`
`interface`	`prim`
`view`	`virt`
`const`	`btype`
`struct`	`type`
`typedef`	`type o btype`
`set`	`{ }`
`bag`	`{ }`
`list`	`< >`
`array`	`< >`

Tabella 6.1: Corrispondenze tra tipi ODL e OCDL

In questo capitolo per *variabili di tipo intero* intenderemo quelle variabili dei tipi che mappano sul tipo `integer`.

6.1.3 Traduzione dei principali concetti

In questa sezione sono presentate, schematicamente, le principali corrispondenze tra i costrutti ODL e quelli OCDL, queste corrispondenze danno un'idea del funzionamento e della complessità del problema.

Dichiarazione di una classe

```
ODL:   interfaces  Employee {...};
OCDL:  prim Employee ^ [...];
```

In ODMG dichiarare un'interface serve per dichiarare esplicitamente le 'caratteristiche' visibili degli oggetti di un certo tipo.

Subtype, ereditarietà

```
ODL:   interface Professor: Employee {...};
OCDL:  prim Professor = Employee & ^ [...];
```

Ereditarietà multipla

```
ODL:   interface Teaching_Assistant: Employee, Student {...};
OCDL:  prim Teaching_Assistant = Employee & Student & ^ [...];
```

`struct` **(tipi strutturati)**

```
ODL:   struct Address
          {
          string college;
          string room_number;
          };
OCDL:   type Address = [  college : string ,
                          room_number : string ];
```

Attributi

```
ODL:   attribute string name;
       attribute float arr[10];
       attribute int arr3[10][20][30];
OCDL:  name : string,
       arr  : <float>,
       arr3 : <int>,
```

Si noti che il costruttore *array* ODMG può essere solo mappato nel costruttore *serie* OCDL.

Relazioni

```
ODL:   -- in Section
       relationship Professor is_taught_by
                    inverse Professor::teaches;
       -- in Professor
       relationship set<Section>teaches
                    inverse Section::is_taught_by;
OCDL:  -- in Section
       is_taught_by: Professor,
       -- in Professor
       teaches: {Section},
```

In OCDL non esiste il costruttore `relation` dell'ODL. Le `relation` vengono mappate in coppie di attributi.
Siamo consapevoli che questo riduce il contenuto informativo dello schema. Questo è un problema noto su cui si stà lavorando, infatti è prevista l'estensione della teoria sul modello OCDL in modo da supportare le relazioni intensionali bidirezionali.

Inoltre le regole sono esprimibili sono in OCDL. Abbiamo introdotto un'estensione di ODL per rappresentarle.

Regole di integrità

```
ODL:   rule rule_1 forall X in Professor:
           ( X.rank = "Research" )
       then
           X.annual_salary <= 40000;
OCDL:  antev rule_1a = Professor &
                       ^ [ rank : vstring "Research" ] ;
       consv rule_1c = Professor &
                       ^ [ annual_salary : range 0 40000 ] .
```

6.2 Un esempio di traduzione

In questa sezione è descritto uno schema ODL completo accettato dal traduttore.

Lo schema è essenzialmente lo stesso di quello in sezione 2.3.3. In questo esempio si può vedere come possono essere sfruttate le potenzialità derivate dall'OCDL per introdurre informazioni aggiuntive direttamente nello schema.

L'esempio descrive lo schema di un database in cui si registrano informazioni su studenti, professori, assistenti, corsi e sessioni dei corsi all'interno di un'università.

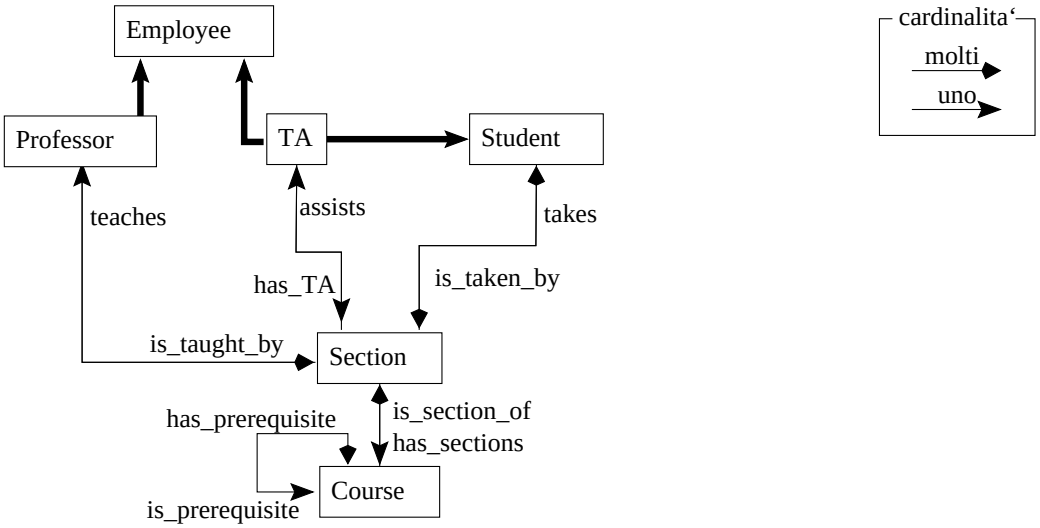


Figura 6.1: Schema dell'esempio

In figura 6.2 è rappresentato lo schema senza le `rules`, in figura 6.2 vi è la rappresentazione completa dello schema in cui sono rappresentate anche le regole di integrità.

```
interface Course        ( extent courses keys name, number )
{       attribute string name;
        attribute unsigned short number;
        relationship list<Section> has_sections
                inverse Section::is_section_of
                {order_by Section::number};
        relationship set<Course> has_prerequisites
                inverse Course::is_prerequisite_for;
        relationship set<Course> is_prerequisite_for
                inverse Course::has_prerequisites;
        boolean offer (in unsigned short semester)
                raises (already_offered);
        boolean drop  (in unsigned short semester)
                raises (not_offered);
        };
interface Section       ( extent sections key (is_section_of, number))
{       attribute string number;
        relationship Professor is_taught_by
                inverse Professor::teaches;
        relationship TA has_TA
                inverse TA::assists;
        relationship Course is_section_of
```

```
                inverse Course::has_sections;
        relationship set<Student> is_taken_by
                inverse Student::takes;
        };
interface Employee      ( extent employees key (name, id) )
{       attribute string name;
        attribute short id;
        attribute unsigned short annual_salary;
        void fire () raises (no_such_employee);
        void hire ();
        };
interface Professor: Employee   ( extent professors )
{       // attribute enum Rank { full, associate, assistant} rank;
        attribute string rank;
        relationship set<Section> teaches
                inverse Section::is_taught_by;
        short grant_tenure () raises (ineligible_for_tenure);
        };
interface TA: Employee, Student ()
{       relationship Section assists
                inverse Section::has_TA;
        };
interface Student       ( extent students  keys name, student_id)
{       attribute string name;
        attribute integer student_id;
        attribute struct Address
                  {
                        string college;
                        string room_number;
                  } dorm_address;
        relationship set<Section> takes
                inverse Section::is_taken_by;
        boolean register_for_course
                (in unsigned short course, in unsigned short Section)
                raises (unsatisfied_prerequisites, section_full,
                       course_full);
        void drop_course (in unsigned short Course)
                raises (not_registered_for_that_course);
        void assign_major (in unsigned short Department);
        short transfer(in unsigned short old_section,
                        in unsigned short new_section)
                raises (section_full, not_registered_in_section);
        };
```

```
//
// rules
//
rule rule_1 forall X in Professor:  X.rank = "Research"
   then                             X.annual_salary <= 40000 ;

rule rule_2 forall X in Professor:  X.rank = "Associate"
   then   X.annual_salary >= 40000 and X.annual_salary <= 60000 ;

rule rule_3 forall X in Professor:  X.rank = "Full"
   then                             X.annual_salary >= 60000 ;

rule rule_4 forall X in Employee:   X.annual_salary >= 40000
   then                             X in Professor ;

rule rule_5 forall X in Employee:   X.annual_salary <= 30000
   then                             X in TA  ;

rule rule_6 forall X in Student:    X.student_id <= 2000
   then                             X in TA  ;

rule rule_7 forall X in Professor:  X.rank  = "Full"
   then
     exists X1 in X.teaches:
        ( X1.is_section_of in Course and
          X1.is_section_of.number = 1 ) ;
```

Le regole in questo caso sono usate per classificare gli oggetti delle classi `Student`, `Employee` e `Professor`.

Le regole dalla 1 alla 3, determinano i livelli del salario in base alla qualifica del professore,

`rule_1` se il professore è di “rango” “research” allora il salario è inferiore a 40000,

`rule_2` se il professore è di “rango” “Associate” allora il salario è compreso tra 40000 e 60000,

`rule_3` se il professore è di “rango” “Full” allora il salario dev’essere superiore a 60000.

Le 4 e 5 discriminano tra gli impiegati in base allo stipendio.

La regola `rule_4` impone che se un impiegato ha uno stipendio maggiore di 30000 allora tale impiegato è un professore.

Analogamente la regola `rule_5` impone che se un impiegato ha uno stipendio inferiore a 30000, tale impiegato è un assistente.

Le regole 6 e 7 sono regole di esempio.
La regola `rule_6` impone che i codici identificatori degli studenti inferiori a 2000 sono riservati agli assistenti.
La `rule_7` afferma che se un professore è di tipo “full” allora insegna almeno in una sezione del corso numero “1”. Questa regola è stata introdotta solo per mostrare le possibilità offerte dall’OCDL.
Di seguito è riportata la traduzione OCDL generata dal traduttore.

```
type Address =    [  college : string , room_number : string ] ;
prim Student =  ^ [ name : string ,
                    student_id : integer ,
                    dorm_address : Address ,
                    takes : { Section }  ] ;
prim TA =       Employee &
                Student &
                ^ [  assists : Section  ] ;
prim Professor =  Employee &
                  ^ [ rank : string ,
                      teaches : { Section }  ] ;
prim Employee = ^ [ name : string ,
                    id   : integer ,
                    annual_salary : integer ] ;
prim Section =  ^ [ number : string  ,
                    is_taught_by : Professor  ,
                    has_TA : TA  ,
                    is_section_of : Course  ,
                    is_taken_by : { Student }  ] ;
prim Course =  ^ [ name   : string ,
                   number : integer ,
                   has_sections : { Section }  ,
                   has_prerequisites : { Course }  ,
                   is_prerequisite_for : { Course }  ] ;
antev rule_7a = Professor & ^ [ rank : vstring "Full" ] ;
consv rule_7c = Professor &
                ^ [ teaches :
                     !{ ^ [ is_section_of : ^ [ number : vstring "1" ] ,
                            is_section_of : Course ] } ] ;
antev rule_6a = Student & ^ [ student_id : range -inf 2000 ] ;
consv rule_6c = Student & TA ;
antev rule_5a = Employee & ^ [ annual_salary : range -inf 30000 ] ;
consv rule_5c = Employee & TA ;
antev rule_4a = Employee & ^ [ annual_salary : range 30000 +inf ] ;
consv rule_4c = Employee & Professor ;
```

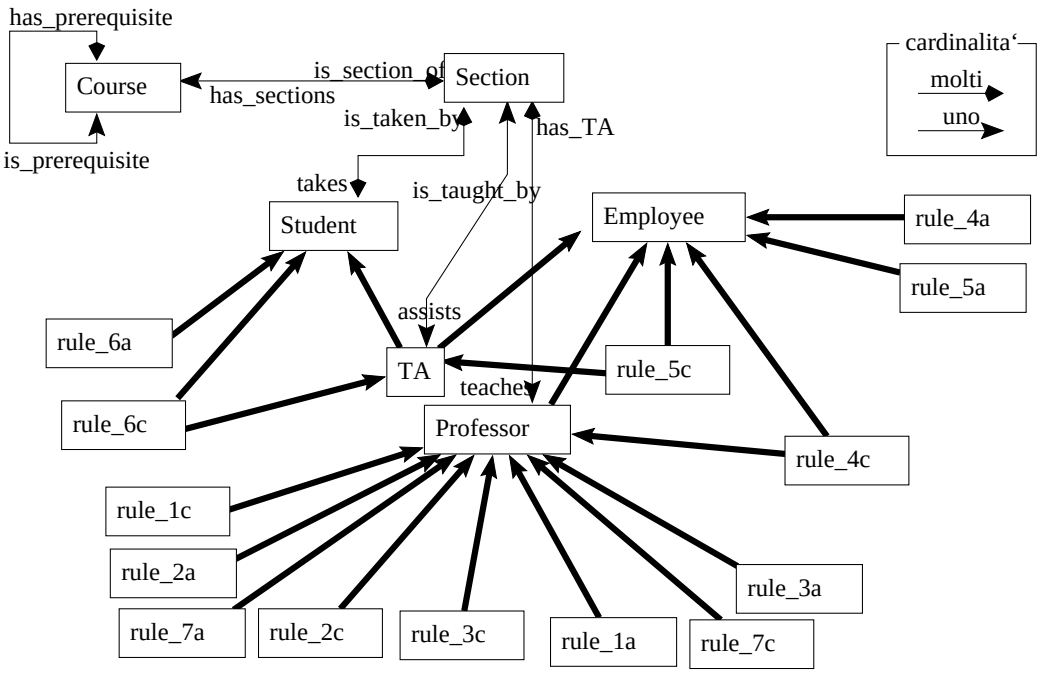


Figura 6.2: Schema dell'esempio con rules

```
antev rule_3a = Professor & ^ [ rank : vstring "Full" ] ;
consv rule_3c = Professor & ^ [ annual_salary : range 60000 +inf ] ;
antev rule_2a = Professor & ^ [ rank : vstring "Associate" ] ;
consv rule_2c = Professor & ^ [ annual_salary : range 40000 60000 ] ;
antev rule_1a = Professor & ^ [ rank : vstring "Research" ] ;
consv rule_1c = Professor & ^ [ annual_salary : range -inf 40000 ] .
```

Si noti che nella traduzione sono state perse tutte le informazioni relative ai metodi definiti nelle varie interfaces e che le relazioni sono state tradotte come coppie di attributi.

6.3 Il traduttore: struttura del programma

La sintassi di ODL è stata diffusa in formato *lex&yacc*[1], tale scelta è in accordo con la politica dell'ODMG di rendere ODL facile da *implementare*.

Attorno alla sintassi Lex&Yacc sono state aggiunte:

[1] **lex&yacc** sono due programmi, *lex* e *yacc*, molto usati in ambiente *UNIX* per realizzare analizzatori sintattici, (si veda appendice C)

1. Le *actions* della parte Yacc. Nelle actions vengono memorizzati tutti i concetti letti dall'ODL, vengono inoltre eseguiti alcuni (pochi) controlli semantici.

2. Routine di controllo della coerenza dei dati. Tra queste routine è presente l'algoritmo di traduzione delle rules.

3. Routine di stampa in formato OCDL.

La parte più complessa del traduttore è quella che si occupa di tradurre le regole.
Il programma è composto dai seguenti moduli

modulo principale (`c_main.c`) : Si ha l'inizializzazione delle variabili globali, e chiama in sequenza: *il parser, il controllore coerenza, la routine di stampa dei concetti in OCDL*

parser della sintassi ODL : È composto dal modulo *lex* (`odl.l`) e dal modulo *yacc* (`odl.y`). Svolge il controllo sintattico della sintassi ODL grazie alle routine generate con Lex&Yacc. Durante tale controllo "carica" le strutture dati.

controllore di coerenza (`c_coeren.c`) : Controlla l'univocità dei nomi de: `const`, `struct`, `typdef` ed `interfaces`. Controlla che esistano i campi inversi delle "relazioni" e calcola la traduzione OCDL delle rules.

routine di stampa dei concetti in OCDL (`c_procdl.c`) : Dalle strutture dati allocate dal parser estrae la forma OCDL.
Di seguito saranno descritte solo le parti *notevoli* del programma. Quelle ritenute più importanti, interessanti o particolarmente complesse.

6.3.1 Alcune strutture dati

Vediamo alcune tra le strutture dati usate per la memorizzazione delle informazioni da tradurre. Queste sono *riempite* durante la fase di controllo sintattico del testo ODL. Per questo motivo le strutture dati rispecchiano fortemente la grammatica[2] dell'ODL.
Una cosa da tenere a mente è che ODL è organizzato a moduli, OCDL è *piatto*. Le strutture dati impiegate permettono entrambi le rappresentazioni. In pratica, per i costrutti `struct`, `typedef`, `const` e `interfaces`, accanto ad una

[2]Intesa nel senso di insieme e struttura delle regole che descrivono la sintassi ODL.

rappresentazione ad albero dettata dall'organizzazione ODL è gestita anche una rappresentazione a lista che raggruppa tutti i concetti dei vari tipi.

In realtà la rappresentazione ad albero è inutile, ma c'è un motivo per cui è stata adottata. Il primo passo nel realizzare il traduttore è stato realizzare un programma in grado di memorizzare tutte le informazioni per poi ristamparne le linee essenziali. Tale programma, manteneva la struttura gerarchica delle informazioni. In seguito tale programma è diventato *Il traduttore* ed ha mantenuto le strutture dati tali da poter gestire i moduli. La gestione gerarchica dei moduli in seguito non è stata eliminata in quanto in futuro potrebbe essere utili per realizzare funzioni come controlli di visibilità tra i tipi definiti a vari livelli.

Di seguito è presentata a grandi linee la struttura con cui è memorizzato uno schema, è presente anche la descrizione di alcune strutture. Ritengo inutile la presentazione completa di tutte le strutture dati. Tale lavoro risulterebbe lungo, noioso e privo di utilità pratica. Di seguito sono descritte le strutture dati più significative, una volta capita la relazione tra sintassi e strutture dati, non è difficile capire lo scopo ed il significato di ogni campo di una qualunque struttura guardando il codice sorgente.

Rappresentazione dello schema Uno schema è visto come un *modulo*. La struttura che descrive un modulo è un record del tipo `s_definition_list`.

```
struct s_definition_list
   {
   char         type;  /* puo' valere:
                        * ---- tipi di definizioni riconosciute
                        *   i  interface, punta a t_interface_type
                        *   m  dichiarazione di moduli
                        *   t  dichiarazione di tipi
                        *   c  dichiarazione di costanti
                        *   r  dichiarazione di regole
                        *  ---- tipi non riconosciouti
                        *   e  dichiarazione di eccezioni
                        */
   union
      {
      struct s_interface_type   *i;
      struct           /* modulo ... */
         {
         char         *name;
                      /* nome modulo */
```

```
        struct       s_definition_list  *defi;
                     /* lista delle definizioni nel modulo */
        } m;
     struct s_type_list        *t;
     struct s_const_type       *c;
     struct s_rule_type        *r;
     } definition;
  struct s_definition_list *next;
  }
```

I costrutti che compongono il modulo sono memorizzate in una lista di record sempre del tipo `s_definition_list`. `s_definition_list` permette di rappresentare tutti i costrutti dichiarabili all'interno di un modulo. È composta da un flag `type` che indica il tipo del costrutto rappresentato, e da una `union` che associa ad ogni costrutto la relativa struttura dati.

La fase di estrazione delle informazioni in OCDL **non** fa uso di questo tipo di struttura (usata per una rappresentazione gerarchica) ma utilizza le liste con cui i principali costrutti sono *legati*.

Rappresentazione di un'*interface* Vediamo ad esempio come sono memorizzate le informazioni di un'*interface*.

```
struct s_interface_type
   {
   char         *name;
                /* nome interfaccia */
   char         fg_interf_view;
                /* indica se l'"interfaccia" e' un'interfaccia
                 * oppure una view
                 * vale:
                 *   'i'    interfaccia
                 *   'v'   view
                 */
   struct       s_iner_list  *iner;
                /* lista delle superclassi (eriditarieta')*/
   struct       s_prop_list  *prop;
                /* lista delle proprieta' */
   struct s_interface_type *next;
                /* lista globale delle interfaces */
   char         fg_used;
```

```
                /* flag usato per evirtare i cicli
                 * nella ricerca ricorsiva degli attributi
                 * nelle classi e nei suoi genitori
                 * puo' valere:
                 *   ' ' se non usato
                 *   '*' se usata
                 */
}
```

Il campo `s_interface_type` serve per la gestione delle `view`. Infatti una `view` è stata definita in modo da avere le stesse proprietà di un'interface. il campo `prop` punta alla lista delle proprietà[3]. Il campo `next` serve per la gestione della lista globale di tutte le `interface` dello schema.
Un record della lista delle proprietà di un'`interface` può descrivere:

- a attributo, punta ad una struttura s_attr_type
- r relazione, punta ad una struttura s_rela_type
- t tipo di dato (`typedef`)
- c costanti

Merita di essere citata la gestione delle `struct`. A differenza di altri costrutti come il `typedef`, le definizioni delle `struct` si trovano nella definizione di altri costrutti, ad esempio in `typedef`, nella dichiarazione di attributi o nella dichiarazione di un'altra `struct`. Questo rende un pò più complicata la gestione gerarchica delle definizioni delle struct. Per semplicità non ne gestito la memorizzazione gerarchica (che non è necessaria per realizzare il traduttore). Tutte le struct vengono memorizzate in una lista globale.

Legame strutture e grammatica Si può notare un forte legame tra i tocken definiti nella sintassi Lex&Yacc e le `struct` dichiarate. Questo è dovuto al modo di funzionare di Lex&Yacc (vedi appendice C a pagina 145). Yacc crea una procedura in grado di riconoscere `tocken`, per questo motivo risulta comodo memorizzare tutte le informazioni relative ad un tocken in una struttura dati dinamica allocata nel momento stesso in cui il `tocken` è riconosciuto. Così viene associata una `struct` ad ogni `tocken` del linguaggio, tale che la struttura sia in grado di memorizzare l'intero contenuto informativo del `tocken`.

[3]vedere le proprietà dell'ODL sezione 2.2.5 a pagina 14

6.4 Tipi `range`

Il tipo `range` permette di definire variabili intere con un dominio limitato. La dichiarazione di un attributo range ha la forma:

```
attribute range 40000, 100000 salary;
```

Significa che l'attributo `salary` può assumere solo valori interi compresi tra 4000 e 100000 inclusi.

Dove si può usare il tipo `range`: tale tipo fà parte dei `BaseTypeSpec` assieme a tipi come `long`, `float` o `boolean`.

Per questo motivo si può usare ove ODL prescrive l'utilizzo di tali tipi. Ad esempio è utilizzabile in ogni occasione in cui è utilizzabile il tipo `long`.

Non è possibile definire costanti di tipo `range`.

Nelle `rules` le variabili di tipo `range` sono compatibili con tutte le costanti di tipo `integer` (vedi 6.1).

Sintassi:

$$
\begin{array}{rcl}
< \text{RangeType} > & ::= & \textbf{range (} \ < \text{RangeSpecifier} > \ \textbf{)} \\
< \text{RangeSpecifier} > & ::= & < \text{IntegerValue} > \ \textbf{,} \ < \text{IntegerValue} > \ | \\
 & & < \textbf{IntegerValue} > \ \textbf{, + inf} \ | \\
 & & \textbf{- inf ,} \ < \textbf{IntegerValue} >
\end{array}
$$

6.5 Gestione dei btypes

In ODL è normale una dichiarazione del tipo:

```
typedef int newint;
```

che può essere vista come la definizione di un nuovo nome con cui dichiarare oggetti del tipo `int`.

La traduzione immediata ma "errata" è

```
type newint = integer ;
```

In questo caso dichiaro come tipo dato composto un tipo-base[4]. Questo in OCDL non ammesso. Per questo motivo ho dovuto distinguere tra due tipi di `typedef`:

[4] i tipi base OCDL sono: `integer`, `range`, `real`, `boolean` e `string`.

i typedef che *ridefiniscono* i tipi-base direttamente o indirettamente (*ridefinendo* un tipo che ridefinisce un tipo-base) e gli altri. I primi vengono dichiarati come `btype`, gli altri come `type`. Esiste una particolarità nella gestione dei primi, la dichiarazione di un `btype` in OCDL ammette la sintassi:

```
btype novo_nome = nome_tipo_base ;
```

ove `nome_tipo_base` è proprio un tipo base. Allora, una dichiarazione del tipo:

```
typedef  int      new_int1;
typedef  new_int1 new_int2;
typedef  new_int2 new_int3;
```

deve essere tradotta in:

```
btype new_int1 = integer;
btype new_int2 = integer;
btype new_int3 = integer;
```

per questo motivo, per i soli `btype` memorizzo la *descrizione originale*. `new_int3` risulterà `int` in quanto `new_int2` è `bype` e la sua descrizione originale è `int`.

6.6 Gestione delle `rule`

Per le `rules` si è cercata una sintassi che fosse:

- Tale da rendere intuitivo il significato delle regole
- Che permetta di esprimere *tutte* le condizioni esprimibili in OCDL
- Coerente con le sintassi proposte da ODMG. Per questo è stato scelto il costrutto `forall` utilizzato nell'OQL (Object Query Language) ODMG-93.

Esempio:

```
rule rule_7 forall X in Professor:
     ( X.rank  = "Full" )
   then
      exists X1 in X.teaches:
        ( X1.is_section_of in Course and
          X1.is_section_of.number = 1 )
```

La regola inizia con la parola chiave `rule`, seguita dalla dichiarazione della parte antecedente, poi la parola `then` e la parte conseguente. La parte antecedente deve per forza iniziare con il costrutto `forall` sugli elementi dell'estensione di una interface. Poiché OCDL non distingue tra il nome della classe e la sua estensione, indicando il nome di una interface in realtà ci si riferisce alla sua estensione.

6.6.1 Sintassi

< RuleDcl > ::= **rule** < Identifier >
< RuleAntecedente > **then** < RuleConseguente >

< RuleAntecedente > ::= < Forall > < Identifier > **in** < Identifier > **:**
< RuleBodylist >

< RuleConseguente > ::= < RuleBodyList >

< RuleBodyList > ::= **(** < RuleBodyList > **)** |
< RuleBody > |
< RuleBodylist > **and** < RuleBody > |
< RuleBodylist > **and (** < RuleBodyList > **)**

< RuleBodylist > ::= < DottedName > < RuleConstOp > < LiteralValue > |
< DottedName > < RuleConstOp > < RuleCast >
< LiteralValue > |
< DottedName > **in** < DottedName > |
< ForAll > < Identifier > **in** < DottedName > **:**
< RuleBodylist > |
exists < Identifier > **in** < DottedName > **:**
< RuleBodylist >

< RuleConstOp > ::= = | >= | <= | < | >

< RuleCast > ::= **(** < SimpleTypeSpec > **)**

< DottedName > ::= < Identifier > |
< Identifier > **.** < DottedName >

< ForAll > ::= **for all** | **forall**

6.6.2 manuale utente (inserimento delle `rule`)

Come scrivere una rule: vediamo alcune rule semplici ma particolarmente esplicative:

```
rule r1 forall X in Workers:
     ( X.salary > 10000000  )
   then
      X in AgiatePersons;
```

Si può leggere così : *"per ogni elemento, che indico con X, dell'estensione della classe* `Workers`*, se l'attributo* `salary` *di X ha valore maggiore di 10 milioni, allora l'elemento X deve far parte anche della classe* `AgiatePersons` *"*.

```
rule r1 forall X in Persons:
      ( X in AgiatePersons )
   then
      X.taxes = "High".
```

"*per ogni elemento, che indico con X, dell'estensione della classe* `Persons`*, se l'oggetto X appartiene anche all'estensione della classe* `AgiatePersons` *allora l'attributo* `taxes` *di X deve aver valore "*`high`*" "*.

Dot notation (nomi con punti)

All'interno di una condizione gli attributi e gli oggetti sono identificati mediante una notazione a *nomi con punti* (*dotted name*).
Con questa notazione è possibile identificare gli attributi di oggetti specificando il percorso che porta all'attributo.
Ad esempio, data la seguente dichiarazione:

```
interface Class1()
   {
   attribute string         c1a1;
   attribute range{1, 15}   c1a2;
   };

interface Class2()
   {
   attribute real           c2a1;
   attribute Class1         c2a2;
   };
```

```
interface Class3()
   {
   attribute long          c3a1;
   attribute Class2        c3a2;
   };
```

Dato un oggetto X di tipo `Class3` si ha che:
`X.c3a1`: è di tipo `long`, fa riferimento direttamente all'attributo definito nella classe `Class3`.
`X.c3a2`: è un oggetto della *classe* `Class2`.
`X.c3a2.c2a1`: è di tipo `real`, fa riferimento all'attributo definito nella classe `Class2`, questo è possibile in quanto l'attributo `c3a2` è un oggetto della classe `Class2`.
`X.c3a2.c2a2.c1a1`: è di tipo `string`, e fa riferimento all'attributo definito nella classe `Class1`.
Nelle rule sono possibili operazioni di confronto tra valori, Scrivere `X.c3a1 = 15` oppure `X.c3a2.c2a2.c1a1 = "pippo"` si intende confrontare il valore dell'attributo indicato attraverso il dotted name con il valore della costante.

Costrutti delle `rule`

I costrutti che possono apparire in una lista di condizioni sono:

- condizioni di appartenenza ad una classe

`identificatore_di_oggetto` **in** `nome_classe`

esprime la condizione di appartenenza di un oggetto all'estensione di una classe.
Esempi:
`X in AgiatePersons`
ove X individua un oggetto, la condizione è *vera* se X fà parte dell'estensione di `AgiatePersons`
`X1.is_section_of in Course`
ove X1 individua un oggetto, la condizione è verificata se X1 fà parte di `Course`.

- condizione sul tipo di un attributo

`identificatore_di_attributo` **in** `nome_tipo`

Esempio:
`X.age in range {18, 25}`

ove X individua un oggetto con attributo `age` di tipo `range` o intero[5], la condizione è verificata se `X.age` ha un valore compreso tra 18 e 25 inclusi.
Nelle rules, i ranges sono compatibili solo con costanti di tipo intero.

- condizioni sul valore di un attributo

`identificatore_di_attributo` **operatore** `costante`

La costante può essere un letterale oppure una `const` purché dello stesso tipo dell'attributo.
Gli operatori di disuguaglianza possono essere usati solo con attributi di tipo intero, questo perché le disuguaglianze vengono tradotte condizioni sul tipo `range`.
Esempi:

```
X.tass = "High"
X.age > 18
X.salary > lo_limit and X.salary < hi_limit
```

- condizioni su collezioni ed estensioni

forall `iteratore` **in** `collezione:  lista_di_condizioni`

esprime una condizione (**and**) su tutti gli oggetti di una *collezione*.
La condizione `forall` è *vera* quando tutti gli elementi della collezione soddisfano a tutte le condizioni della `lista_di_condizioni`.
Importante: tutti i *dotted name* della lista di condizioni associata al forall, cioè le variabili che appaiono tra le parentesi tonde del forall, **devono** iniziare con il nome dell'iteratore del forall. L'iteratore dev'essere quello dell'ultimo forall, ovvero del forall di livello inferiore. Non sono permessi confronti con variabili di forall di livelli superiori.
Esempio:

```
forall X1 in X.teaches:
   ( X1.is_section_of in Course and
     X1.is_section_of.number = 1 )
```

`X.teaches` dev'essere un tipo collezione, la condizione di `forall` è *vera* quando tutti gli oggetti in `X.teaches` hanno
`X1.is_section_of` **in** `Course` e
`X1.is_section_of.number = 1`.
Non sono permesse scritture del tipo

[5]per *interi* si intendono tutti quei tipi ODL che mappano nel tipo OCDL `integer`, vedi sezione 6.1.2 a pagina 57 .

```
forall X1 in X.teaches:
   (X1.is_section_of.number =  X.annual_salary )
```

oppure

```
forall X1 in X.teaches:
   (X1.is_section_of in Course and X.annual_salary = 4000 )
```

Caso particolare: il `forall` con cui inizia una regola *antecedente* esprime un condizione sui singoli oggetti dell'estensione di una interface. In questo caso l'iteratore individua degli oggetti. Il costrutto `forall` cambia di significato, non è una condizione di `and` tra le condizioni dei singoli oggetti dell'estensione ma indica di valutare la lista di condizioni del forall per ogni singolo oggetto. Se il singolo oggetto verifica le condizioni allora la *regola* impone che siano verificate anche le condizioni della condizione *conseguente*.

exists `iteratore` **in** `collezione:  lista_di_condizioni`

simile al `forall` esprime una condizione (**or**) su tutti gli oggetti di una *collezione*.
La condizione `exists` è *vera* esiste almeno un elemento della collezione che soddisfa a tutte le condizioni della `lista_di_condizioni`
Esempio:

```
exists X1 in X.teaches:
  ( X1.is_section_of in Course and
    X1.is_section_of.number = 1 )
```

`X.teaches` dev'essere un tipo collezione, la condizione di `exists` è *vera* quando almeno un oggetto in `X.teaches` ha
`X1.is_section_of` **in** `Course`
e
`X1.is_section_of.number = 1`.

6.6.3 Algoritmo di traduzione delle `rule`

In questa sezione presentiamo l'algoritmo più *complesso* dell'intero *traduttore*. Tale algoritmo è realizzato nel modulo `c_coeren.c` e svolge contemporaneamente la traduzione ed un primo controllo semantico di ogni singola regola.
Controlli di coerenza realizzati:

Strutture dati utilizzate Vediamo in che modo le informazioni relative alle rules sono memorizzate, cioè ovvero la rappresentazione interna delle informazioni delle rules lette in fase di parsing dal formato ODL-esteso.

```
struct s_rule_type
{
   char   *name;         /* nome della rule */
   char   *ocdlante;     /* output OCDL della relazione */
   char   *ocdlcons;     /* questo campo \'e "riempito" */
                         /* in fase di validazione ed usato */
                         /* in fase di stampa dell'OCDL */
   struct s_rule_body_list *ante;
   struct s_rule_body_list *cons;
   /* lista globale di tutte le regole */
   struct s_rule_type      *next;
}
```

La struct `s_rule_type` serve per memorizzare le varie rules. Tutte le rules dichiarate in uno schema sono unite in una lista. Come si vede, una `rule` è descritta da una parte antecedente ed una conseguente. È previsto che in questa struttura sia memorizzata la traduzione della rule in formato OCDL (nei campi `ocdlante` e `ocdlcons`). Le due condizioni antecedenti e conseguenti sono descritte come una lista di condizioni, anche in questo caso la struttura segue la grammatica che descrive la sintassi delle regole.

Di seguito è riportata la struttura dati che permette di memorizzare una lista di condizioni. Si può notare che un record di tale lista è in grado di descrivere uno qualunque dei costrutti base che compongono una condizione di una regola.

```
struct s_rule_body_list
{
   char type;
    /* flag che pu\'o valere:
     *   'c'  dichiarazione di costante
     *   'i'  dichiarazione di tipo
     *   'f'  la regola \'e un forall o un'exists
     */
   char fg_ok;
    /* variabile di comodo per sapere se una data condizione
     * e' gia' stata considerata
     * puo' valere:
     *   ' '  condizione NON ancora considerata
     *   '*'  condizione gia' considerata
```

```
  */
 char fg_ok1;
  /* variabile di comodo
   * usata SOLO per la body_list del primo livello
   * della parte conseguente di una rule
   * serve per sapere se una data condizione
   * e' gia' stata considerata
   * infatti nel caso particolare del primo livello
   * di una condizione conseguente
   * si hanno due tipi di condzioni
   *  1. quelle che coinvolgono X come iteratore
   *     es:   X.int_value = 10
   *     queste devono essere racchiuse tra par. quadre
   *  2. quelle che coinvolgono X in quanto indica
   *     il membro dell'estensione
   *     es:   X in TManager
   *     queste devono essere messe in and con il tipo classe
   *     es:  Manager & TManager ...
   * pu\'o valere:
   *   ' '  condizione NON ancora considerata
   *        questo \'e il valore di default
   *   '*'  condizione gi\'a considerata
   */
 char *dottedname;           /* nome variabile interessata */
 union
 {
       struct
       {     /* in questo caso
              * dottedname e' la variabile da mettere
              * in relazione con la costante
              */
             char *operator;
             char *cast;        /* NULL se manca il cast */
             char *value;
       } c;
       struct
       {     /* in questo caso
              * dottedname e' la variabile su cui imporre
              * il tipo
              */
             char *type;         /* identif. tipo  */
       } i;
       struct
```

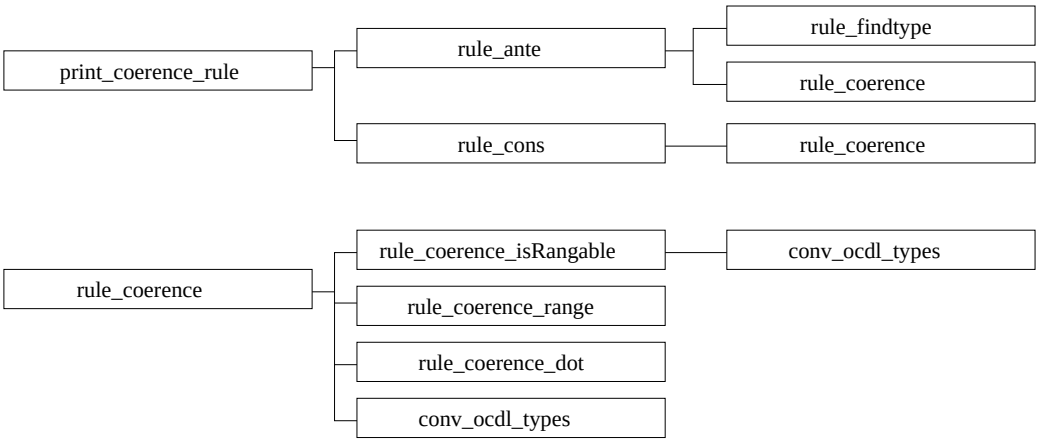


Figura 6.3: Legami tra le funzioni di gestione delle `rule`

```
        {     /* in questo caso
               * dottedname \‘e la lista su cui iterare
               */
              char fg_forall;    /*  puo valere:
                                  *   ’f’  forall
                                  *   ’e’  exists
                                  *        significa che il
                                  *        tipo \‘e un’exists
                                  * questo flag \‘e stato
                                  * introdotto in quanto i
                                  * tipi EXISTS e FORALL
                                  * hanno quasi la stessa traduzione
                                  * in comune
                                  */
              char *iterator;    /* nome iteratore */
              struct s_rule_body_list *body;
        } f;
   } r;
   struct s_rule_body_list *next;
}
```

Descrizione delle routine

Dal diagramma di figura 6.6.3 è possibile vedere come è strutturato l’algoritmo di controllo-semantico e traduzione delle `rule`.

Di seguito sono presentate le funzioni che realizzano l’algoritmo. Gli esempi fanno riferimento all’esempio a pagina 59.

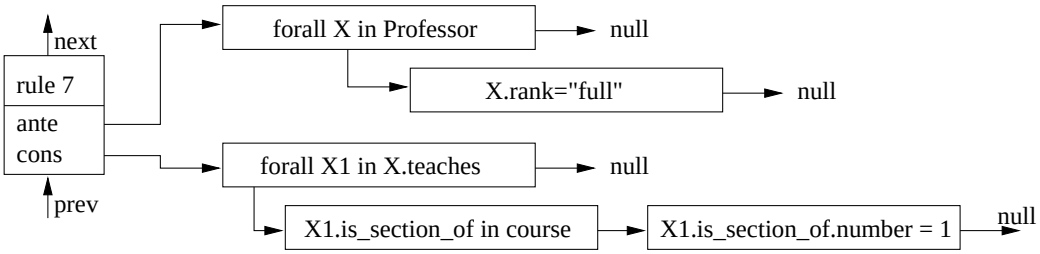


Figura 6.4: Struttura di memorizzazione della `rule_7`

memorizzazione di una `rule` In figura 6.4 è riportata la struttura dati con cui, in fase di *parsing* vengono memorizzate le informazioni relative alla seguente regola:

```
rule rule_7 forall X in Professor:
     ( X.rank  = "Full" )
   then
      exists X1 in X.teaches:
        ( X1.is_section_of in Course and
          X1.is_section_of.number = 1 )
```

la cui traduzione è:

```
antev rule_7a = Professor & ^ [ rank : vstring "Full" ] ;
consv rule_7c = Professor &
                ^ [ teaches :
                   !{ ^ [ is_section_of : ^ [ number : vstring "1" ] ,
                          is_section_of : Course
                        ]
                    } ] ;
```

`print_coerence_rule` La funzione `print_coerence_rule` effettua il **controllo di coerenza** della `rule`.

input

La lista di tutte le rules, cioè una lista concatenata di record definiti dalla struct `s_rule_type`.

output

La traduzione OCDL di tutte le rules. Per ogni `rule` calcola la traduzione OCDL della parte antecedente e conseguente e le memorizza in due stringhe. Se una rule è non coerente pone il traduttore in stato di errore e continua la verifica delle incoerenze.

cosa fà

richiama le funzioni `rule_ante` e `rule_cons` passando come argomento rispettivamente la lista delle condizioni della parte antecedenete e la lista delle condizioni della parte conseguente.

rule_ante La funzione `rule_ante` effettua un controllo semantico sui tipi e la traduzione in OCDL della parte antecedente di una `rule`.

input

una lista di condizioni, cioè una lista di record definiti dalla struct `s_rule_body_list`.

output

Una stringa che contiene la traduzione OCDL della condizione.

esempio

data la descrizione in forma di strutture di

```
forall X in Professor ( X.rank  = "Full" )
```

ne restituisce la seguente traduzione

```
Professor & ^ [ rank : vstring "Full" ]
```

cosa fà

Controlla che la regola antecedente inizi cono un `forall` sull'estensione di una interface. Per fare ciò chiama la funzione `rule_findtype`. Aggiunge l'iteratore del `forall` nella lista degli iteratori. Chiama la funzione `rule_coerence` per la gestione della lista delle condizioni del forall.

Gestione degli iteratori durante il calcolo della traduzione OCDL di una regola tengo traccia degli iteratori definiti dai vari `forall`. Per ogni iteratore memorizzo il tipo dell'elemento a cui punta. Poiché le variabili sono espresse con la notazione puntata (pag. 72), il *tipo* dell'iteratore è un'informazione necessaria per risalire al *tipo* di una variabile. Il *tipo* dell'iteratore puòessere tipo oggetto e viene memorizzato con il prefisso `interface` seguito dal nome dell'interface, un tipo struct e viene memorizzato con il prefisso `struct`, oppure il tipo dell'iteratore può essere un tipo qualunque dei tipi supportati da ODL, in tal caso il nome del tipo viene memorizzato senza prefissi. La lista degli iteratori può essere vista come una sorta di `stack` il cui ultimo elemento è l'iteratore a cui fanno riferimento le variabili citate nella lista di condizioni in fase di elaborazione. Quando è stato *progettato* il traduttore si è pensato di prevedere l'estensione dell'uso nelle *condizioni* ad un certo livello di forall degli iteratori definiti nei livelli di forall superiori. È per questo motivo che la gestione degli iteratori ha assunto una propria dignità con struttura dati e funzioni dedicate all'inserimento e alla cancellazione.

rule_cons La funzione `rule_cons` effettua un controllo semantico sui tipi e la traduzione in OCDL della parte conseguente di una `rule`.

input

una lista di condizioni, cioè una lista di record definiti dalla struct `s_rule_body_list`.

output

Una stringa che contiene la traduzione OCDL della condizione.

esempio

data la descrizione della parte conseguente

```
exists X1 in X.teaches:
  ( X1.is_section_of in Course  and  X1.is_section_of.number = 1 )
```

ne restituisce la traduzione

```
Professor & ^ [ teaches :
                !{ ^ [ is_section_of : ^ [ number : vstring "1" ] ,
                       is_section_of : Course
                     ]
                 }
               ]
```

cosa fà La lista conseguente è associata all'iteratore definito nel forall principale della relazione antecedente. Tale iteratore principale non fà riferimento agli elementi di una collezione bensì ai singoli oggetti dell'estensione di una interface. Per questo motivo la routine tratta in modo particolare gli elementi della lista passata come argomento. Nella prima parte della funzione si trova la gestione condizioni di "in" sull'iteratore. in una rule del tipo

```
forall X in Storage
```

le condizioni conseguenti del tipo

```
X in SStorage
```

vengono mappate come ereditarietà

```
Storage & SStorage ...
```

Nella prima parte la funzione chiama la funzione `rule_coerence` per la gestione delle rimanenti condizioni.

rule_coerence La funzione `rule_coerence` effettua il controllo e la traduzione in OCDL di una qualunque lista di *condizioni*.

input

Una lista di condizioni, cioè una lista di record definiti dalla struct `s_rule_body_list`.

output

Una stringa in cui le singole condizioni sono tradotte in OCDL e separate tra loro da virgole.

esempio

data la descrizione di una lista di condizioni

```
exists X1 in X.teaches: (
X1.is_section_of in Course  and  X1.is_section_of.number = 1 )
```

ne restituisce la traduzione

```
^ [ teaches : !{ ^ [ is_section_of : ^ [ number : vstring "1" ] ,
                   is_section_of : Course
                   ] }
  ]
```

cosa fà

Questa funzione effettua la gestione ricorsiva del corpo delle rules cioè di una lista di condizioni, tenta di tradurre il corpo delle rules.
Per ogni condizione della lista non ancora esaminata, costruisce una stringa `rv1` che è la traduzione della singola condizione questa viene poi unita alla stringa `rv` che è la traduzione ocdl della lista di condizioni.
Ci sono due motivi per cui una condizione può essere già stata esaminata: (1) è una condizione che proviene dalla parte conseguente di una rule e tale condizione riguardava l'ereditarietà, (2) fà parte di una disequazione già tradotta in range.
Per ogni condizione della lista:
controlla che ogni dotted-name della condizione inizi con l'iteratore del `forall` o `exists` a cui appartiene la lista di condizioni costruisce la parte centrale "core" della condizione:
se si tratta di una condizione sul tipo scrive solo il tipo in dev'essere mappato il dotted name.
se si tratta di una condizione sulle costanti verifica se è una condizione trasformabile in range (chiama la funzione `rule_coerence_isRangable`)
se è "rangable" la trasformo in range, nel fare questo trasformo anche le altre condizioni che possono far parte del medesimo range (chiama la funzione

`rule_coerence_range`).
se non è una condizione trasformabile in range controlla la coerenza tra il tipo implicito del dotted-name e quello della costante che appare sulla destra.
se si tratta di una condizione di `forall` o `exists`: verifica che il dotted-name sia un tipo collezione, discrimina tra i tipi di collezione tra `set` e `sequence`, verifica se il dotted-name è una collezione di oggetti oppure struct, a questo punto la funzione `rule_coerence` chiama ricorsivamente se stessa per costruire la lista di condizioni associata al forall o l'exsist, ricopre la condizione di `[ ]` oppure `^ [ ]` rispettivamente per condizione su struct o oggetti, ricopre la condizione di `{ }` o `! { }` o `< >` o `! < >` in quanto trattasi di forall o exists su collezione ordinata o meno.
A questo punto il "core" della condizione è stato costruito, occorre costruire la parte rimanente della condizione. La parte rimanente della condizione tiene conto di tutte quelle informazioni che derivano dal dotted name.
Per fare ciò chiama la funzione `rule_coerence_dot` che "ricopre" il core appena costruito.

rule_coerence_dot `rule_coerence_dot` è una funzione ricorsiva chiamata da `rule_coerence`, ricopre il "core" calcolato da quest'ultima con l'equivalente OCDL della notazione dotted.

input

Il una stringa contenente il *core* da avvolgere, il *dotted-name* parziale da tradurre, il *dotted-name* completo di cui il *dotted-name* parziale è parte, la lista degli iteratori usati.

output

Il core ricoperto con l'equivalente del dotted-name parziale.

esempio

nella routine rule_coerence gestisco i vari tipi di condizioni però NON gestisco le strutture nel senso che, data una condizione, rule_coerence si preoccupa di generare la corretta parte "centrale", "destra" della condizione OCDL
es: `X.aa.bb = 5`
rule_coerence genera "`vinteger 5`" che è il "core" ma lascia il compito di generare la parte "sinistra", quella "esterna" a questa routine
`rule_coerence_dot` deve generare:
se aa è di tipo struct : `aa : [ bb : "core" ]`
se aa è di tipo interface: `aa : ^ [ bb : "core" ]`

cosa fà

funzionamento dell'algoritmo ricorsivo:
se `dotted` ha lunghezza nulla restituisce solo `core`,

se `dotted` non contiene punti restituisce: `dotted : core`
se `dotted` contiene chiama la funzione rule_findtype per trovare il tipo del dotted parziale e restituisce:
se il dotted parziale si riferisce ad una struct:
`first_dot : [ "core1" ]`
se il dotted parziale è di tipo interface `first_dot : ^ [ "core1" ]`
ove `''core1''` è dato dalla chiamata ricorsiva a questa funzione `rule_coerence_dot`

rule_findtype Cerca il tipo di un dotted-name.

input

Una stringa contenente un dotted-name, la lista degli iteratori.

output

il tipo ODL della variabile specificata dal dotted-name.

esempio

data la seguente dichiarazione:

```
interface Material ()
   {
   attribute string                name;
   attribute int                   risk;
   attribute set<string>           feature;
   };
typedef long t_long;
interface  AAI ()
   {
   attribute string                code;
   attribute string                class;
   attribute struct s_aa1
      {
      set < int >          aa2;
      set < list <int> >   aa3;
      t_long               aa4;
      } aa1;
   attribute Material     aa5;
   attribute Set < Material > aa6;
   };
```

ecco cosa restituisce questa funzione:

dotted-name	**Output**
AAI	interface AAI
AAI.code	string
AAI.aa1	struct s_aa1
AAI.aa1.aa2	set < int >
AAI.aa1.aa3	set < list < int > >
AAI.aa1.aa3	long (i tyedef vengono esplosi)
AAI.aa5.risk	iint
AAI.aa5	interface Material
AAI.aa6	set < Material >

cosa fà

descrizione dell'algoritmo: estrae il primo nome dal dotted name, questo può essere (1) un iteratore presente nell'`iterator-list` oppure (2) il nome di una `interface`. Estrae il successivo nome dal *dotted-name* questa dev'essere una variabile definita nella struttura dati del *tipo* del nome precedente. Il tipo precedente può dunque essere:

tipo precedente	notazione
struct	`struct <nome struct>`
interface	`interface <nome interface>`
view	`view <nome view>`

se il tipo precedente è un typedef allora lo "esplodo" fino ad arrivare ad una struct, ad un'interface o una view.

Un dotted-name è sbagliato se:

- il primo nome del dotted name non è il nome di un'interface o di un iteratore.
- un qualunque nome *intermedio*, ad esempio `bb` o `cc` di `aa.bb.cc.dd`, non e' (direttamente o tramite un typedef) di tipo `struct`, `interface` o `view`.
- il nome successivo non è attributo del tipo del nome precedente, es: in `aa.bb.cc.dd`, supponiamo che `bb` sia di tipo interface `i_bb` (un oggetto), allora "`cc`" deve essere nome di un attributo di tale interface.

Capitolo 7

L'Applet Java

Origini L'Applet Java di nome `scvisual` presentata in questa sezione è uno strumento per la rappresentazione grafica di uno schema di basi di dati ad oggetti. Nell'interfacciare il prototipo ad Internet si è cercato di mantenere la rappresentazione grafica degli schemi che è il mezzo più intuitivo e veloce per catturare il significato dello schema.

Così è nato questo strumento, progettato per essere inserito in una pagina di un Browser, per essere flessibile e semplice da utilizzare.

In figura 7.1 si vede come l'Applet si presenta all'utente.

L'area occupata dall'applet è divisa in due parti. Nella parte superiore è visualizzato lo schema, in quella inferiore, vi sono i pulsanti di comando che permettono di eseguire l'algoritmo di ordinamento del grafo o scegliere quali oggetti visualizzare.

cosa è in grado di visualizzare :

`scvisual` è in grado di visualizzare:

classi suddivise in `class` ed in `rules`. Sono visualizzate come rettangoli. Per l'applet la differenza tra `class` e `rule` è che le class sono sempre visibili mentre le rule possono essere nascoste facendo *click* sull'apposito `Checkbox`.

relazioni di *ereditarietà* (relazioni ISA). Sono visualizzate come spesse freccie grigie che uniscono due classi.

relazioni aggregazione Sono visualizzate con sottili freccie rosse che uniscono due classi.

diverse versioni dello stesso grafo È possibile passare all'applet la descrizione

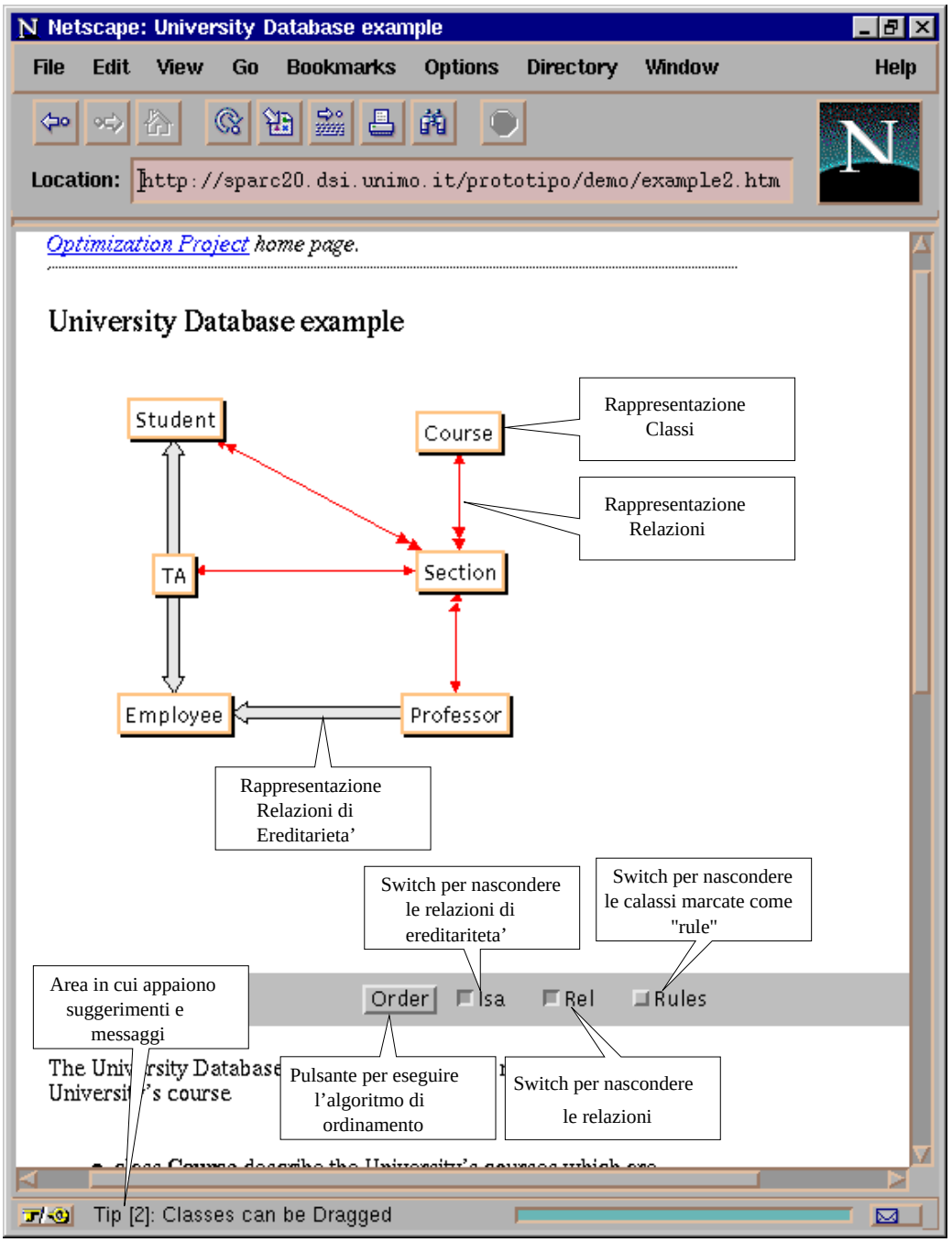


Figura 7.1: Come appare l'applet Java `scvisual`

di più schemi contemporaneamente, in tal caso tra i pulsanti di comando appare un pulsante di *Choice* che permette di scegliere quale schema visualizzare.

L'applet legge la descrizione dei costrutti da visualizzare da un file ASCII che dev'essere in un formato particolare, creato appositamente, detto *visual form*.

Una volta visualizzato un grafo è possibile generare la descrizione del grafo visualizzato in formato `XFig 3.1`.

algoritmi particolari Per la realizzazione di questo strumento si sono dovuti risolvere alcuni problemi specifici, in questo paragrafo sono elencati i problemi e gli algoritmi progettati per risolverli. Le sezioni finali di questo capitolo sono dedicate all'analisi dettagliata di questi stessi algoritmi.

Algoritmo di ordinamento

Si tratta di un algoritmo euristico che ha il compito di trovare una disposizione "ordinata" delle classi sullo schermo.

Algoritmo di creazione relazioni ISA fittizie

Serve a risolvere il seguente problema:
uno schema può contenere delle *classi*, che ereditano da *rule*, che a loro volta ereditano da altre classi. In questo caso le prime *classi* "ereditano" indirettamente dalle seconde. Se in fase di visualizzazione nascondo le *rule*, corro il rischio di perdere tali relazioni di ereditarietà.

Esempio (in figura 7.2):
date le classi C1, C2 e la rule R1, e date le seguenti relazioni di ereditarietà: `C2 isa R1`, `R1 isa C1`, allora è vero che `C2 isa C1`.

Questo algoritmo calcola tutte le relazioni "fittizie" del tipo `C2 isa C1` da mostrare quando le *rules* sono da nascondere.

Algoritmo di creazione relazioni di aggregazione fittizie

È un problema simile a quello precedente. Supponiamo che sia dato lo schema di figura 7.2 in cui vi sono le seguenti relazioni: `C2 isa R1`, `R1 isa C1`, `C3 rel R1`. In questo caso C2 eredita da R1 la relazione che ha con C3. Se però non visualizzo le rules corro il rischio di perdere questa informazione. Questo algoritmo calcola tutte le relazioni *fittizie* del tipo `C3 rel C2` da mostrare quando le *rules* sono da nascondere.

Limiti dello strumento Il numero massimo di oggetti che l'applet è in grado di gestire a volte dipende dalle risorse della macchina su cui viene fatta girare. Di seguito sono riportati i limiti che dipendono dalle scelte fatte sul dimensionamento di array di oggetti.

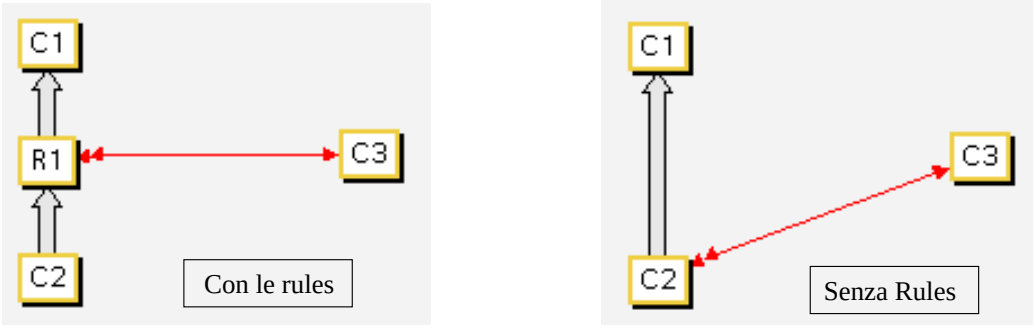


Figura 7.2: esempio di applicazione degli algoritmi di creazione delle relazioni fittizie

- numero massimo di classi tra *class* e *rules* per ogni *livello*: 250
- numero massimo di relazioni di aggregazione per ogni *livello*, comprese le relazioni fittizie: 250
- numero massimo di relazioni di ereditarietà per ogni *livello*, comprese le relazioni fittizie: 250

7.1 Manuale Utente (Uso dell'Applet)

In questo manuale utente è descritto quali sono i costrutti del modello ad oggetti che possono essere visualizzati dall'applet, come è possibile interagire con l'applet, e come inserire l'applet in una pagina HTML.

7.1.1 Rappresentazione grafica dei costrutti

L'applet `scvisual` tratta solo alcuni dei costrutti definiti nell'ambito della modellazione ad oggetti:

classi Sono rappresentate con i rettangoli al cui interno viene scritto il nome della classe. Un rettangolo può contenere anche più righe di testo, ogni riga è un nome di classe, il fatto che appaiano in un unico rettangolo significa che queste classi sono tra loro equivalenti, cioè hanno i medesimi attributi e lo stesso significato semantico.

relazioni di ereditarietà (ISA) Sono rappresentate da **spesse** freccie. Se la freccia collega la classe A alla classe B con la freccia rivolta verso B, significa che

esiste una relazione di ereditarietà tra A e B, in particolare A isa B, cioè A *eredita* da B.

relazioni di aggregazione Sono rappresentate freccie sottili. Sono visualizzabili diversi tipi di relazioni, una relazione da A a B può avere più significati a seconda delle freccie presenti alle estremità. Sono rappresentabili:

(1) Attributi che mappano in classi: sono freccie *unidirezionali* da A a B, se la freccia ha una sola punta significa che nella classe A c'è un attributo fà riferimento ad un oggetto della classe B, se la freccia ha due punte significa che nella classe A c'è un attributo *collezione* che fà riferimento ad oggetti della classe B.

(2) Relazioni *vere e proprie*, sono freccie *bidirezionali*. Le estremità con una sola freccia indicano che la relazione ha cardinalità "uno", le estremità con freccie a due punte indicano cardinalità "molti". Data una relazione tra A e B, questa può essere solo dei seguenti quattro i tipi

tipo	freccia
uno a uno	A <---> B
uno a molti	A <--->> B
molti a uno	A <<---> B
molti a molti	A <<--->> B

7.1.2 Funzionalità in fase di visualizzazione

Dal browser è possibile interagire con l'applet in vari modi:

- Ogni volta che si fà click sullo sfondo dell'applet viene visualizzato un *Tip*, cioè un suggerimento che illustra brevemente le funzionalità dell'applet.

- Si possono spostare i rettangoli che corrispondono alle classi trascinandoli con il mouse.

- Si possono visualizzare gli attributi di una classe facendo un doppio click sul rettangolo della classe stessa. Gli attributi vengono visualizzati in una nuova finestra.

- Si può visualizzare il nome e le caratteristiche di una relazione facendo doppio click in prossimità della freccia che la rappresenta. La descrizione appare nella stessa finestra in cui sono mostrati gli attributi di una classe. Quando si fà doppio click in punti in cui passano più di una relazione allora, vengono visualizzate le proprietà di tutte le relazioni che passano per tale punto.

- Facendo un triplo click sullo sfondo, sulla `java console` viene generata la descrizione del grafo sullo schermo in formato Xfig 3.1.

Naturalmente, il modo migliore per avere un'idea precisa delle funzionalità è l'interazione diretta all'indirizzo Internet:
`http://sparc20.dsi.unimo.it`

Livelli La gestione dei livelli è stata introdotta per visualizzare gli schemi che si ottengono durante i vari passi dell'ottimizzazione. Infatti, gli algoritmi di ottimizzazione sono iterativi e procedono per passi, con l'introduzione dei livelli è possibile seguire graficamente i vari passi dell'ottimizzazione. Per questo motivo si è pensato di permettere la descrizione di diversi schemi all'interno di un unico file di descrizione. Ogni "schema" è associato ad un numero chiamato *livello*. Se il file di descrizione contiene più di uno schema, appare un pulsante di controllo mediante il quale è possibile scegliere quale *livello* mostrare.

7.1.3 Inserimento di `scvisual` in una pagina HTML

Vediamo un esempio tratto dalla pagina di presentazione di uno degli schemi di esempio presenti nel sito Web del prototipo.

```
<APPLET CODEBASE=/java CODE=scvisual  WIDTH=600 HEIGHT=400
 alt="Your browser understand but can't execute Java Applets" >
   <param name=source_url
value="http://sparc20.dsi.unimo.it/prototipo/esempi/primo/schema_1.odl">
   <table border=1 align=center>
      <TR> <TD>
      <font size=+2 color=#FFFF00>
      This space is intended for the Schema drawing java applet.
      If you see this message then your browser
      doesn't support Java Applets.
      </font>
   </table>
</APPLET>
```

Prima di descrivere in modo particolareggiato l'esempio, notiamo che l'unica particolare caratteristica di questa applet è il parametro `source_url` che serve a passare all'applet i dati da visualizzare. La seguente scrittura:

```
<param name=source_url
value="http://sparc20.dsi.unimo.it/prototipo/esempi/primo/schema_1.odl">
```

assegna al parametro di nome `source_url` il nome dell'URL:
`http://sparc20.dsi.unimo.it/prototipo/esempi/primo/schema_1.odl`
tale URL serve per indicare dove si trova la descrizione dello schema da visualizzare. L'applet legge il file corrispondente al nome dell'URL con postfisso `.vf`. In pratica l'applet invocata dall'esempio visualizza ciò che trova descritto in formato *visual* nell'URL di indirizzo:
`http://sparc20.dsi.unimo.it/prototipo/esempi/primo/schema_1.odl.vf`
Seguendo l'esempio, vediamo ora come inserire una generica applet in una pagina HTML:
I "Tags" `<applet>` e `</applet>` indicano il punto in cui inserire l'applet, indicano rispettivamente l'inizio e la fine del testo della pagina HTML che descrive quale applet lanciare, i parametri da passare ed altre caratteristiche che vedremo di seguito.
La riga
`<APPLET CODEBASE=/java CODE=scvisual  WIDTH=600 HEIGHT=400...>`
informa il browser che:
(CODE) l'applet da eseguire ha nome `scvisual`,
(CODEBASE) l'applet si trova nella directory`/java` del browser attuale,
(WIDTH e HEIGHT) di riservare un rettangolo di 600x400 pixel per la visualizzazione dell'applet stessa,
(ALT) nel caso in cui il browser utilizzato riconosca le Applet Java ma non sia in grado (anche solo temporaneamente) di eseguire l'applet, il browser deve visualizzare la scritta:
" Your browser understand but can't execute Java Applets "
Un "tag" specifico delle applet è `<PARAM ...>`: indica i parametri da passare all'applet. Il funzionamento è simile alle variabili di ambiente di UNIX o del DOS, il parametro è identificato da un nome (NAME) ed ha un certo valore (VALUE). In Java, tra i metodi della classe a̲pplet, esiste la funzione che, dato il nome di un parametro, ne estrae il valore.
Nel caso in cui il Browser che legge la pagina HTML non riconosca le Applet Java, il browser applica il default per le direttive HTML, cioè le ignora. Il browser allora visualizza tutto ciò che non è "tag" Per questo motivo è buona norma inserire tra i tags
`<APPLET>` ed `</APPLET>`
il messaggio da visualizzare al posto dell'applet nel caso in cui questa non possa essere eseguita. Nell'esempio appare il messaggio:
"This space is intended for the Schema drawing java applet. If you see this message then your browser doesn't support Java Applets."
Le altre direttive quali `<table border=1 align=center>` o

`<font size=+2 color=#FFFF00>` servono per formattare il testo in modo che appaia in grande, centrato, bordato ed in un colore giallo intenso.

7.2 File di descrizione del grafo (Visual Form)

In questa sezione è descritta la sintassi del file di testo usato per passare i dati all'applet.
La sintassi è piuttosto semplice, consiste in un insieme di direttive simili ai "tags" dell'HTML, vi è sempre una direttiva di inizio di un blocco e la corrispondente direttiva che chiude il blocco.

7.2.1 Un esempio

L'esempio che segue mostra la descrizione dello schema "università" già presentato in sezione 6.2 che è riportato anche in figura 7.1:

```
<LEVEL 0 LAST>
<CLASSES_TO_DISPLAY>
<CLASS> Student
        ATTRIBUTE attribute string name;
        ATTRIBUTE attribute integer student_id;
        ATTRIBUTE attribute struct Address dorm_address;
</CLASS>
<CLASS> TA
</CLASS>
<CLASS> Professor
        ATTRIBUTE attribute string rank;
</CLASS>
<CLASS> Employee
        ATTRIBUTE attribute string name;
        ATTRIBUTE attribute short id;
        ATTRIBUTE attribute unsigned short annual_salary;
</CLASS>
<CLASS> Section
        ATTRIBUTE attribute string number;
</CLASS>
<CLASS> Course
        ATTRIBUTE attribute string name;
        ATTRIBUTE attribute unsigned short number;
</CLASS>
<CLASS> rule_7a RULE
```

```
        ATTRIBUTE antev rule_7a = Professor &
  ^ [ rank : vstring "Full" ]
</CLASS>
<CLASS> rule_7c RULE
        ATTRIBUTE consv rule_7c =
                Professor &
                ^ [ teaches :
                    { ^ [ is_section_of : ^ [ number : vinteger 1 ] ,
                          is_section_of : Course ] } ]
</CLASS>
<CLASS> rule_6a RULE
        ATTRIBUTE antev rule_6a = Student &
  ^ [ student_id : range -inf 2000 ]
</CLASS>
<CLASS> rule_6c RULE
        ATTRIBUTE consv rule_6c = Student & TA
</CLASS>
<CLASS> rule_5a RULE
        ATTRIBUTE antev rule_5a =
                Employee & ^ [ annual_salary : range -inf 20000 ]
</CLASS>
<CLASS> rule_5c RULE
        ATTRIBUTE consv rule_5c = Employee & TA
</CLASS>
<CLASS> rule_4a RULE
        ATTRIBUTE antev rule_4a =
                Employee & ^ [ annual_salary : range 30000 +inf ]
</CLASS>
<CLASS> rule_4c RULE
        ATTRIBUTE consv rule_4c = Employee & Professor
</CLASS>
<CLASS> rule_3a RULE
        ATTRIBUTE antev rule_3a = Professor &
  ^ [ rank : vstring "Full" ]
</CLASS>
<CLASS> rule_3c RULE
        ATTRIBUTE consv rule_3c =
                Professor & ^ [ annual_salary : range 60000 +inf ]
</CLASS>
<CLASS> rule_2a RULE
        ATTRIBUTE antev rule_2a =
                Professor & ^ [ rank : vstring "Associate" ]
</CLASS>
```

```
<CLASS> rule_2c RULE
        ATTRIBUTE consv rule_2c =
                Professor & ^ [ annual_salary : range 40000 60000 ]
</CLASS>
<CLASS> rule_1a RULE
        ATTRIBUTE antev rule_1a =
                Professor & ^ [ rank : vstring "Research" ]
</CLASS>
<CLASS> rule_1c RULE
        ATTRIBUTE consv rule_1c =
                Professor & ^ [ annual_salary : range -inf 40000 ]
</CLASS>
</CLASSES_TO_DISPLAY>
<ISA>
        TA ISA Employee
        TA ISA Student
        Professor ISA Employee
        rule_7a ISA Professor
        rule_7c ISA Professor
        rule_6a ISA Student
        rule_6c ISA Student
        rule_6c ISA TA
        rule_5a ISA Employee
        rule_5c ISA Employee
        rule_5c ISA TA
        rule_4a ISA Employee
        rule_4c ISA Employee
        rule_4c ISA Professor
        rule_3a ISA Professor
        rule_3c ISA Professor
        rule_2a ISA Professor
        rule_2c ISA Professor
        rule_1a ISA Professor
        rule_1c ISA Professor
</ISA>
<RELATIONS_TO_DISPLAY>
        set<Section>    REL set<Student>
                        PATH takes      INVERSE is_taken_by
        Section REL TA  PATH assists    INVERSE has_TA
        set<Section>    REL Professor
                        PATH teaches    INVERSE is_taught_by
        Course  REL list<Section>
                        PATH is_section_of
```

```
INVERSE has_sections
        set<Course>     REL set<Course>
                        PATH has_prerequisites
INVERSE is_prerequisite_for
</RELATIONS_TO_DISPLAY>
</LEVEL>
```

Note sul parser del formato *visual* La routine usata per leggere i dati dal file descrizione è molto semplice ed è in grado di elaborare una sola riga alla volta. Per ciò le singole direttive come le `REL` o `ATTRIBUTE`, devono essere scritte su di una sola riga, lo stesso vale per i "tag" che individuano *blocchi* di informazione, anche questi devono stare da soli su di un unica riga[1].

In caso di errori di sintassi, il comportamento normale del parser è quello di ignorare le righe non riconosciute. Fa eccezione il riconoscimento delle `REL` per le quali è stato realizzato parser (sempre molto semplice) che è in grado di rilevare gli errori di sintassi.

La descrizione dello schema di basa sul concetto di "blocco". I blocchi contengono informazioni omogenee, e sono delimitati da una direttiva (un "tag") di inizio-blocco ed una di fine-blocco.

7.2.2 Descrizione delle direttive

Ordine delle direttive vediamo la lista delle direttive e l'ordine in cui queste devono apparire:

```
<LEVEL 0>
   <CLASSES_TO_DISPLAY>    ...  </CLASSES_TO_DISPLAY>
   <CLASSES_COLORS>        ...  </CLASSES_COLORS>
   <EQU>                   ...  </EQU>
   <ISA>                   ...  </ISA>
   <ISA_COLORS>            ...  </ISA_COLORS>
   <RELATIONS_TO_DISPLAY>  ...  <RELATIONS_TO_DISPLAY>
</LEVEL>

...

<LEVEL ...>
   ...
</LEVEL>
```

[1] Nell'esempio, per motivi di spazio alcune direttive sono state spezzate su più righe.

```
...

<LEVEL xx LAST>
   ...
</LEVEL>
```

In particolare occorre rispettare le seguenti regole:

1. Il primo livello deve sempre essere il livello 0 (zero) e l'ultimo livello deve sempre essere "marcato" con la parola chiave `LAST`

2. `<CLASSES_TO_DISPLAY>` deve essere la prima direttiva del livello. Questo permette di sapere se nelle rimanenti direttive vi sono informazioni da non memorizzare come la descrizione di relazioni tra classi che non sono citate tra le classi da visualizzare[2]

3. `<CLASSES_COLORS>` deve apparire dopo `<CLASSES_TO_DISPLAY>`

4. `<ISA_COLORS>` deve apparire dopo `<ISA>`.

Commenti È possibile inserire dei commenti all'interno del file di descrizione, è ignorato, cioè è considerato come commento, tutto ciò che appare su una riga dopo il tag "// " (come in C++). Attenzione a lasciare uno spazio tra `//` e la parola successiva in modo che il parser riconosca il tag.

Descrizione di classi Come si vede bene dall'esempio a pagina 94, la sezione per la descrizione delle classi ha la forma:

[2]Si è fatta questa scelta per aumentare la flessibilità dello strumento, infatti questo permette di visualizzare solo parte di uno schema partendo dalla descrizione completa (ad esempio per la creazione di schemi di navigazione), semplicemente cancellando o commentando la descrizione delle classi che non interessano.

<**CLASSES_TO_DISPLAY**>
<**CLASS**> *nome_classe*
ATTRIBUTE *descrizioneAttributo1*
...
ATTRIBUTE *descrizioneAttributoN*
</**CLASS**>
...
<**CLASS**> nome_regola **RULE**
ATTRIBUTE *descrizioneAttributo1*
...
ATTRIBUTE *descrizioneAttributoN*
</**CLASS**>
</**CLASSES_TO_DISPLAY**>
...

Per ogni classe si può specificare il nome, sulla stessa riga.

La parola chiave `RULE` indica che la classe è da considerare una regola di integrità, per queste l'applet gestisce la visualizzazione OPZIONALE.

Tutto ciò che appare dopo la parola chiave `ATTRIBUTE` verrà trattato come "attributo" della classe e mostrato nella finestra informativa quando si fa un doppio click sulla classe.

Descrizione dei colori delle classi Con questa direttiva è possibile associare ad ogni classe un diverso colore in fase di visualizzazione.

<**CLASSES_COLORS**>
nome_classe *#codice_colore*
...
</**CLASSES_COLORS**>

il codice del colore è una stringa di sei caratteri `#RRGGBB` che descrive il codice in formato RGB (Red Green Blu).

descrizione dell'equivalenza tra classi Gli algoritmi di ottimizzazione del prototipo sono in grado di scoprire eventuali equivalenze tra classi rispetto la relazione di sussunzione. Questa direttiva è stata introdotta per rappresentare tali equivalenze. In pratica, questo costrutto permette di definire nomi diversi per identificare la medesima classe.

<**EQU**>
nome_classe1 **EQU** *nome_classe2*
...
</**EQU**>

Quando si dichiarano equivalenti (1) le classi vengono mostrate in un unico ret-

tangolo in cui appaiono i vari nomi. (2) Le singole classi dichiarate equivalenti mantengono la propria identità di `rule` o classe e l'informazione relativa al proprio colore. Se una rule è dichiarata equivalente ad una classe, la rule viene visualizzata assieme alla classe solo se è abilitata la visualizzazione delle rule. (3) Viene persa la descrizione degli attributi della classe con nome *nome_classe2* in quanto, dichiarando le classi equivalenti, le due classi si presume abbiano le stesse caratteristiche.

Descrizione relazioni ISA si specificano con la sintassi:

<**ISA**>
nome_classe1 **ISA** *nome_classe2*
...
</**ISA**>

Le relazioni ISA sono le informazioni più importanti considerate dall'algoritmo di ordinamento.

Descrizione colore relazioni ISA È possibile specificare un dato colore per tutte le relazioni di ereditarietà che hanno come discendente una data classe.

<**ISA_COLORS**>
nome_classe *#codice_colore*
...
</**ISA_COLORS**>

Questa direttiva è stata introdotta per evidenziare le relazioni di ereditarietà tra le Query, mappate come classi virtuali, e le classi da interrogare.

Descrizione relazioni di aggregazione come già detto a pagina 91, è possibile descrivere due tipi di relazioni: (1) normali relazioni di aggregazioni, oppure (2) riferimenti a classi.

In ogni caso la descrizione deve apparire tra i tag:

`<RELATIONS_TO_DISPLAY>` e `</RELATIONS_TO_DISPLAY>`

La sintassi che deve avere la singola riga che descrive una relazione è descritta in tabella 7.1.

il seguente esempio descrive un *puntatore* attributo dalla classe FORM che mappa in un oggetto della classe TO:

FROM		TO		FORM-path
storage	REL	material	PATH	stock.all.item

significa che nella classe FORM esiste un campo che si chiama `stock.all.item` che è di tipo `storage`, esaminando il campo della classe TO si deduce che la classe FORM è `material` e la classe TO è `storage`,

< RelSyntax >	::=	< rClass > **rule** < rClass > **PATH** < Path > |
		< rClass > **rule** < rClass > **PATH** < Path >
		INVERSE < Path >
< rClass >	::=	**set**< < Class > > |
		list< < Class > > |
		< Class >
< Path >	::=	< DottedName >
< Class >	::=	< Identifier >

Tabella 7.1: Sintassi riga che descrive una relazione

La parte FROM indica il tipo di attributo che si trova nella classe FORM e non è il nome della classe di partenza, lo stesso vale per la parte TO. Il nome della classe TO si ricava dal tipo del campo presente nella classe FROM. Questa sintassi è non ridondante per la rappresentazione delle relationship ma è poco intuitiva per la rappresentazione dei *puntatori*.

Tipi puntatori:

Vi sono due tipi di puntatori. Puntatori semplici, cioè con molteplicità *uno*, sono visualizzati come freccie sottili.

Puntatori multipli, sono degli *insiemi di puntatori*, hanno molteplicita *molti* e sono visualizzati con freccie sottili con una doppia punta. I puntatori multipli possono essere dei `set` di puntatori oppure delle `list`, l'applet però non distingue questi due tipi. I possibili tipi di puntatori sono rappresentati in tabella 7.2

FROM		TO		FORM-path	tipo freccia	molteplicità
aaa	REL	bbb	PATH	abc	bbb `-->` aaa	uno a uno
set<aaa>	REL	bbb	PATH	abc	bbb `->>` aaa	uno a molti
list<aaa>	REL	bbb	PATH	abc	bbb `->>` aaa	uno a molti

Tabella 7.2: tipi di puntatori

tipi relazioni:

Vi sono quattro tipi di relazioni che corrispondono alle quattro possibili combinazioni dei puntatori con molteplicità *uno* o *molti*. I possibili tipi di relazioni sono rappresentati in tabella 7.3

esempio:

FROM		TO		FORM-path		TO-path
aaa	REL	bbb	PATH	abc	INVERSE	cba
set<aaa>	REL	bbb	PATH	abc	INVERSE	aba
aaa	REL	set<bbb>	PATH	abc	INVERSE	cba
set<aaa>	REL	set<bbb>	PATH	abc	INVERSE	cba

Tabella 7.3: tipi di relazioni

`aaa REL set<bbb> PATH abc INVERSE cba`
significa che nella classe FROM, che è `bbb`, c'è un campo che si chiama `abc` di tipo `aaa` e nella classe TO, che è `aaa`, c'è un campo che si chiama `cba` di tipo `set<bbb>`.

7.2.3 Il modulo C per la generazione del formato visual

Per facilitare l'uso di questo strumento è stato realizzato un modulo C per la generazione del formato Visual. Il modulo può essere incluso in un qualunque programma C che generi il formato *visual*, ad esempio il programma traduttore descritto nel capitolo 6 fa uso di questo modulo.

Il modulo è composto da un file ".h" ed un ".c". Nel file ".h" sono descritte le *signature* delle routine realizzate, mentre nel file ".c"vi è l'implementazione di tali routines.

Di seguito sono riportate alcune dichiarazioni di funzioni tratte dal file ".h":

```
/* selezione dello stream di output */
void vf_set_output_stream(FILE *file);

/* --- Gestione Livelli
 */
void vf_level_begin(
   int number,    /* numero del livello */
   boolean last   /* puo' valere
                   *  0: non e' l'ultimo livello
                   *  1: e' l'ultimo livello
                   */
   );
void vf_level_end();

/* ---  gruppo dei nodi (classi) da visualizzare
 */
void vf_class_group_begin();
```

```
void vf_class_group_end();
/* ---  singole classi
 */
void vf_class_begin(
   char *name,      /* nome della classe */
   boolean rule     /* se la classe \'e una Rule
                     * le 'rule' sono classi che possono essere nascoste.
                     * puo' valere:
                     *   0: Non puo' essere nascosta
                     *   1: Puo' essere nascosta
                     */
   );
void vf_class_end();
void vf_attribute(
   char *name       /* nome dell'attributo, una qualunque stringa */
   );

/* ---  per cambiare il colore delle classi
 */
void vf_classesColors_begin();
void vf_classesColors_end();
void vf_classesColors(char *class, int r, int g, int b);
```

Le singole funzioni non fanno altro che scrivere in un dato stream di output le corrispondenti direttive. Modulo è molto semplice ma nella sua semplicità apporta dei vantaggi: (1) velocizza la scrittura di programmi con output in formato *visual*, (2) riduce la possibilità di commettere errori in fase di scrittura di tali programmi. (3) in caso di modifiche al formato *visual*, vengono ridotte e accorpate le modifiche da apportare al codice.

7.3 Il programma (applet `scvisual`

Descrizione di massima delle classi Il programma è una normale applet Java, in questa sezione procederemo descrivendo gli oggetti che compongono il programma scendendo sempre più in dettaglio.

In figura 7.3 è riportato lo schema dell'applet. Tale figura è stata realizzata usando la funzionalità di output in formato XFig dell'applet, previa descrizione delle classi in un file in formato *visual*.

L'oggetto principale è `scvisual`, la cui dichiarazione è:

```
public class scvisual extends Applet implements Runnable
```

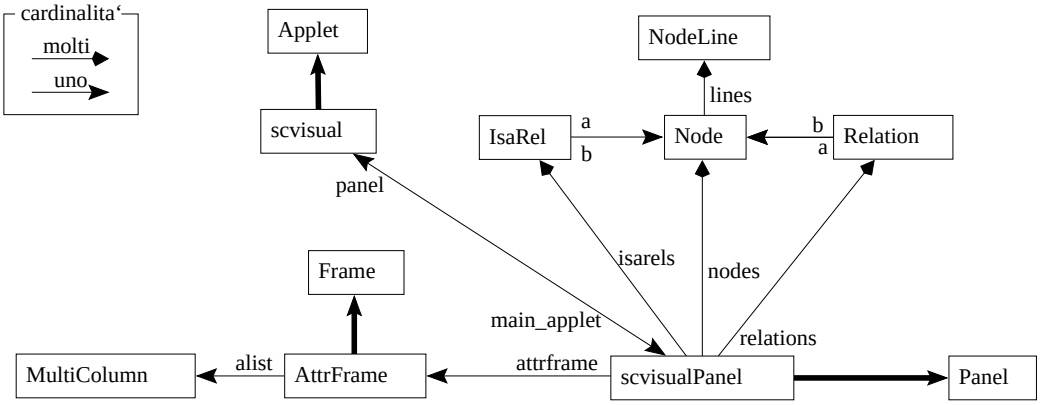


Figura 7.3: Classi che compongono l'applet `scvisual`

Significa che la classe `scvisual` eredita dalla classe `Applet` ed "implementa" l'interface `Runnable`.

Il *layout* dell'applet è `BorderLayout`, in questo layout è possibile definire cinque oggetti grafici, uno (*center*) che si trova al centro ed altri quattro che stanno ai bordi come se facessero da cornice (*nord*, *est*, *sud*, *ovest*). Nel nostro caso sono definiti solo gli oggetti *center* e *sud*.

Al centro è inserito un oggetto di tipo `scvisualPanel` che è l'oggetto che si preoccupa della gestione della visualizzazione del grafo.

A sud vi è un oggetto di tipo `Panel` che a sua volta contiene altri oggetti di tipo `Button`, `Checkbox` e `Choice`. Questo oggetto è quello che gestisce i pulsanti di controllo.

L'oggetto principale realizza le funzioni:

- caricamento dei dati, cioè apre l'URL passata come parametro ed effettua il parsing del file che descrive lo schema. Vengono memorizzati solo i dati necessari e non ridondanti, ad esempio vengono scartate tutte le relazioni che coinvolgono classi non dichiarate in precedenza, vengono scartate anche le relazioni duplicate.

- della gestione dei pulsanti di controllo, cioè permette di (1) lanciare l'algoritmo di ordinamento del grafo, (2) scegliere se visualizzare o no le relazioni ISA, (3) scegliere se visualizzare o no le relazioni intensionali, (4) scegliere se visualizzare o no le classi `rule`, (5) scegliere quale *livello del grafo* visualizzare.

- della gestione dei "tips", i suggerimenti che appaiono nella barra dei messaggi del browser.

La classe scvisualPanel Come è gia stato detto, gli oggetti di questa classe realizzano la gestione della visualizzazione del grafo. Nell'applet vi è un unico oggetto di questa classe, questo è associato alla variabile `panel`.
`panel` contiene i tre array degli oggetti da visualizzare, sono *array di puntatori ad oggetti*, ognuno può *puntare* al massimo 250 oggetti:

```
Node     nodes[]     = new Node[250];      // lista dei nodi
IsaRel   isarels[]   = new IsaRel[250];    // lista ISA
Relation relations[] = new Relation[250];  // lista relazioni
```

Contiene inoltre un oggetto di tipo `AttrFrame`, questo si preoccupa della gestione della finestra "`Description Window`" in cui appaiono gli attributi delle classi e delle relazioni.
L'oggetto `scvisualPanel` realizza le funzioni:

- Visualizzazione delle classi e delle relazioni.
- Gestione del mouse per *trascinare* i rettangoli che rappresentano le classi, o del doppio o triplo click.
- Genera la descrizione del grafo in formato XFig.
- Mette a disposizione diversi metodi che agiscono sugli array di nodi o relazioni. Realizzano la funzione di inserimento e di cancellazione di un elemento, e altre funzioni come `nodeHit` che verifica se in un dato punto dello schermo è presente un nodo, o `relationHit` che, dato un punto dello schermo, genera la lista di tutte le relazioni che "passano" per quel punto.

La classe Node Ogni oggetto di questa classe rappresenta una o più *class* o *rule*. In un nodo (un oggetto della classe Node), vengono memorizzate:

- Informazioni come il nome, il colore e se si tratta di una rule, di tutte le classi dichiarate equivalenti. Questo è fatto con un array di oggetti di tipo NodeLine.
- La lista degli attributi delle classi.
- Informazioni per la visualizzazione. In particolare ogni Nodo memorizza la dimensione e la posizione del rettangolo con cui è rappresentata. Altri flag servono per sapere se un nodo è da visualizzare (ricordo che le rule possono essere nascoste).

- Alcune variabili di comodo utilizzate dall'algoritmo di calcolo dell'ordinamento.

Vi sono diverse funzioni che un oggetto `Node` realizza:

- Routine di calcolo della dimensione del rettangolo da visualizzare. La dimensione dipende dalla grandezza delle labels associate al nodo, da quante labels sono da visualizzare e quali labels sono da visualizzare.
- Funzioni di interrogazione. E' possibile sapere se un nodo e' uguale ad un altro o se i rettangoli si sovrappongono.
- la funzione paint che è la routine di visualizzazione. Disegna il rettangolo bordato e vi scrive dentro le labels.
- Routine di output del grafo in formato XFig. Simile alla routine di visualizzazione, pero' produce un output a caratteri su un dato stram.
- Il metodo `Point node_better_point_one(Node b)` calcola il punto di "arrivo" migliore sul bordo del nodo per la freccia dal nodo `b`. Questa routine è usata per tracciare le freccie tra i nodi.

La classe IsaRel Gli oggetti di questa classe rappresentano le relazioni di ereditarietà ISA, sia quelle lette dal file dati che quelle calcolate (relazioni *fittizie*). Per ogni relazione viene memorizzato:

- I due nodi superclasse e sottoclasse.
- Un flag che indica se la relazione è *fittizia*.
- Informazioni per la visualizzazione come Il colore della freccia ed un flag `display`, questo indica istante per istante se la relazione è visibile.

I metodi caratteristici dalla classe `IsaRel` sono: `paint` per la visualizzazione a video e `toXfig` per l'output XFig. Da notare che la visualizzazione avviene invocando il metodo `paint` della classe `Arrow`.

La classe Relation La classe `Relation` contiene gli oggetti che descrivono le relazioni di aggregazione o puntatori ad oggetti. A pagina 100 sono elencati i tipi di relazioni che possono essere rappresentate dall'applet.

Questa classe è in grado di gestire gli autoanelli cioè le relazioni da un nodo a se stesso. Questi vengono gestiti in modo molto semplice, viene costruito un anello *fisso* costituito di quattro segmenti attorno all'angolo in alto a sinistra del nodo.

La realizzando la gestione degli autoanelli si è introdotto un caso particolare che ha *rovinato* la generalità degli algoritmi per la gestione delle relazioni.
Per ogni relazione viene memorizzato:

- I due nodi da mettere in relazione, le molteplicità nelle due direzioni, se si tratta di relazione ordinata e i path-name con cui sono identificate le relazioni all'interno delle due classi.
- Un flag che indica se la relazione è *fittizia*.
- Informazioni per la visualizzazione come Il colore della freccia, il flag `display` che indica istante per istante se la relazione è visibile, i punti di partenza e di arrivo della freccia. Questi ultimi sono usati dalla funzione `isHit` per verificare se per un dato punto passa la freccia corrispondente a una relazione.

I metodi caratteristici dalla classe `Relation` sono:

- Funzioni per l'output grafico: `paint` per la visualizzazione a video e `toXfig` per l'output XFig. Anche in questo caso la visualizzazione avviene invocando il metodo `paint` della classe `Arrow`.
- La funzione `inArea`:
  `boolean inArea(Point p, Point pa, Point pb, int delta)`
  è in grado di determinare se il punto `p` cade in un area di larghezza `delta` attorno alla retta da `pa` a `pb`. Questa funzione si basa sull'applicazione della formula della distanza punto-retta in uno spazio a due dimensioni.
- La funzione `isHit`. Utilizza la funzione `inArea` e, dato un punto `p`, determina se per `p` passa la freccia che rappresenta la relazione descritta dall'oggetto corrente.
- La funzione `inv_equals` serve a verificare se una relazione è già stata memorizzata. Una *relazione di aggregazione* tra due classi A e B, può essere descritta in due modi equivalenti: (1) descrivendola da A verso B oppure (2) descrivendola in senso *inverso* da B verso A. Questa funzione confronta due relazioni ed è in grado di stabilire se le due relazioni sono una l'*inversa* dell'altra.

La classe Arrow Si tratta della classe che disegna le freccie sia a video che nell'output XFig. Questa classe è in grado di disegnare freccie di qualunque spessore e colore, con un qualunque numero dato punte per ciascuna delle due direzioni. Vi sono due tipi di freccie. Le freccie *spesse* caratterizzate dallo spessore del *corpo*

maggiore di un pixel, che a video sono disegnate del colore specificato, con un bordo nero. E le freccie *sottili* con spessore minore o uguale a uno, il cui *corpo* è disegnato con una linea del colore specificato.
Ogni freccia è caratterizzata dalle seguenti informazioni:

Point pa punto di *partenza* della freccia.

Point npa numero di punte nella direzione di a.

Point pb punto di *arrivo* della freccia.

Point npb numero di punte nella direzione di b.

Color c colore della freccia.

float s spessore del corpo della freccia. indica lo spessore in numero di pixel, un valore minore o uguale a 1 significa di costruire la freccia *semplice* usando una sola linea.

int defl lunghezza in numero di pixel della punta della freccia semplice.

int defh larghezza in numero di pixel della punta della freccia semplice.

double deflung coefficiente di lunghezza della punta per le freccie *spesse* relativo allo spessore del corpo, di quanto la punta rientra nel corpo.

double defsporg coefficiente di sporgenza per le freccie *spesse* relativo allo spessore del corpo, di quanto la punta esce dalla sagoma del corpo.

La procedura di `paint` (disegno a video) tratta in modo diverso le freccie *semplici* da quelle *spesse*. Le prime sono più semplici da disegnare, richiedono il calcolo di due soli punti per ogni punta disegnata. Le per disegnare la punta di una freccia spessa occorre calcolare almeno quattro punti.
Nota sull'output in formato XFig: in formato XFig il diverso numero delle punte di una freccia è viene rappresentato cambiando lo stile della punta stessa. Come si può vedere in figura 7.3, le freccie con una sola punta sono rappresentate con freccie più *svasate* mentre quelle con due punte diventano freccie con una sola punta ma più *piena*. Il motivo di questa scelta va fatto risalire a considerazioni di comodità di editing di uno schema e alla semplicità del formato XFig. Per XFig le freccie sono delle *poli-linee* con particolari intestazioni, purtroppo XFig non prevede freccie con più punte. Secondo la scelta fatta, una freccia rimane un unico oggetto facile da muovere, orientare e ridimensionare. L'alternativa era realizzare freccie con più punte disegnando le varie punte come oggetti a se stanti, questo rendeva *noiosa* una eventuale fase di editing dello schema in quanto, se avessimo

spostato, o peggio, ruotato una freccia le singole punte della freccia non avrebbero seguito la freccia ma sarebbero dovute essere spostate e orientate una ad una.

7.4 Descrizione algoritmi particolari

7.4.1 Algoritmo di ordinamento dei nodi per la visualizzazione

L'algoritmo descritto in questa sezione tenta un piazzamento dei nodi del grafo sul *pannello*. È un algoritmo euristico, concettualmente semplice. Si cerca di visualizzare la struttura ad albero di rispetto alle relazioni di ereditarietà, per ogni nodo viene calcolato il *livello* e la *colonna* in cui dev'essere visualizzato. Funziona in tre passi:

1. In base alle relazioni di ereditarietà si decide a che livello visualizzare ogni singolo nodo, in modo che in alto vi stiano i *genitori* ed in basso i nodi *figli* che ereditano dai nodi *genitori*. In questo passo si tiene conto della presenza di eventuali cicli, cioè di situazioni in vi sono nodi che sono discendenti e antenati di se stessi.

2. Si decide in che ordine presentare i nodi di ogni livello. In questa fase i nodi vengono disposti su colonne e si assegna la colonna migliore, rispetto ad una data funzione di costo, ad ogni singolo nodo. In questa fase il problema maggiore è dovuto alla gestione dell'ereditarietà multipla.

3. Si decide come mappare le colonne del passo precedente in coordinate video. Solo in questo passo si tiene conto dalle dimensioni dell'area video utilizzabile.

Si tratta di un algoritmo euristico in quanto non cerca il *piazzamento* ottimo ma, applica un serie di criteri per ottenere un risultato che, a mio avviso, è accettabile.
L'algoritmo è in un certo senso *non deterministico*, infatti, il piazzamento che si ottiene dipende dall'ordine dei nodi all'interno dell'array `nodes[]`, cambiando tale ordine, lasciando il medesimo contenuto informativo dato dai nodi e dalle relazioni, si ottiene un diverso piazzamento.

Assegnamento dei Nodi ai Livelli

Ogni nodo ha l'attributo `l` che indica a quale livello è stato assegnato e `cl` che indica se il nodo appartiene ad un ciclo rispetto alla relazione di ereditarietà.
Se non ci fossero cicli, questa parte dell'algoritmo consisterebbe in una routine iterativa in cui si scorre un albero, dalla radice verso le foglie cioè:

(1) prima si cercano le radici dell'albero, che sono i nodi senza genitori, a questi viene associato il livello zero. Poi, (2) si applica la routine iterativa per cui ad ogni passo si incrementa il livello, i nodi di questo livello sono tutti i nodi che ereditano dai nodi del livello precedente.

Nell'applet ho utilizzato questa routine, con qualche ritocco per la gestione dei grafi in presenza di cicli.

Per gestire i cicli ho introdotto il concetto di *ciclo puro*: un ciclo lo considero puro quando tutti i nodi componenti il ciclo ereditano solo da nodi del ciclo stesso.

Nell'algoritmo però utilizzo questa "definizione": un ciclo è considerato un *ciclo puro* quando tutti i nodi componenti il ciclo ereditano o da nodi del ciclo stesso oppure da nodi di livelli superiori (nodi già elaborati).

Inoltre per l'algoritmo non possono far parte di un ciclo nodi a cui è già stato assegnato il livello oppure che sono già stati riconosciuti come membri di un ciclo puro.

Cosa centrano queste definizioni?

Il problema per piazzare un ciclo è trovare un *bandolo* da cui cominciare la rappresentazione. Se per un ciclo fisso il livello di un nodo B ed ignoro le relazioni di ereditarietà per cui B eredita dagli altri nodi del ciclo, allora posso permettermi di trattare gli altri nodi del ciclo come se fossero un semplice albero sotteso a B.

A questo punto il problema è, come trovare tale *bandolo* in modo che l'intero ciclo sia piazzato nel grafo in modo "ordinato", in pratica occorre scegliere il momento giusto per estrarre un nodo da un ciclo. Per fare ciò occorre risolvere il seguente problema:

Dati i nodi del ciclo, li denotiamo con IC (Insieme nodi del Ciclo). dati i nodi IG (Insieme Genitori), sono quei nodi G per cui esiste almeno una relazione di ereditarietà per cui G è genitore di un nodo del ciclo. Allora tutti i nodi IC devono avere livello inferiore a tutti i nodi IG, questo perchè i nodi IC fanno parte di un ciclo, e quindi un qualunque nodo di IG è genitore di tutti i nodi del ciclo.

Per questo motivo, il più *alto* nodo del ciclo deve essere visualizzato *sotto* al più basso nodo di IG.

Bene, applicando la routine iterativa descritta sopra, un nodo è da piazzare nel livello corrente solo se: (1) eredita solo da nodi dei livelli superiori oppure (2) fa parte di un ciclo puro. A questo punto si ha che:

- Per la definizione estesa di ciclo puro, mi garantisce che tutti gli elementi del ciclo devono avere livello almeno inferiore al livello attuale, altrimenti il ciclo non è puro.

- Poichè per l'algoritmo non possono far parte di un ciclo nodi a cui è già stato assegnato il livello oppure che sono già stati riconosciuti come mem-

bri di un ciclo puro, nei livelli successivi i nodi appartenenti a cicli già riconosciuti sono trattati come nodi dell'albero sotteso al nodo *bandolo* e, computazionalmente, non devo più cercare cicli tra i nodi già trattati.

Esempio, dato il grafo di figura 7.4:

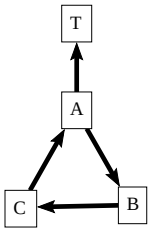


Figura 7.4: esempio di grafo ciclico

A livello 0 vi è solo il nodo T, l'algoritmo dev'essere in grado di riconoscere il ciclo `A-B-C-A`, questo dev'essere piazzato a partire dal livello 1, perchè A eredita da T, dunque tutti i nodi del ciclo devono stare sotto a T. Per la definizione data di ciclo puro, tale ciclo viene riconosciuto come ciclo puro solo dal livello 1.

Note sulla complessità computazionale Se fossero presenti tutte le relazioni possibili tra tutti i nodi la complessità per la ricerca di un ciclo può essere N fattoriale ove N è il numero dei nodi.
Questo è un caso estremamente improbabile, soprattutto per l'uso per cui è nato questo strumento che è visualizzare schemi in cui non sono presenti cicli.

Assegnamento dei nodi dei livelli alle Colonne

Questa è senz'altro la parte dell'algoritmo più complessa, anche se il principio applicato è semplice.
L'algoritmo prende in considerazione un livello alla volta e cerca di piazzare i nodi del livello attuale sulle colonne esistenti. Il numero di colonne varia dinamicamente. L'iterazione parte dal livello 0 con una sola colonna.

l'algoritmo Di seguito vedremo quali sono le operazioni che permettono di assegnare le colonne ai nodi di un certo livello, occorre tenere presente che ai nodi dei livelli precedenti, le colonne sono già state assegnate.

1. Provo a fare il piazzamento dei nodi nelle colonne del livello attuale, per ogni nodo scelgo la colonna migliore in cui metterlo e glielo metto secondo

una certa funzione di costo. Ovviamente, al livello zero, tutti i nodi del livello verranno assegnati all'unica colonna.

2. Controllo se ci sono colonne con più di un nodo del livello considerato. Per ogni colonna con più di un nodo:

   - la colonna viene suddivisa in tre colonne attraverso l'operazione di *split*, esempio:

numero colonne prima dello split	3			4	6
situazione prima dello split		O		O	O
situazione dopo lo split	n	O	n	O	O
numero colonne dopo lo split	3	4	5	6	7

     lo split si propaga alle colonne dei livelli superiori

   - gli elementi della colonna originale vengono ridistribuiti tra le tre colonne secondo una certa funzione di costo.

     N.B. perche' l'algoritmo termini impongo che almeno un nodo venga "tolta" dalla colonna originale

   - se ci rimane una colonna vuota la elimino.

     Attenzione: NON posso lasciare la colonna originale vuota! Almeno in uno livello, quello attuale o tra i precedenti, ci dev'essere un nodo.

     Ad ogni split controllo che la colonna che ha generato lo split non si vuoti di tutti i nodi, se si vuota allora scelgo di mantenere sulla colonna il nodo che ci "guadagna" di più o meglio, che ci "perde" di meno ad esser spostato. Questo viene fatto nel seguente modo:

     una volta effettuato lo split, se non ci sono nodi sulla colonna C allora ad uno ad uno provo a spostare i nodi delle colonne adiacenti sulla colonna C e ri-sposto su C quel nodo che ha la minima differenza tra $funzioneCosto(C) - funzioneCosto(colonna\ in\ cui\ si\ trova)$

3. ripeto il punto 2 fino a che in ogni colonna non sia rimasto al più un nodo.

La funzione di costo La funzione di costo ritorna un indicatore di quanto buona è la "posizione" di un nodo rispetto al livello e la colonna ad esso assegnate tenendo conto della "posizione" degli altri nodi. La funzione funziona anche se ci sono dei nodi a cui non è stata ancora assegnata la colonna. Più la posizione è buona, più il numero ritornato è grande.

Il calcolo del costo tiene conto di tre fattori:

1. La lunghezza delle freccie che rappresentano le relazioni ISA. Questo è il principale fattore da ridurre.

2. La lunghezza delle freccie che rappresentano le relazioni di aggregazione, devono essere il più corte possibile.

3. Il livello di accoppiamento tra gli alberi sottesi ai vari nodi. Quando ad un nodo non è ancora stata assegnata la colonna, non è possibile calcolare la lunghezza delle freccie. Questo fattore serve proprio in questi casi. Vedremo di seguito di cosa si tratta di preciso.

Il *livello di accoppiamento* tra gli alberi sottesi a due nodi viene valutato contando i nodi comuni ai due sottoalberi. In questo caso, considero come appartenenti all'albero sotteso ad un dato nodo N tutti i nodi che in qualche modo sono collegati da freccie di ereditarietà ad N, cioè:

1. I nodi che ereditano dal nodo *radice* a cui è stato assegnato un livello inferiore al livello del nodo radice.

2. Ma anche i nodi **antenati** del nodo *radice* a cui però è stato assegnato un livello inferiore al livello del nodo radice.

Per rendere più efficiente il calcolo del livello di accoppiamento faccio uso di una particolare struttura dati inizializzata una volta per tutte da una routine ricorsiva.
La struttura dati consiste in array di boolean di dimensione pari al numero di nodi da visualizzare. Ogni nodo N ha un tale array di boolean di nome `cor`, si ha che `cor[i]` vale `true` se il nodo `nodes[i]` fa parte dell'albero sotteso ad N. Grazie a tali array, per calcolare il livello di accoppiamento tra due nodi N de M, basta contare il numero di elementi in `N.cor` ed `M.cor` tali che `N.cor[i]` e `M.cor[i]` valgono entrambi `true`.

Piazzamento di *livelli* e *colonne* sullo schermo

Una volta stabilito il numero di riga e di colonna per ogni nodo il piazzamento è praticamente già fatto. Basta dividere l'area video disponibile nel numero di righe e colonne date ed assegnare *proporzionalmente* le coordinate a video.
La routine di piazzamento contiene una procedura che tenta di disporre meglio i livelli sovraffollati: nel caso in cui vi siano dei nodi dello stesso livello che appaiono come *sovrapposti*, allora la riga del livello viene divisa in due righe: i nodi su colonne pari sono piazzati sulla riga più in alto mentre i nodi su colonne dispari sono piazzati sulla riga più in basso.

7.4.2 Algoritmo di creazione relazioni ISA fittizie

Come già detto a pagina 89, questo algoritmo calcola le relazioni isa fittizie, cioè quelle relazioni isa che sono implicite nella dichiarazione dello schema ma che

coinvolgono delle `rule` e per questo motivo, quando le `rule` sono nascoste, potrebbero non essere mostrate.

Per ogni classe C che eredita da una `rule` R, viene applicato l'algoritmo realizzato dalla funzione `create_norules_isarels` che è ricorsiva.

Tale funzione cerca tutti i *primi* antenati C1 (genitori, nonni) di R che siano classi e non rule e crea le relazioni fittizie C ISA C1. Usando il termine *primi* antenati intendo che la ricerca ricorsiva procede solo tra i genitori che sono delle regole e si ferma non appena trova una classe non regola.

All'algoritmo vengono passati due parametri, la classe di partenza `sou` e il nodo da esaminare `cild` che è sempre una `rule` per costruzione ricorsiva.

ad ogni passo l'algoritmo cerca le superclassi del nodo `cild` tali che:

1. Non siano gia' state prese in considerazione (per evitare i cicli).
2. Se sono classi e non rule crea la relazione fittizia tra la classe sou e la classe appena trovata e termina la ricerca ricorsiva.
3. Se sono rules continua ricorsivamente la ricerca.

7.4.3 Algoritmo di creazione relazioni di aggregazione fittizie

Come già detto a pagina 89, questo algoritmo calcola le relazioni di aggregazione fittizie, cioè quelle relazioni che sono implicite nella dichiarazione dello schema ma che coinvolgono delle `rule` e per questo motivo, quando le `rule` sono nascoste, potrebbero non essere mostrate.

Per ogni classe C che eredita da una `rule` R, viene applicato l'algoritmo realizzato dalla funzione `create_norules_relations` che è ricorsiva.

All'algoritmo vengono passati due parametri, la classe di partenza `sou` e il nodo da esaminare `r` che è sempre una `rule` per costruzione ricorsiva.

Partendo dalla classe sou (base) cerca tutte le `rules` R antenate di `sou` che hanno relazioni con altri classi D non rules. Per ogni relazione (R, D) trovata crea la relazione fittizia (`sou`, D).

La ricorsione continua con i genitori di `r` che sono delle `rule` non ancora prese in considerazioni (uso un flag, è utile in presenza di cicli).

Capitolo 8

Conclusioni e sviluppi futuri

Nella presente tesi è stato studiato lo stato dell'arte nel campo delle basi di dati ad oggetti con particolare attenzione all'accoppiamento con tecniche dell'intelligenza artificiale.

È stato realizzato un sito WWW a disposizione della comunità di ricerca attraverso il quale è possibile vedere i risultati ottenibili grazie all'applicazione di tali tecniche al progetto di schemi e all'ottimizzazione di interrogazioni nell' ambito delle basi di dati ad oggetti.

Per la realizzazione del sito è stata realizzata un'interfaccia grafica utilizzando il linguaggio **Java**, un linguaggio ad oggetti recentemente sviluppato da Sun Microsystems che sta ottenendo un enorme successo sia in ambito di ricerca che tra gli sviluppatori di applicazioni.

Nell'ambito della tesi è stato inoltre realizzato un traduttore tra due linguaggi di descrizione di schemi di basi di dati ad oggetti: ODL (Object Definition Language) descritto nello standard ODMG-93 per OODBMS ed il linguaggio OCDL sviluppato presso il Dipartimento di Scienze dell'Ingegneria (DSI). Per tale realizzazione è stato utilizzato il software Lex&Yacc, un potente strumento per la creazione di parser di linguaggi.

Possibili evoluzioni della seguente tesi Dal punto di vista teorico potrebbe essere studiata un'estensione al linguaggio OCDL in grado di supportare associazioni tra classi, questo permetterebbe la piena gestione delle `relationship` dell'ODL.

Inoltre si potrebbe utilizzare Applet Java come punto di partenza per la realizzazione di uno strumento per il disegno di di schemi di basi di dati ad oggetti, permettendo al progettista di svolgere la propria attivitá attraverso un'interfaccia grafica *intelligente*.

Dal punto di vista realizzativo potrebbero essere sviluppate le seguenti estensioni.

Il traduttore dovrebbe essere esteso per permettere l'assegnamento di espressioni algebriche a costanti. Inoltre, la generazione del formato *visual* potrebbe essere migliorata. Attualmente vengono tradotte in "relazioni tra classi" solo le `relationship`. Una possibile estensione potrebbe essere quella di riconoscere gli attributi *che assumono valori in classi* e tradurli nel formato *visual* come *relazioni unidirezionali tra classi*.

L'*Applet Java* `scvisual` sviluppata nella tesi potrebbe essere migliorata nella parte grafica rendendola ancora più *user friendly*. Ad esempio, migliorando la visualizzazione degli attributi delle classi e delle proprietà delle relazioni. Infine, il formato *visual* potrebbe essere rivisto e corretto, scrivendo un nuovo parser nell'applet in grado di gestire costrutti più potenti e flessibili.

Appendice A

Il modello OCDL

OCDL (Object and Constraint Definition Language) è un modello per basi di dati orientat e agli oggetti che permette la modellazione di valori complessi, dell'ereditarietà multipla e l'espressione di un numero considerevole di vincoli di integrità (vedi [**?**, **?**]). Esso è costituito dall'estensione con vincoli di integ rità del modello **ODL** (*Object Definition Language*, vedi [**?**]) che integra tecniche di inferenza introdotte per le *Logiche Descrittive* (*DL*) nell'ambito dell'Intelligenza Artificiale.

A.1 Il modello ODL

Si assume una ricca struttura per il *sistema di tipi atomici* o *sistema di tipi base*: oltre ai tipi atomici *integer*, *boolean*, *string*, *real* e *tipi mono–valore*, consideriamo anche la possibilità che siano usati dei sottoinsiemi di tipi atomici, come ad esempio intervalli di interi.

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

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

Ai tipi può essere dato un nome: avremo quindi nomi di tipo–valore e nomi di tipo–classe. I nomi di tipo–classe (alcune volte verrà usato semplicemente il termine nome di classe) può essere *base* (denotato anche come primitivo) o *virtuale*, per riflettere la differente semantica discussa nell'introduzione.

Di seguito si affrontano in maniera più rigorosa questi primi concetti.

A.1.1 Sistema dei tipi atomici

Si indichi con $\mathcal{D}$ l'insieme infinito numerabile dei valori atomici (indicati con $d_1, d_2, \ldots$), e.g., l'unione dell'insieme degli interi, delle stringhe e dei booleani. L'insieme numerabile di designatori di tipi atomici che si considera è

$$\mathbf{B} = \{\texttt{integer}, \texttt{string}, \texttt{boolean}, \texttt{real}, i_1\text{–}j_1, i_2\text{–}j_2, \ldots, d_1, d_2, \ldots\},$$

dove gli i_k–j_k indicano tutti i possibili intervalli di interi e i d_k indicano tutti gli elementi di `integer` $\cup$ `string` $\cup$ `boolean` (i_k può essere $-\infty$ per denotare il minimo elemento di `integer` e j_k può essere $+\infty$ per denotare il massimo elemento di `integer`).

A.1.2 Oggetti complessi, tipi e classi

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

Definizione 1 (Tipi) *Dati gli insiemi* $\mathbf{A}$, $\mathbf{B}$ *e* $\mathbf{N}$, *il* sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ *denota l'insieme di tutte le* descrizioni dei tipi *(S, S', ...), detti anche brevemente* tipi, *su* $\mathbf{A}, \mathbf{B}, \mathbf{N}$, *che sono costruiti rispettando la seguente regola sintattica astratta (assumendo $a_i \neq a_j$ per $i \neq j$):*

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

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

Definizione 2 (Valori) *Dati gli insiemi $\mathcal{O}$ e $\mathcal{D}$, si definisce l'insieme $\mathcal{V}(\mathcal{O})$ dei* valori su $\mathcal{O}$ *(denotati da v, v') come segue (assumendo $p \geq 0$ e $a_i \neq a_j$ per*

$i \neq j$*):*

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

Definizione 3 (Dominio) *Dato un insieme di identificatori di oggetti $\mathcal{O}$, un* dominio δ *su $\mathcal{O}$ è una funzione totale da $\mathcal{O}$ a $\mathcal{V}(\mathcal{O})$.*

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

A.1.3 Schema di base di dati

Definizione 4 (Schema) *Dato un sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, uno* schema σ *su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ è una funzione totale da $\mathbf{N}$ a $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$.*

Uno schema σ associa ai nomi di tipi–classe e di tipi–valore la loro descrizione: introduciamo i simboli $\sigma_{\mathtt{P}}$, $\sigma_{\mathtt{V}}$, $\sigma_{\mathtt{T}}$ per rappresentare rispettivamente lo schema di una classe primitiva, di una classe virtuale e di un tipo-valore.
La possibilità di utilizzare un nome di tipo nella descrizione di un altro nome può far sorgere nello schema *descrizioni circolari*, cioè descrizioni di nome che fanno riferimento, direttamente o indirettamente, al nome stesso. Formalmente i cicli sono riconosciuti attraverso la nozione di *dipendenza*:

Definizione 5 (Dipendenza) *Dati N e $N' \in \mathbf{N}$, diciamo che N dipende direttamente da N' sse N' è contenuto nella descrizione di $N, \sigma(N)$. La chiusura transitiva "dipende direttamente da" verrà indicata con "dipende da".*

Definizione 6 (Cicli) *Dato uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, la descrizione di un nome di tipo N è detta* circolare *o, più brevemente, il nome di tipo N è detto* ciclico*, sse N "dipende da" N.*

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

Definizione 7 (Ereditarietà) *Dato uno schema σ su $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$, diremo che N* eredita da *N', denotato con $N \prec_\sigma N'$, sse $\sigma(N) = S_1 \sqcap \cdots \sqcap S_n,\ n > 0$, e $N' = S_i$, per alcuni $i,\ 1 \leq i \leq n$.*

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

Perchè uno schema sia *ben formato*, le *definizioni dei tipi-valore* e le *relazioni di ereditarietà* devono essere *ben fondate*.

La prima condizione garantisce che i tipi valore definiti usando altri nomi di tipi valore descrivono sempre valori con annidamento finito, si intende cioè evitare una definizione di tipi con annidamento infinito, come nell'esempio seguente:

$$\begin{aligned} T1 &= \{T2\} \\ T2 &= \langle T3 \rangle \\ T3 &= T1 \sqcap T2 \end{aligned}$$

Come per i tipi-valore, chiediamo che la relazione di ereditarietà espressa dalla congiunzione nella definizione di una classe o di un tipo sia ben fondata, cioè non contenga cicli `isa`. Il seguente esempio mostra un insieme di descrizioni *non* ben fondate rispetto all'ereditarietà:

$$\begin{aligned} \texttt{person} &\leq \triangle[\texttt{name} : \texttt{string}] \\ \texttt{employee} &= \texttt{person} \sqcap \texttt{worker} \sqcap \triangle[\texttt{worksin} : \texttt{branch}] \\ \texttt{manager} &= \texttt{employee} \sqcap \triangle[\texttt{head} : \texttt{branch}] \\ \texttt{worker} &= \texttt{manager} \sqcap \texttt{person} \end{aligned}$$

Risulta che `employee` $\preceq_\sigma$ `employee`, cioè abbiamo un ciclo nella relazione di ereditarietà.

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

Schemi ben fondati rispetto ai tipi e all'ereditarietà sono chiamati *ben formati*. Nel seguito si assume sempre che lo schema sia ben–formato.

A.1.4 Uno schema di esempio: il dominio Società

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

Le persone (`person`) hanno un nome; i dipendenti (`employee`) sono persone che lavorano in un'azienda (`branch`), percepiscono un salario e sono inquadrate in un livello. I dirigenti (`manager`) sono persone che lavorano e dirigono

$$
\begin{aligned}
\mathbf{T} &= \{\texttt{activities}, \texttt{level}, \texttt{mdmLevel}, \texttt{advLevel}\} \\
\mathbf{C} &= \{\texttt{person}, \texttt{branch}\} \\
\mathbf{D} &= \{\texttt{manager}, \texttt{clerk}, \texttt{department}, \texttt{sector}, \\
&\quad\ \texttt{employee}, \texttt{secretary}, \texttt{office}\}
\end{aligned}
$$

$$
\begin{aligned}
\sigma(\texttt{level}) &= 1\text{–}10 \\
\sigma(\texttt{mdmLevel}) &= 2\text{–}7 \\
\sigma(\texttt{advLevel}) &= 8\text{–}10 \\
\sigma(\texttt{activities}) &= \{\texttt{string}\} \\
\sigma(\texttt{person}) &= \triangle\Big[\texttt{name}\colon \texttt{string}\Big] \\
\sigma(\texttt{branch}) &= \triangle\Big[\texttt{name}\colon [\texttt{bname}\colon \texttt{string}]\Big] \\
\sigma(\texttt{sector}) &= \triangle\Big[\texttt{name}\colon [\texttt{sname}\colon \texttt{string}], \texttt{activity}\colon \texttt{activities}\Big] \\
\sigma(\texttt{employee}) &= \texttt{person} \sqcap \triangle\Big[\texttt{salary}\colon \texttt{real}, \texttt{worksin}\colon \texttt{branch}, \\
&\qquad\qquad\qquad \texttt{level}\colon \texttt{level}\Big] \\
\sigma(\texttt{manager}) &= \texttt{person} \sqcap \triangle\Big[\texttt{salary}\colon \texttt{real}, \texttt{worksin}\colon \texttt{branch}, \\
&\qquad\qquad\qquad \texttt{head}\colon \texttt{branch}, \texttt{level}\colon \texttt{advLevel}\Big] \\
\sigma(\texttt{clerk}) &= \texttt{employee} \sqcap \triangle\Big[\texttt{worksin}\colon \texttt{department}, \texttt{level}\colon \texttt{mdmLevel}\Big] \\
\sigma(\texttt{department}) &= \texttt{branch} \sqcap \triangle\Big[\texttt{employs}\colon \{\texttt{clerk}\}\Big] \\
\sigma(\texttt{secretary}) &= \texttt{employee} \sqcap \triangle\Big[\texttt{worksin}\colon \texttt{office}, \texttt{level}\colon \texttt{mdmLevel}\Big] \\
\sigma(\texttt{office}) &= \texttt{branch} \sqcap \texttt{sector} \sqcap \triangle\Big[\texttt{employs}\colon \{\texttt{secretary}\}, \\
&\qquad\qquad\qquad\qquad\qquad \texttt{typing}\colon \texttt{secretary}\Big]
\end{aligned}
$$

Tabella A.1: Esempio schema ”Organizzazione Società”

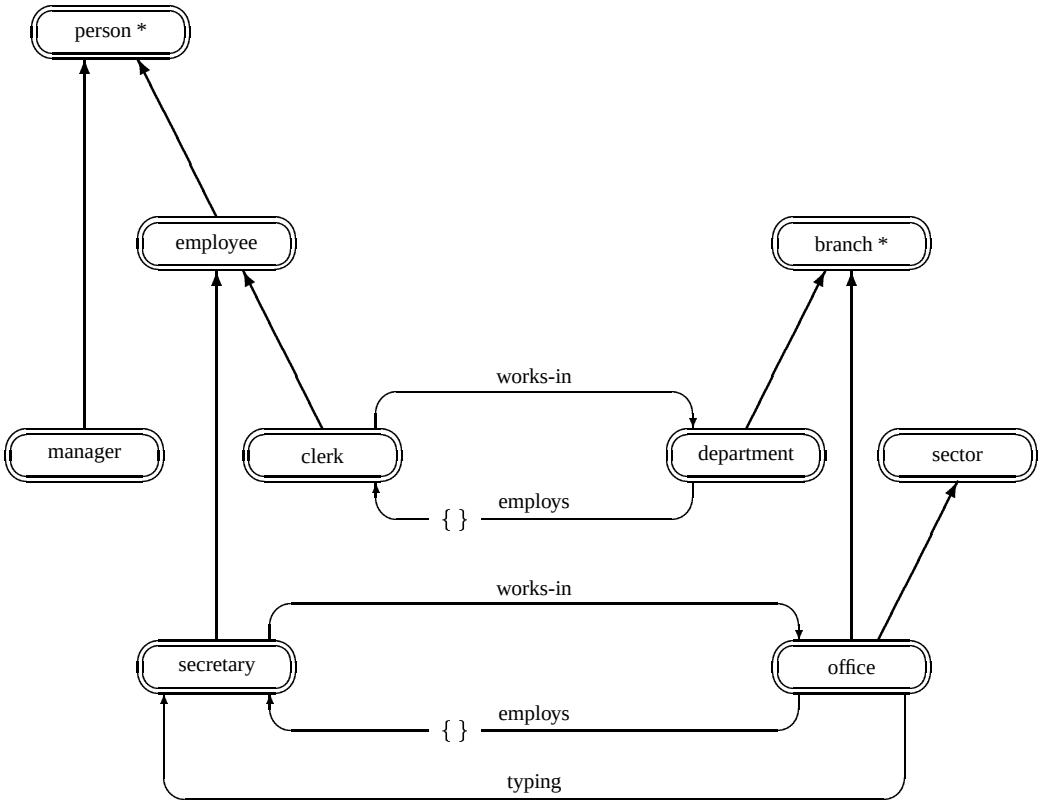


Figura A.1: Gerarchia delle classi relativa alla struttura organizzativa di una società

un'azienda, percepiscono un salario e sono inquadrati in livelli più alti. I settori (`sector`) hanno un nome e un insieme di attività; le aziende hanno un nome. Gli impiegati (`clerk`) sono dipendenti che lavorano in un dipartimento (`department`); i dipartimenti, a loro volta, sono aziende che impiegano esclusivamente impiegati. I segretari (`secretary`) sono dipendenti che lavorano in un ufficio (`office`); gli uffici sono aziende e settori che impiegano esclusivamente segretari e hanno un segretario che ricopre il ruolo di dattilografo.

Uno schema corrispondente a tali descrizioni è mostrato in tabella A.1, dove si è assunto che tutte le classi, ad eccezione delle persone e delle aziende, siano classi virtuali. Si può facilmente verificare che questo schema è ben–formato.

La figura A.1 mostra la relativa gerarchia delle classi; le classi base (o primitive) sono contrassegnate con un asterisco. La figura riporta anche gli attributi che danno origine alle classi virtuali cicliche `clerk`, `department`, `secretary` e `office`; essi sono indicati con archi orientati con in aggiunta il simbolo $\{\ \}$ nel caso in cui gli attributi siano multi–valore.

A.1.5 Istanza legale di uno Schema

In questa sezione viene presentata la definizione di "istanza legale di uno schema" sulla base dei concetti visti finora; più avanti, dopo l'introduzione dei vincoli di integrità, si vedrà la definizione di "istanza legale di uno schema con regole di integrità", che è quella considerata da OCDL.

Sia $\mathcal{I}_{\mathbf{B}}$ si indica la funzione di interpretazione standard (fissata) da $\mathbf{B}$ a $2^{\mathcal{D}}$ tale che per ogni $d \in \mathcal{D}$: $\mathcal{I}_{\mathbf{B}}[d] = \{d\}$.

Definizione 8 (Interpretazione) *Dato un sistema di tipi* $\mathbf{S}$ *e un dominio* δ*, l'*interpretazione

$\mathcal{I}$ di $\mathbf{S}$ *su* δ*, è una funzione da* $\mathbf{S}$ *a* $2^{\mathcal{V}(\mathcal{O})}$ *tale che:*

$$\begin{aligned}
\mathcal{I}[\top] &= \mathcal{V}(\mathcal{O}) \\
\mathcal{I}[B] &= \mathcal{I}_{\mathbf{B}}[B] \\
\mathcal{I}[C] &\subseteq \mathcal{O} \\
\mathcal{I}[D] &\subseteq \mathcal{O} \\
\mathcal{I}[t] &\subseteq \mathcal{V}(\mathcal{O}) - \mathcal{O} \\
\mathcal{I}[\{S\}] &= \Big\{\{v_1, \ldots, v_p\} \Big| v_i \in \mathcal{I}[S], 0 \leq i \leq p\Big\} \\
\mathcal{I}[\langle S\rangle] &= \Big\{\langle v_1, \ldots, v_p\rangle \Big| v_i \in \mathcal{I}[S], 0 \leq i \leq p\Big\} \\
\mathcal{I}\Big[[a_1\colon S_1, \ldots, a_p\colon S_p]\Big] &= \Big\{[a_1\colon v_1, \ldots, a_q\colon v_q] \Big|\ p \leq q, v_i \in \mathcal{I}[S_i], 0 \leq i \leq p, \\
&\qquad\qquad v_j \in \mathcal{V}(\mathcal{O}), p+1 \leq j \leq q\Big\} \\
\mathcal{I}[S \sqcap S'] &= \mathcal{I}[S] \cap \mathcal{I}[S'] \\
\mathcal{I}[\triangle S] &= \Big\{o \in \mathcal{O} \Big| \delta(o) \in \mathcal{I}[S]\Big\}.
\end{aligned}$$

Si nota immediatamente che l'*interpretazione* è funzione che associa ad ogni tipo un insieme di valori. Un'altra osservazione da fare è che per i tipi tupla si adotta una semantica di mondo aperto simile a quella in [**?**]. Ad esempio:

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

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

Definizione 9 (Istanza Legale) *Dato uno schema* σ *su* $\mathbf{S}$ *e un dominio* δ*, un'interpretazione* $\mathcal{I}$ *di* $\mathbf{S}$ *su* δ *è detta* istanza legale *di* σ *su* δ *sse* δ *è* finito *e per ogni* $C \in \mathbf{C}, D \in \mathbf{D}, T \in \mathbf{T}$*:*

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

Con il concetto di istanza possibile di uno schema σ si impone che gli oggetti nell'interpretazione di una classe base C siano un sottoinsieme degli oggetti nell'interpretazione della descrizione della classe $\sigma(C)$. Questo significa che gli oggetti *istanziati* in una classe base sono esplicitamente forniti dall'utente e soddisfano la descrizione della classe stessa. Invece, per i nomi di classi virtuali e per i nomi di tipo–valore l'interpretazione è uguale all'interpretazione della descrizione del nome, cioè per tali nomi N l'interpretazione è calcolata sulla base della loro descrizione $\sigma(N)$.
In altri termini, la descrizione $\sigma(N)$ costituisce un insieme di condizioni *necessarie e sufficienti* per i nomi di tipo $N \in \mathbf{D} \cup \mathbf{T}$, mentre costituisce un insieme di condizioni *solamente necessarie* per i nomi di tipo $N \in \mathbf{C}$.

A.2 Sussunzione e coerenza

In questa sezione definiamo una relazione di inclusione semantica, detta *relazione di sussunzione*, tra i tipi in uno schema, indicata con il simbolo $\sqsubseteq$; essa permette di introdurre la nozione di incoerenza di uno schema e, quindi, quella di coerenza.

Definizione 10 (Sussunzione) *Dato uno schema σ su $\mathbf{S}$ e due tipi $S, S' \in \mathbf{S}$, si definisce la relazione di sussunzione $\sqsubseteq_\sigma$ come segue:*

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

La relazione $\sqsubseteq_\sigma$ è un preordine[1] che induce una relazione di *equivalenza* $\simeq_\sigma$ sui tipi: $S \simeq_\sigma S'$ sse $S \sqsubseteq_\sigma S'$ e $S' \sqsubseteq_\sigma S$. La relazione di equivalenza $\simeq_\sigma$ permette di definire i tipi S che hanno, nello schema σ, un'interpretazione sempre vuota.

Definizione 11 (Incoerenza) *Dato uno schema σ su $\mathbf{S}$, un tipo $S \in \mathbf{S}$ è detto* incoerente *nello schema σ sse $S \simeq_\sigma \bot$.*

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

Proposizione 1 *Dato uno schema σ, siano $N, N' \in \mathbf{N}$; se $N \preceq_\sigma N'$ allora $N \sqsubseteq_\sigma N'$.*

[1] una relazione è un *preordine* se è antisimmetrica e gode delle proprità riflessiva e transitiva

In generale il contrario non vale. La principale ragione è la semantica data ai nomi $N \in \mathbf{D} \cup \mathbf{T}$. Un'altra ragione è che un nome di tipo può essere incoerente. Per schemi coerenti possiamo dare un viceversa parziale alla precedente proposizione.

Proposizione 2 *Dato uno schema coerente σ, sia $N \in \mathbf{C} \cup \mathbf{D}$ e $N' \in \mathbf{C}$; allora $N \sqsubseteq_\sigma N'$ sse $N \preceq_\sigma N'$.*

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

A.3 Schemi e vincoli d'integrità

Gli schemi di basi di dati reali sono, di fatto, forniti in termini di classi base mentre l'ulteriore conoscenza è espressa attraverso vincoli d'integrità che dovrebbero garantire la consistenza dei dati. Per questo motivo è stato sviluppato il modello OCDL che estende i concetti presentati in precedenza relativi ad ODL aggiungendo le nozioni di *quantificatore esistenziale, tipi cammino quantificati* e *regole d'integrità*.

Il quantificatore esistenziale serve per denotare gli insiemi in cui *almeno* un elemento è di un certo tipo. I tipi cammino quantificati vengono introdotti per gestire in modo potente ed agile le strutture nidificate. I cammini, che sono essenzialmente sequenze di attributi, rappresentano l'elemento centrale dei linguaggi d'interrogazione OODB per navigare attraverso le gerarchie di aggregazione di classi e tipi di uno schema. La possibilità di esprimere cammini *quantificati*, permette di navigare attraverso gli attributi multivalore: le quantificazioni ammesse sono quella esistenziale e quella universale e possono comparire più di una volta nello stesso percorso.

Tramite i vincoli di integrità è possibile esprimere correlazioni fra proprietà strutturali della stessa classe o condizioni sufficienti per il popolamento di sottoclassi di una classe data. In particolare, una classe può essere rappresentata tramite una dichiarazione di inclusione dove l'antecedente è il nome della classe e il conseguente la sua descrizione.

A.4 Schemi e regole di integrità

I vincoli (o regole) di integrità sono asserzioni di tipo *if then* che devono essere vere per ogni oggetto di una base di dati e hanno lo scopo di imporre la consistenza di una base di dati.

Nel modello OCDL, come nella maggior parte dei modelli ad oggetti, alcuni vincoli di integrità sono già esprimibili nello schema e vengono imposti agli oggetti tramite la nozione di istanza legale; li possiamo così suddividere:

- *vincoli di dominio*: per ogni attributo è specificato l'insieme dei valori ammissibili;
- *vincoli di integrità referenziale*: un oggetto che è referenziato da un altro oggetto deve esistere;
- *vincoli di ereditarietà*: se un oggetto appartiene ad una classe C allora deve appartenere anche alle superclassi di C.

Nel modello considerato si possono esprimere vincoli di specializzazione [**?**] su una sequenza di attributi senza essere costretti a definire ulteriori sottoclassi.
Ad esempio, con riferimento allo schema di tabella A.1, la sottoclasse `officex` di `office` la cui componente `sname` del `name` è ristretta a "*Database*" e nella quale gli oggetti che ricoprono il ruolo di `typing` hanno l'attributo `level` uguale a 3, può essere descritta nel seguente modo:

$$\sigma(\texttt{officex}) \;=\; \texttt{office} \sqcap \triangle\Big[\texttt{name}\colon [\texttt{sname}\colon \textit{"Database"}],\; \texttt{typing}\colon \triangle\,[\texttt{level}\colon 3]\Big]$$

senza dover introdurre la sottoclasse di `secretary` che ha `level` uguale a 3.
Dato un insieme di attributi $\mathbf{A}$, definiamo un *cammino* p su $\mathbf{A}$ come una sequenza di elementi $p = e_1\,.\,e_2\,.\,\ldots\,.\,e_n$, con $e_i \in \mathbf{A} \cup \{\langle\rangle, \triangle, \forall, \exists\}$. Si denota con ϵ il *cammino vuoto*.
Possiamo ora estendere la definizione di descrizioni di tipi con il quantificatore esistenziale ed il tipo-cammino:

Definizione 12 (Tipi)
Dati gli insiemi $\mathbf{A}$, $\mathbf{B}$ *e* $\mathbf{N}$, *il* sistema di tipi $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$ *denota l'insieme di tutte le* descrizioni dei tipi *(*S, S', *. . .), detti anche brevemente* tipi, *su* $\mathbf{A}, \mathbf{B}, \mathbf{N}$, *che sono costruiti rispettando la seguente regola sintattica astratta (assumendo* $a_i \neq a_j$ *per* $i \neq j$*):*

$$
\begin{array}{rcl}
S & \rightarrow & \top \\
 & & B \\
 & & N \\
 & & \forall\{S\} \\
 & & \exists\{S\} \\
 & & \langle S \rangle \\
 & & [a_1 \colon S_1, \ldots, a_k \colon S_k] \\
 & & S \sqcap S' \\
 & & \triangle S \\
 & & (p : S)
\end{array}
$$

dove p è un cammino su $\mathbf{A}$.

$\forall\{S\}$ corrisponde al comune costruttore di insieme ed indica insiemi i cui elementi sono *tutti* dello stesso tipo S. Invece, il costruttore $\exists\{S\}$ denota un insieme in cui *almeno* un elemento è di tipo S. Il costruttore $(p : S)$, detto *tipo–cammino*, permette di navigare facilmente attraverso la gerarchia di aggregazione mediante sequenze di attributi; questo è un aspetto fondamentale anche nei linguaggi di interrogazione orientati agli oggetti.

La *funzione interpretazione* $\mathcal{I}$ è una funzione da $\mathbf{S}$ a $2^{\mathcal{V}(\mathcal{O})}$ tale che: $\mathcal{I}[\top] = \mathcal{V}(\mathcal{O})$, $\mathcal{I}[B] = \mathcal{I}_{\mathbf{B}}[B]$, $\mathcal{I}[C] \subseteq \mathcal{O}$, $\mathcal{I}[D] \subseteq \mathcal{O}$, $\mathcal{I}[T] \subseteq \mathcal{V}(\mathcal{O}) - \mathcal{O}$. L'interpretazione è estesa agli altri tipi come segue:

$$
\begin{array}{rcl}
\mathcal{I}[[a_1 \colon S_1, \ldots, a_p \colon S_p]] & = & \left\{ [a_1 \colon v_1, \ldots, a_q \colon v_q] \middle| \begin{array}{r} p \leq q, v_i \in \mathcal{I}[S_i], 1 \leq i \leq p, \\ v_j \in \mathcal{V}(\mathcal{O}), p{+}1 \leq j \leq q \end{array} \right\} \\
\mathcal{I}[\forall\{S\}] & = & \left\{ \{v_1, \ldots, v_p\} \middle| v_i \in \mathcal{I}[S], 1 \leq i \leq p \right\} \\
\mathcal{I}[\exists\{S\}] & = & \left\{ \{v_1, \ldots, v_p\} \middle| \exists i, 1 \leq i \leq p, v_i \in \mathcal{I}[S] \right\} \\
\mathcal{I}[\triangle S] & = & \left\{ o \in \mathcal{O} \middle| \delta(o) \in \mathcal{I}[S] \right\} \\
\mathcal{I}[S \sqcap S'] & = & \mathcal{I}[S] \cap \mathcal{I}[S']
\end{array}
$$

Per i tipi cammino abbiamo $\mathcal{I}[(p \colon S)] = \mathcal{I}[(e \colon (p' \colon S))]$ se $p = e.p'$ dove

$$\mathcal{I}[(\epsilon \colon S)] = \mathcal{I}[S],\ \mathcal{I}[(a \colon S)] = \mathcal{I}[[a \colon S]],\ \mathcal{I}[(\triangle \colon S)] = \mathcal{I}[\triangle S],$$

$$\mathcal{I}[(\forall \colon S)] = \mathcal{I}[\forall\{S\}],\ \mathcal{I}[(\exists \colon S)] = \mathcal{I}[\exists\{S\}]$$

Da questa definizione segue immediatamente che il tipo cammino è in effetti una *notazione abbreviata* per le altre espressioni di tipo, cioè valgono le seguenti equivalenze:

$$
\begin{array}{rcl}
(\epsilon\colon S) & \simeq_\sigma & S \\
(a\colon S) & \simeq_\sigma & [a\colon S] \\
(\forall\colon S) & \simeq_\sigma & \forall\{S\} \\
(\exists\colon S) & \simeq_\sigma & \exists\{S\} \\
(\langle\rangle\colon S) & \simeq_\sigma & \langle S\rangle \\
(\triangle\colon S) & \simeq_\sigma & \triangle S
\end{array}
$$

$$
(p\colon S) \simeq_\sigma (e\colon (p'\colon S)) \text{ se } p = e.p'
$$

Ad esempio, il tipo–cammino $(\triangle.\texttt{name}\colon \textit{“Silvano”})$ individua tutti gli oggetti (il primo elemento del cammino è $\triangle$) che hanno un attributo `name` con valore *“Silvano”*; in particolare questi oggetti possono appartenere ad una generica classe che ha l'attributo `name` definito come stringa. Per considerare una determinata classe, ad esempio `employee`, il tipo–cammino deve essere congiunto con il nome della relativa classe: $\texttt{employee} \sqcap (\triangle.\texttt{name}\colon \textit{“Silvano”})$. Nello stesso modo, il tipo–cammino $S = (\triangle.\texttt{managed-by}.\triangle.\texttt{salary}\colon 40 \div 60)$ non impone restrizioni sul dominio dell'attributo `managed-by`; se si considera la sua congiunzione con la classe `storage`, cioè $\texttt{storage} \sqcap S$, allora $\sigma(\texttt{storage})$ impone implicitamente che gli oggetti del dominio di `managed-by` appartengano alla classe `manager`. Inoltre, è possibile imporre esplicitamente una classe come dominio di un attributo nel seguente modo: $S' = (\triangle.\texttt{managed-by}\colon \texttt{tmanager} \sqcap (\triangle.\texttt{salary}\colon 40 \div 60))$.

Ora è possibile definire formalmente la nozione di *regola di integrità* come segue:

Definizione 13 (Regola di Integrità) *Dato un sistema di tipi* $\mathbf{S}$*, una* regola di integrità*, o più semplicemente* regola*,* R *su* $\mathbf{S}$ *è un elemento* (S^a, S^c) *del prodotto cartesiano* $\mathbf{S} \times \mathbf{S}$.

Informalmente, una regola di integrità $R = (S^a, S^c)$ ha lo scopo di vincolare ulteriormente l'istanza legale di uno schema, stabilendo una *relazione inclusione* tra il tipo S^a e il tipo S^c: per ogni valore v, se v è di tipo S^a ($v \in \mathcal{I}[S^a]$) allora v deve essere di tipo S^c ($v \in \mathcal{I}[S^c]$). Una regola di integrità R è quindi universalmente quantificata su tutti i valori $\mathcal{V}(\mathcal{O})$.

Nella regola di integrità $R = (S^a, S^c)$ i tipi S^a e S^c vengono chiamati rispettivamente *antecedente* e *conseguente* della regola e la regola verrà scritta anche nella usuale forma $R = S^a \to S^c$.

Introduciamo ora la nozione di istanza legale di uno schema con regole come un'interpretazione nella quale un valore istanziato in un nome di tipo ha una descrizione corrispondente a quella del nome di tipo stesso e dove sono valide le relazioni di inclusione stabilite tramite le regole.

Definizione 14 (Istanza Legale con Regole) *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *su* $\mathbf{S}$*, un'*istanza $\mathcal{I}$ *di* σ *su* δ *è detta* istanza legale *di* $(\sigma, \mathbf{R})$ *sse per ogni* $R \in \mathbf{R}$*,* $\mathcal{I}[S^a] \subseteq \mathcal{I}[S^c]$.

Si noti che da questa definizione di istanza legale di uno schema con regole segue immediatamente che una descrizione di classe base C con

$$\sigma_{\mathtt{P}}(C) = S$$

può essere espressa in maniera equivalente attraverso la regola

$$R_i : C \rightarrow S$$

Definiamo la nozione di schema con regole di integrità.

Definizione 15 (Schema con Regole di Integrità)
Dato un sistema di tipi $\mathbf{S}$*, uno* schema con regole *su* $\mathbf{S}$ *è una coppia* $(\sigma, \mathbf{R})$*, dove* σ *è uno schema su* $\mathbf{S}$ *e* $\mathbf{R}$ *è un insieme di regole su* $\mathbf{S}$.

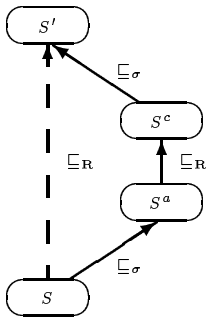
La relazione di sussunzione rispetto ad uno schema con regole è definita rispetto alle istanze legali dello schema con regole come segue:

Definizione 16 (Sussunzione con Regole) *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *su* $\mathbf{S}$*, siano* $S, S' \in \mathbf{S}$*; definiamo la relazione di sussunzione rispetto a* $(\sigma, \mathbf{R})$ *come segue:*

$$S \sqsubseteq_{\mathrm{R}} S' \quad \textit{sse} \quad \mathcal{I}[S] \subseteq \mathcal{I}[S'] \quad \textit{per ogni istanza } \mathcal{I} \textit{ di } (\sigma, \mathbf{R}).$$

La relazione di equivalenza sui tipi indotta da $\sqsubseteq_{\mathrm{R}}$ viene indicata con $\simeq_{\sigma\,\mathrm{R}}$.
È immediato verificare che, per ogni $S, S' \in \mathbf{S}$, se $S \sqsubseteq_\sigma S'$ allora $S \sqsubseteq_{\mathrm{R}} S'$. Il contrario, in generale, non vale in quanto le inclusioni tra le interpretazione dei tipi stabilite tramite le regole fanno sorgere *nuove* relazioni di sussunzione. Intuitivamente, come mostrato dall'esempio di figura A.2, se $S \sqsubseteq_\sigma S^a$ e $S^c \sqsubseteq_\sigma S'$ allora $S \sqsubseteq_{\mathrm{R}} S'$. In tal caso diciamo che la regola R è applicabile a S.
Più precisamente, dato uno schema con regole $(\sigma, \mathbf{R})$ su $\mathbf{S}$, sia $S \in \mathbf{S}$, $R \in \mathbf{R}$ e p un cammino; allora diremo che

R è *applicabile* a S con cammino p se $S \sqsubseteq_\sigma (p : S^a)$.



$S = \texttt{manager} \sqcap (\triangle.\texttt{level}: 5 \div 8)$
$S' = (\triangle.\texttt{salary}: 30 \div 70)$
$R_1 = \texttt{manager} \sqcap (\triangle.\texttt{level}: 5 \div 10)$
$\quad \rightarrow (\triangle.\texttt{salary}: 40 \div 60)$

$S \sqsubseteq_R S', \;\; S \not\sqsubseteq_\sigma S'$

Figura A.2: Relazione di sussunzione forzata mediante una regola.

Dalla semantica delle regole segue immediatamente che dato un tipo S e una regola R applicabile a S, intersecando il tipo S con il conseguente S^c di R si ottiene un tipo che è equivalente a quello originale rispetto allo schema con regole.

Proposizione 3 *Per ogni $S \in \mathbf{S}$ e per ogni $R \in \mathbf{R}$, se R è applicabile a S con il cammino p, allora $S \sqcap (p\colon S^c) \simeq_{\sigma R} S$.*

Naturalmente, al tipo $S \sqcap (p\colon S^c)$ ottenuto tramite l'applicazione della regola R possono essere applicate altre regole che non erano applicabili al tipo originale. Ad esempio, la regola R_3 è applicabile con cammino ϵ al tipo

$$\texttt{storage} \sqcap \triangle[\texttt{category}\colon \texttt{''B4''}, \texttt{managed-by}\colon \triangle[\texttt{level}\colon 4 \div 10]]$$

ottenendo il tipo

$$\texttt{storage} \sqcap \triangle[\texttt{category}\colon \texttt{''B4''}, \texttt{managed-by}\colon \texttt{tmanager} \sqcap \triangle[\texttt{level}\colon 4 \div 10]]$$

al quale è applicabile la regola R_1 con il cammino $\triangle.\texttt{managed-by}$ ottenendo

$$\texttt{storage} \sqcap \triangle[\texttt{category}\colon \texttt{''B4''}, \texttt{managed-by}\colon \texttt{tmanager} \sqcap \triangle[\texttt{level}\colon 4 \div 10, \texttt{salary}\colon 40 \div 60]]$$

Il tipo che si ottiene applicando *tutte* le trasformazioni possibili è detto *espansione semantica* ed è definito nella prossima sezione.

A.5 Espansione semantica di un tipo

L'espansione semantica di un tipo permette di incorporare ogni possibile restrizione che non è presente nel tipo originale ma che è *logicamente implicata* dal tipo e dallo schema. Formalmente questo viene espresso tramite la seguente definizione di espansione semantica:

Definizione 17 (Espansione Semantica) *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *su* $\mathbf{S}$*, e un tipo* $S \in \mathbf{S}$*; l'*espansione semantica *di* S *rispetto a* $(\sigma, \mathbf{R})$*,* $EXP(S)$*, è un tipo di* $\mathbf{S}$ *tale che:*

1. $EXP(S) \simeq_{\sigma \mathrm{R}} S$*;*
2. *per ogni* $S' \in \mathbf{S}$ *tale che* $S' \simeq_{\sigma \mathrm{R}} S$ *si ha che* $EXP(S) \sqsubseteq_\sigma S'$*.*

$EXP(S)$ è il tipo più specializzato tra tutti i tipi $\simeq_{\sigma \mathrm{R}}$–equivalenti al tipo S in quanto include tutte le possibili restrizioni implicate dalle regole $\mathbf{R}$. In questo modo è il tipo più piccolo rispetto alla relazione $\sqsubseteq_\sigma$ tra tutti i tipi $\simeq_{\sigma \mathrm{R}}$–equivalenti a S. Si noti che $EXP(S)$ individua una classe di tipi $\simeq_\sigma$–equivalenti, nella quale ogni elemento è un tipo $\simeq_{\sigma \mathrm{R}}$–equivalente al tipo S.
Il metodo proposto per determinare l'espansione semantica è caratterizzato dai seguenti punti:

- iterazione della trasformazione "se un tipo implica l'antecedente di una regola allora il conseguente di tale regola può essere ad esso congiunto";
- valutazione delle implicazioni logiche tramite il calcolo della sussunzione tra tipi.

Allo scopo di individuare tutte le trasformazioni che uno schema con regole $(\sigma, \mathbf{R})$ induce su un tipo, si introduce la funzione totale $\Gamma : \mathbf{S} \longrightarrow \mathbf{S}$ tale che

$$\Gamma(S) \;=\; \begin{cases} S \sqcap \bigsqcap_k (p_k : S_k^c) & \forall R_k, p_k : \; S \sqsubseteq_\sigma (p_k \colon S_k^a),\; S \not\sqsubseteq_\sigma (p_k \colon S_k^c) \\ S & \text{altrimenti} \end{cases}$$

e si definisce $\tilde{\Gamma} = \Gamma^{\bar{i}}$, dove $\bar{i}$ è il più piccolo intero tale che $\Gamma^{\bar{i}} = \Gamma^{\bar{i}+1}$. L'esistenza di $\bar{i}$ è garantita dal fatto che il numero di regole è finito e una regola non può essere applicata più di una volta con lo stesso cammino grazie al controllo $S \not\sqsubseteq_\sigma (p \colon S^c)$.
Il seguente teorema, dimostrato in [**?**] , assicura la possibilità di utilizzare la funzione $\tilde{\Gamma}(S)$ in luogo di $EXP(S)$:

Teorema 1 *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *su* $\mathbf{S}$*, per ogni* $S \in \mathbf{S}$*,* $EXP(S)$ *può essere effettivamente calcolata tramite* $\tilde{\Gamma}(S)$*.*

Tramite $\tilde{\Gamma}(S)$ possiamo dunque formalizzare il legame esistente tra le relazioni di sussunzione $\sqsubseteq_{\mathrm{R}}$ e $\sqsubseteq_\sigma$.

Corollario 1 *Dato uno schema con regole* $(\sigma, \mathbf{R})$ *su* $\mathbf{S}$*, per ogni* $S, S' \in \mathbf{S}$*, si ha che* $S \sqsubseteq_{\mathrm{R}} S'$ *se e solo se* $\tilde{\Gamma}(S) \sqsubseteq_\sigma S'$*.*

Pertanto, il calcolo della sussunzione in uno schema con regole $(\sigma, \mathbf{R})$ può essere effettuato prima determinando l'espansione semantica di un tipo e quindi calcolando la sussunzione in σ.

Gli algoritmi di calcolo dell'espansione semantica e della sussunzione sono disponibili rispettivamente in [**?**] e in [**?**].

Appendice B

Java: Una panoramica

Java è stato progettato con lo scopo di rendere i programmi il più possibile indipendenti dall'*implementazione*. Java permette di scrivere un programma e di eseguirlo su *qualunque* macchina connessa ad Internet.

Per rendere portabile anche l'eseguibile, un programma Java è compilato in una forma detta *bytecode* che è interpretata dalla *Java Virtual Machine*.

Java, il linguaggio[1], è un linguaggio di programmazione *general-purpose*, multithreaded, basato sulle classi e orientato agli oggetti.

Caratteristica di Java sono le *Applet*. Una *Applet* è un oggetto di una particolare classe di Java che supporta l'interazione con i Browsers[2].

Vediamo una panoramica delle caratteristiche del linguaggio Java tratta da [**?**].

B.1 Java è

Vediamo un elenco delle principali caratteristiche di Java e del linguaggio Java
Semplice

Java è stato scritto in modo da assomigliare il più possibile al C++. È stato fatto anche per rendere il sistema più famigliare, più comprensibile e per accorciare i tempi di apprendimento del nuovo linguaggio. Rispetto al C++, sono state eliminate alcune funzioni complesse, raramente utilizzate come la gestione dell'ereditarietà multipla tra classi. Rispetto al C, non sono più supportati gli header files, il pre-processore, l'aritmetica dei puntatori, strutture, `union` ed array multidimensionali. In Java è stato introdotto un meccanismo per il recupero automatico della

[1] Tratto da [**?**]

[2] Un *Browser* è uno strumento che permette di *navigare* in internet. In pratica è un programma che supporta più protocolli di comunicazione tra cui `HTTP`

memoria non più utilizzata (*garbage collection*), questo riduce i problemi legati all'allocazione-deallocazione della memoria che è spesso causa di problemi.

Piccolo

Java può essere eseguito su macchine con *poca* memoria. Questo è conseguenza del fatto che è *semplice* ed è nato per l'elettronica di consumo.

Object-Oriented

Java non supporta l'ereditarietà multipla, tuttavia supporta le classi astratte.

Esiste il concetto di `interface`. Un interfaccia è una collezione di definizioni di metodi (senza l'*implementazione*) e di costanti. Le interfacce sono usate per definire un protocollo di metodi che devono essere implementati dalle classi.

Le `interfaces` possono essere usate per:

- catturare le similarità tra classi scorrelate;
- dichiarare i metodi che devono essere "implementati" in una o più classi;
- mostrare l'API (programmig interface) di oggetti senza mostrarne la classe. Gli oggetti che fanno questo sono detti *anonymous* e possono essere utili quando si distribuiscono pacchetti di classi ad altri sviluppatori.

Tuttavia non supporta altre funzionalità tipiche dell'ereditarietà multipla:

- dalle *interfaces* non si possono ereditare variabili;
- dalle *interfaces* non si possono ereditare le implementazioni dei metodi, le interface contengono la dichiarazione dei metodi (la *signature*) ma non la definizione (l'implementazione) dei metodi stessi.
- la gerarchia delle interface è indipendente dalla gerarchia delle classi. Le classi che *implementano* la stessa *interface* possono avere o non avere relazioni di ereditarietà.

Distribuito

Grazie alla gestione delle URL, un programma Java ha accesso a dati remoti attraverso i protocolli HTTP e FTP.

Interpretato (e compilato)

In realtà Java è sia compilato che interpretato. Solo il venti per cento del codice Java è realmente interpretato dal Browser. L'abilità di java di girare su piattaforme diverse è legato al fatto che l'ultimo passo di compilazione avviene localmente, sulla macchina su cui il codice deve essere eseguito.

Robusto

Per linguaggio di programmazione “robusto” si intende che i programmi scritti in tale linguaggio difficilmente vanno in crash ed è più probabile che siano privi di “bugs”. I linguaggi fortemente tipati permettono maggiori controlli durante la compilazione, questo solitamente riduce la probabilità di errori a run-time. Un programma Java non può causare un crash di sistema perché il java ha un’autolimitazione sull’uso della memoria.

Sicuro

La sicurezza di Java è ancora da provare. Scaricare del Software dalla rete per eseguirlo sulla propria macchina è sempre pericoloso, ad esempio si corre il rischio di infettare il proprio elaboratore con dei virus.

Altre caratteristiche di Java, come la robustezza, lo rendono sicuro, in realtà Java è per lo più sicuro in quanto è stato progettato per esserlo. Ad esempio, nel *bytecode* sono sempre incluse informazioni per la verifica del codice, per prevenire violazioni della sicurezza.

Tuttavia, anche se viene caricata un’applet “cattiva”, può fare pochi danni in quanto viene eseguita in un ambiente protetto. Questo ambiente non consente di eseguire funzioni pericolose a meno che l’utente finale non lo permetta.

Indipendente dall’architettura

Un’applicazione Java è adatta alla rete se può essere eseguita su sistemi diversi con CPU diverse e con diversa architettura dei sistemi operativi. Questo è possibile grazie al fatto che il compilatore Java genera il file oggetto di formato neutrale detto *bytecode* rispetto l’architettura del sistema.

Una volta scritto il codice, non dev’essere ricompilato per essere portato su altre piattaforme. Il linguaggio Java ed il bytecode è lo stesso su qualunque computer.

Le istruzioni del bytecode sono facili da *interpretare* su qualunque macchina, non sono orientate verso una particolare piattaforma e, a run-time, sono facilmente traducibili nel codice nativo della macchina.

Portabile

Un linguaggio i cui eseguibili sono indipendenti dall’architettura è per sua natura portabile. Di Java è portabile anche l’ambiente di sviluppo. Il compilatore `javac` è scritto in Java e l’ambiente del run-time è scritto in ANSI C.

Ad alte prestazioni

Il bytecode Java può essere compilato in linguaggio sorgente per le singole architetture. Dai test della Sun Microsystem effettuati su SPARCStation 10, risulta che Java è veloce quasi quanto il C++. Tuttavia, oggi, il C++ è mediamente venti volte più veloce di Java interpretato.

Multithreaded

I *thread* possono essere considerati come dei processi che condividono la memoria principale, sono più “leggeri” dei classici processi. Java possiede un sofi-

sticato set di primitive di sincronizzazione integrate nel linguaggio in modo che risultino robuste e facili da usare. Le prestazioni del multithreading di Java dipendono molto dalla piattaforma su cui gira l'eseguibile, gira meglio su macchine multiprocessore.

Dinamico,

Java è dinamico rispetto al C++ o al C in quanto la connessione tra i moduli e le librerie avviene a run-time. Questo consente, ad esempio, di sostituire parte delle librerie con librerie più recenti senza dover ricompilare l'intero programma. Questo rende Java un linguaggio che ben si adatta agli ambienti in continua evoluzione.

B.2 dal Whitepaper "La piattaforma Java"

Vediamo una panoramica tratta dall'articolo

The Java(TM) Platform A White Paper

Per informazioni più dettagliate si può fare riferimento al sito web:

`http://java.sun.com`

B.2.1 Cos'è la piattaforma Java

Attualmente il mondo del computer ha molte *piattaforme* tra cui Microsoft Windows, Macintosh, OS/2, UNIX e NetWare; il software dev'essere compilato separatamente per girare su ogni piattaforma. I files binari che girano su di una piattaforma non possono girare su un'altra piattaforma, poiché il file binario è specifico della macchina.

La *Java Platform* è un nuovo software per distribuire applets interattive, dinamiche e sicure ed applicazioni per sistemi *networked computer*. Ciò che distingue la piattaforma Java è che stà al di sopra delle altre piattaforme, un file binario Java è in un formato che non è specifico per nessuna macchina fisica ma sono istruzioni macchina di una macchina virtuale. Un programma scritto in Linguaggio Java e compilato nel suo *bytecode* può girare ovunque sia presente la piattaforma Java. Questa portabilità è possibile in quanto alla base della *Java Platform* vi è la *Java VirtualMachine*.

Mentre ogni piattaforma ha la propria *implementazione* della *VirtualMachine* vi è una sola specifica per la macchina virtuale. Per questo motivo la *Java Platform* fornisce una interfaccia di programmazione standard ed uniforme per le applets e le applicazioni su qualunque hardware. La *Java Platform* è adatta ad Internet, ove un programma dev'essere in grado di girare su qualunque computer nel mondo.

La *Java Platform* è stata progettata per fornire questa capacità di "Write Once, Run Anywhere".

Gli sviluppatori usano il "Linguaggio Java" per scrivere il codice per le applicazioni Java. Questi compilano una sola volta il codice per la *Java Platform* e non per il sistema operativo che stà sotto. Il codice sorgente viene compilato in una forma intermedia e portabile di *bytecode* che può girare ovunque sia presente la *Java Platform*.

La piattaforma gestisce la sicurezza, le eccezioni ed effettua il *garbage collection* automatico degli oggetti non più utilizzati. Sono disponibili compilatori *Just-in-time* per velocizzare l'esecuzione delle applicazione convertendo il *bytecode* Java in linguaggio macchina. Inoltre, per realizzare funzioni più veloci o speciali, Java permette di chiamare funzioni scritte in C, C++ o altri programmi.

Il linguaggio Java è l'ingresso verso la piattaforma Java. I programmi scritti in linguaggio Java e compilati gireranno sulla Piattaforma Java.

La piattaforma Java è composta da due parti base:

- Java Virtual Machine
- Java Application Programming Interface (Java API)

Combinate, queste parti, forniscono un ambiente agli utenti finali per utilizzare applicazioni Internet e intranet.

B.2.2 Applet e Application

La piattaforma Java permette agli sviluppatori di creare due differenti tipi di programmi:

Le Applets sono programmi che per girare necessitano di un browser. Il grazie al tag `<applet>` è possibile indicare al browser inserire in una pagina HTML il punto cui inserire l'applet ed il nome dell'applet da eseguire. Quando un utente accede a tale pagina, l'applet viene automaticamente scaricata dal server e viene eseguita sulla macchina client.

Le Applications sono programmi che non necessitano di un browser per girare, non hanno un meccanismo per il download automatico. Le applications sono del tutto simili a programmi in altri linguaggi. Possono essere usate per scrivere applicazioni tradizionali come *word processor*, *spreadsheet* o applicazioni grafiche.

Differiscono per il modo in cui sono invocati. Entrambe possono gran parte delle possibilità offerte dal linguaggio. Per esempio entrambe possono accedere ad un database remoto, trovare i dati richiesti, elaborarli localmente e rispedire i risultati alla macchina remota. Tuttavia un applet per girare necessita di un *browser* mentre l'application non ne ha bisogno.

L'Application ha accesso a tutte le risorse della macchina locale. Al contrario di un'applet non può leggere e scrivere files dal filesystem locale.

L'Applet invece ha anche forti restrizioni all'uso delle risorse della macchina su cui gira. Questo è per far fronte ad un problema di sicurezza, infatti le applet sono scaricate dalla rete e non sempre si può sapere a priori che tipo di programma si vada a scaricare (virus, programmi che cercano informazioni riservate o altro...). Normalmente un Applet può leggere files solo dal server da cui è stata scaricata.

Queste restrizioni possono essere rilassate se si prendono le opportune precauzioni, ad esempio, se si utilizza un sistema di firme digitali e si è sicuri che l'applet è stata scaricata inalterata.

B.2.3 Varie piattaforme

Vi sono più ambienti in cui possono girare i programmi Java, tra questi alcuni sono:

La piattaforma base di Java è la minima piattaforma che permetta di eseguire applets e applications. Questa piattaforma è composta dalla Java Virtual Machine più un insieme minimo di API necessario per eseguire le applicazioni e le applet. Le API di questa piattaforma, dette *Java Base API*, mettono a disposizione: le funzioni del linguaggio, di I/O, alcune utility, funzioni per la rete, interfacce grafiche GUI (Graphical User Interface) e le Applets.

La piattaforma Embedded di Java: è pensata per i dispositivi di consumo come stampanti, fotocopiatrici e telefoni cellulari, questi sono caratterizzati dal minori risorse (memoria o display) e funzioni altamente specializzate.

B.2.4 Come si è diffusa la Java Platform

Java si sta *muovendo* dai browser, ai desktop, workstation e network operating systems, fino ai dispositivi embedded.

È stata sviluppata la famiglia di circuiti integrati *JavaChip* in grado di eseguire il bytecode Java in modo nativo. Questo rende disponibile la piattaforma Java per una gran quantità di applicazioni sia *consumer* che industriali, dai Network Computers, stampanti, fotocopiatrici e telefoni cellulari.

B.2.5 JavaOS

JavaOS è un sistema operativo che "implementa" la piattaforma base di Java e permette di far girare applets e applications. È stato progettato per i *Network Computers* ed altri dispositivi di consumo.

Caratteristiche di questo sistema sono: non hanno bisogno di setup di installazione ne di system administator, sono attivi subito dopo l'installazione e, se connessi alla rete possono eseguire automaticamente l'update del sistema.

Questo sistema sarà portato su molti processori, compresa la famiglia JavaChip.

B.2.6 Ancora sul linguaggio Java

Il codice scritto in linguaggio Java può chiamare API definite nelle *Java Base API* oppure nelle *Java Standard Extensions API* o nuove API definite nel codice sorgente. A durante l'esecuzione i tre tipi di API sono indistinguibili, vengono trattate tutte nello stesso modo.

Le *Java Standard Extensions API* estendono le capacità del linguaggio. Come si vede in tabella B.1 attualmente ve ne sono di due tipi: quelle che entreranno tra le API standard e quelle che rimarranno estensioni.

API che saranno Base	**API che rimarrano Extensions**
Java 2D	Java 3D
Audio	Video , MIDI
Java Media Framework	Java Share
Java Animation	Java Telephony
Java Enterprise	Java Server
Java Commerce	Java Management
Java Security	

Tabella B.1: tipi di *Java Standard Extensions API*

Java 2D API Mette a disposizione strumenti per la gestione della grafica e di immagini. Questi strumenti sono quelli usati per la creazione delle applets.

Java Media Framework API Permette la gestione di strumenti come audio, video e MIDI. Fornisce un modello comune per la temporizzazione, sincronizzazione e composizione. È stato progettato per la gestione di dati in tempo reale oppure memorizzati come compressi o grezzi.

Video API Favorisce la gestione di sorgenti video sia in tempo reale che da brani registrati. Definisce i formati base dei dati e le interfacce di controllo.

Audio API Supporta l'audio campionato e sintetizzato. Favorisce la gestisce di sorgenti audio sia in tempo reale che da brani registrati.

MIDI API Forniscono il supporto per flussi di dati temporizzati. Utilizza le API Media Framework per la sincronizzazione con altre attività.

Java Animation API Supporta le tradizionali animazioni di sprites 2D.

Java Share API Fornisce le astrazioni fondamentali per la comunicazione in tempo reale, bidirezionale, tra più oggetti su diverse reti e protocolli di trasporto.

Java Telephony API Create per la gestione integrata di computer e telefonia. Fornisce le funzionalità di base per effettuare chiamate, rispondere alle chiamate, gestione teleconferenze, trasferimento di chiamata e codifica/decodifica del DTMF.

Java 3D API Fornisce un supporto grafico 3D ad alte prestazioni ed interattivo. Supporta VRLM (Virtual Reality Modeling Language). Permette la specifica dei comportamenti e di controllo per gli oggetti 3D ad alto livello.

Java Enterprise API Le classi Enterprise permettono di utilizzare applicazioni Java assieme alle risorse informative "aziendali". Tra le possibili soluzioni, citiamo `JDBC` che permette di connettere database ODBC (Open DataBase Connectivity) alle applicazioni Java, e RMI (Remote Method Invocation) che permette ad esempio ad un'applet *client* di invocare metodi su di un'altra macchina virtuale su cui gira la parte server dell'applicazione.

Java Commerce API Queste API sono nate per portare sul Web funzioni che permettano il commercio e la gestione finanziaria in modo sicuro.

Java Server API Si tratta di una struttura che permette un facile sviluppo di server Interet ed intranet in Java. L'API contiene librerie per la gestione del server, il controllo degli accessi e la gestione delle risorse del server dinamica.

Anche le *Servlet API* fanno parte delle *Server API*. Si tratta di oggetti Java, indipendenti dalla piattaforma che realizzano la parte "server" della comunicazione

con le applet. Possono essere memorizzate sul server oppure scaricate dal server dalla rete a patto che sia garantita una certa sicurezza del trasferimento. Esistono esempi di Servlet che spaziano da servlets HTTP, che possono sostituire efficientemente le CGI, a Servlet più complesse che permettono di utilizzare database JDBC/ODBC.

Java Management API Si tratta di una collezione di classi Java che mettono a disposizione "blocchi" per la *gestione integrata*. In pratica vengono fornite interfaces, classi, applets e linee guida per facilitare lo sviluppo di soluzioni per la gestione integrata.
Le API *Management* sono composte da diversi e distinti componenti, ognuno con uno specifico compito:

Admin View Module Si tratta di un'estensione dell' AWT (Java Abstract Window Toolkit).

Base Object Interfaces Si tratta di un supporto per la creazione di oggetti che rappresentano risorse distribuite e servizi che caratterizzano le risorse informatiche delle imprese (ambiente *enterprise*).

Managed Notification Interfaces fornisce gli strumenti di base per realizzare servizi di tipo evento-gestione. Il modello prevede la notifica asincrona tra oggetti gestiti e applicazioni che li gestiscono.

Managed Container Interfaces Permette suddividere gli oggetti gestiti in gruppi in modo che possano essere eseguite *azioni* su gruppi invece che sui singoli oggetti. Sono di due tipi: EXTENSIONAL, in cui i gruppi sono costituiti indicando quali oggetti aggiungere o cancellare, e INTENTIONAL, che memorizza solo le caratteristiche che devono avere gli oggetti (memorizza la query), da queste si riempie il gruppo.

Managed Data Interfaces Permettono di mappare gli attributi di database relazionali. Sono un sottoinsieme delle API JDBC (Java Database Connectivity).

Managed Protocol Interfaces Realizzano funzionalità per l'elaborazione distribuita e per la *sicurezza* della comunicazione.

SNMP Interfaces Estendono Permettono di gestire agenti SNMP (Simple Network Management Protocol). Sono un'estensione delle API *Managed Protocol Interfaces*, rendono disponibili le informazioni SNP a tutti gli utenti delle API *Management*.

Java Management API User Interface Style Guide fornisce le linee guide per lo sviluppo di interfacce, per la configurazione e per risolvere eventuali problemi che riguardano sistemi, reti ed elementi di servizio che costituiscono l'infrastruttura per il calcolo.

Appendice C

Lex & Yacc

Lex e *Yacc* sono due utility molto usate per la realizzazione di analizzatori sintattici. Di seguito è riportata una breve descrizione dei due programmi.

In realtà in questa tesi sono stati utilizzati altri due programmi *flex&bison*, diversi ma compatibili con Lec&Yacc.

Questa sezione è stata scritta con la sola intenzione di dare un'idea del funzionamento di tali strumenti in modo che

Ulteriori informazioni: si possono reperire in rete (GNU GENERAL PUBLIC LICENSE) la documentazione di flex&bison. Il libro di riferimento per Lex&Yacc è *Lex&Yacc[**?**]*.

lex o flex *Lex* genera un programma C che svolge una semplice analisi sintattica di un testo. *Felx* è una versione di Lex di pubblico dominio sviluppata nel progetto *gnu*.

Lex è uno strumento per la generazione di "scanners". Gli scanners sono programmi che riconoscono particolari parole in un testo. Lex legge la descrizione dello scanner da generare da un file di input. La descrizione è formata da varie coppie di espressioni regolari e codice C dette *regole*. Lex genera un file sorgente C che dev'essere *linkato* con una opportuna libreria per poter generare l'eseguibile. L'eseguibile, quando lanciato, cerca nel file di input le espressioni regolari definite dalle rules, una regola è riconosciuta allora ne viene eseguita la parte C.

Nella parte Lex di un parser vengono specificate quali sono i "token" del linguaggio. I tocken sono i costrutti di base come le parole chiave, il modo di esprimere i valori costanti ed altre caratteristiche tipiche del linguaggio da controllare sintatticamente. La parte Lex di un programma si preoccupa di riconoscere e discriminare i token del linguaggio.

Yacc o Bison *Yacc*[1] (Yet Another Compiler Compiler), è il generatore di parser che si trova sulla maggior parte di sistemi UNIX. Inoltre, yacc è anche il linguaggio usato per descrivere a yacc (il programma) la sintassi di altri linguaggi. *Bison* è una versione yacc di pubblico dominio sviluppata nel progetto *gnu*.

Bison è un generatore i parsers "general-purpose" che converte una descrizione di una grammatica non contestuale LALR (Look Ahead Left Recursive) in un programma C in grado di analizzare sintatticamente tale grammatica.

Un file yacc ha la seguente struttura:

`DICHIARAZIONI C` contiene definizioni di macro e dichiarazioni di funzioni e variabili che sono usate nelle *actions* delle regole grammaticali (*rules*). Queste sono copiate così come sono all'inizio del programma C generato.

`DICHIARAZIONI YACC` contiene dichiarazioni che definiscono simboli terminali e non terminali, specificano le precedenze e così via.

`REGOLE GRAMMATICALI` contiene una o più *rule*, regole grammaticali, e nient'altro.

`CODICE C` contiene del codice che è copiato così com'è alla fine del programma parser.

Una *rule* definisce come dev'essere un simbolo non terminale, ha la seguente forma:

```
RESULT: COMPONENTS [ | COMPONENTS...]  [ ACTION ];
```

Ove `RESULT` è il simbolo non terminale descritto dalla rule, e COMPONENTS sono i vari simboli terminali o non terminali e `ACTION` è la parte di codice C da eseguire nel caso in cui il parser riconosca il simbolo `RESULT`.

Ad esempio:

```
exp:  exp '+' exp ;
```

[1] da FOLDOC[**?**]

Appendice D

Sintassi ODL

In questa appendice si trova la sintassi ODL riconosciuta dal traduttore. È espressa in formato Yacc, è stata estratta direttamente dal sorgente del programma.

```
%token <String_Type>
        Identifier
        String
        IntegerLiteral
        StringLiteral
        CharacterLiteral
        FloatingPtLiteral
%token <CommandType>
        SEMI
        LPAR
        RPAR
        MODULE
        DOUBLE_COLON
        COLON
        PERSISTENT
        TRANSIENT
        INTERFACE
        VIEW
        LRPAR
        RRPAR
        EXTENT
        KEY
        KEYS
        COMMA
        CONST
        EQUAL
        GREATEQUAL
        LESSEQUAL
        VERT
        HAT
        AMPER
        DOUBLE_RIGHT
        DOUBLE_LEFT
        PLUS
        MINUS
        TIMES
        SLASH
        PERCENT
        TILDE
        TRUE
        FALSE
        TYPEDEF
        FLOAT
        DOUBLE
        INT
        LONG
        SHORT
        UNSIGNED
        CHAR
        BOOLEAN
        ANY
        OCTET
        STRUCT
        UNION
        SWITCH
        CASE
        DEFAULT
        ENUM
        ARRAY
        LEFT
```

```
        RIGHT
        STRING
        LEPAR
        REPAR
        READONLY
        ATTRIBUTE
        SET
        LIST
        BAG
        INVERSE
        RELATIONSHIP
        ORDER_BY
        EXCEPTION
        ONEWAY
        VOID
        IN
        OUT
        INOUT
        RAISES
        CONTEXT
        INFINITE
        RANGE
        AND
        RULE
        THEN
        FOR
        ALL
        FORALL
        EXISTS
        DOT

%type <CommandType>
        ForAll

%type <Definition_list>
        Specification
        Definition
        Module

%type <Interface_type>
        Interface
        InterfaceDcl
        ForwardDcl

%type <Iner_list>
        InheritanceSpec

%type <Prop_list>
        OptInterfaceBody
        InterfaceBody
        Export

%type <Rela_type>
        RelDcl

%type <Attr_type>
        AttrDcl

%type <String_Type>
        AttributeName
        ScopedName
        BooleanLiteral
        ConstExp
        OrExpr
        AndExpr
        ShiftExpr
        AddExpr
        XOrExpr
        MultExpr
        UnaryExpr
        PrimaryExpr
        Literal
        PositiveIntConst
        EnumType
        TypeSpec
        SimpleTypeSpec
        CollectionType
        AttrCollectionSpecifier
        DomainType
        BaseTypeSpec
        FloatingPtType
        IntegerType
        SignedInt
        SignedLongInt
        SignedShortInt
        UnsignedInt
        UnsignedLongInt
        UnsignedShortInt
        SignedFloatingPtLiteral
        SignedIntegerLiteral
        IntegerValue
        LiteralValue
        CharType
        BooleanType
        OctetType
        AnyType
        TemplateTypeSpec
        ArrayType
```

```
        StringType
        ConstrTypeSpec
        StructType
        UnionType
        ArraySizeList
        FixedArraySize
        TraversalPathName1
        TraversalPathName2
        ConstType
        RangeType
        RangeSpecifier
        RuleConstOp
        RuleCast
        DottedName


%type <Character_Type>
        UnaryOperator
        OptOrderBy
        Signes

%type <Enum_enumerators_list>
        Enumerator
        EnumeratorList

%type <Struct_member_list>
        Member
        MemberList

%type <Declarator_type>
        Declarator
        SimpleDeclarator
        ComplexDeclarator
        ArrayDeclarator

%type <Declarator_list>
        Declarators

%type <Rela_target_type>
        TargetOfPath

%type <Rela_inverse_type>
        InverseTraversalPath

%type <Type_list>
        TypeDcl
        TypeDeclarator

%type <Const_type>
        ConstDcl

%type <Rule_type>
        RuleDcl

%type <Rule_body_list>
        RuleAntecedente
        RuleConseguente
        RuleBodyList
        RuleBody

%%

Odl_syntax:
        Specification

Specification:
        Definition
                        |
        Definition
        Specification

Definition:
        TypeDcl
        SEMI
                        |
        ConstDcl
        SEMI
                        |
        ExceptDcl
        SEMI
                        |
        Interface
        SEMI
                        |
        RuleDcl
        SEMI
                        |
        Module
        SEMI
                        |
        error
        SEMI

Module:
        MODULE
        Identifier
        LPAR
        Specification
```

```
        RPAR

Interface:
        InterfaceDcl
                  |
        ForwardDcl

InterfaceDcl:
        INTERFACE
        Identifier
        COLON
        InheritanceSpec
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR
                  |
        INTERFACE
        Identifier
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR
                  |
        VIEW
        Identifier
        COLON
        InheritanceSpec
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR
                  |
        VIEW
        Identifier
        OptTypePropertyList
        OptPersistenceDcl
        LPAR
        OptInterfaceBody
        RPAR

ForwardDcl:
        INTERFACE
        Identifier
```

```
OptInterfaceBody:
        InterfaceBody


OptPersistenceDcl:
        |
        PersistenceDcl

PersistenceDcl:
        PERSISTENT
                  |
        TRANSIENT

OptTypePropertyList:
        |
        TypePropertyList

TypePropertyList:
        LRPAR
        OptExtentSpec
        OptKeySpec
        RRPAR

OptExtentSpec:
                  |
        ExtentSpec

ExtentSpec:
        EXTENT
        Identifier

OptKeySpec:
                  |
        KeySpec

KeySpec:
        KEY
        KeyList
                  |
        KEYS
        KeyList

KeyList:
        Key
        |
        Key
        COMMA
        KeyList
```

```
Key:
        PropertyName
                    |
        LRPAR
        PropertyList
        RRPAR

PropertyList:
        PropertyName
        |
        PropertyName
        COMMA
        PropertyList

PropertyName:
        Identifier

InterfaceBody:
        Export
                    |
        Export
        InterfaceBody

Export:
        TypeDcl
        SEMI
                    |
        ConstDcl
        SEMI
                    |
        ExceptDcl
        SEMI
                    |
        AttrDcl
        SEMI
                    |
        RelDcl
        SEMI
                    |
        OpDcl
        SEMI
                    |
        error
        SEMI

InheritanceSpec:
        ScopedName
                    |
        ScopedName
        COMMA
        InheritanceSpec

ScopedName:
        Identifier
                        |
        DOUBLE_COLON
        Identifier
                        |
        ScopedName
        DOUBLE_COLON
        Identifier

ConstDcl:
        CONST
        STRING
        Identifier
        EQUAL
        StringLiteral
                |
        CONST
        CharType
        Identifier
        EQUAL
        CharacterLiteral
                |
        CONST
        IntegerType
        Identifier
        EQUAL
        SignedIntegerLiteral
                |
        CONST
        FloatingPtType
        Identifier
        EQUAL
        SignedFloatingPtLiteral

SignedIntegerLiteral:
        Signes
        IntegerLiteral
                        |
        IntegerLiteral



SignedFloatingPtLiteral:
        Signes
        FloatingPtLiteral
```

```
                    |
        FloatingPtLiteral

Signes:
        MINUS
                    |
        PLUS

ConstType:
        IntegerType
                    |
        CharType
                    |
        BooleanType
                    |
        FloatingPtType
                    |
        StringType
                    |
        ScopedName

ConstExp:
        OrExpr


OrExpr:
        XOrExpr
                    |
        OrExpr
        VERT
        XOrExpr

XOrExpr:
        AndExpr
                    |
        XOrExpr
        HAT
        AndExpr

AndExpr:
        ShiftExpr
                    |
        AndExpr
        AMPER
        ShiftExpr

ShiftExpr:
        AddExpr
                    |
        ShiftExpr
        DOUBLE_RIGHT
        AddExpr
                    |
        ShiftExpr
        DOUBLE_LEFT
        AddExpr

AddExpr:
        MultExpr
                    |
        AddExpr
        PLUS
        MultExpr
                    |
        AddExpr
        MINUS
        MultExpr

MultExpr:
        UnaryExpr
                    |
        MultExpr
        TIMES
        UnaryExpr
                    |
        MultExpr
        SLASH
        UnaryExpr
                    |
        MultExpr
        PERCENT
        UnaryExpr

UnaryExpr:
        UnaryOperator
        PrimaryExpr
                    |
        PrimaryExpr

UnaryOperator:
        MINUS
                    |
        PLUS
                    |
        TILDE

PrimaryExpr:
        ScopedName
```

```
                |
        Literal
                |
        LRPAR
        ConstExp
        RRPAR
                |
        LRPAR
        error
        RRPAR

Literal:
        IntegerLiteral
                |
        StringLiteral
                |
        CharacterLiteral
                |
        FloatingPtLiteral
                |
        BooleanLiteral

BooleanLiteral:
        TRUE
                |
        FALSE

PositiveIntConst:
        ConstExp

TypeDcl:
        TYPEDEF
        TypeDeclarator
                |
        StructType
                |
        UnionType
                |
        EnumType

TypeDeclarator:
        SimpleTypeSpec
        Declarators
                |
        ConstrTypeSpec
        Declarators

TypeSpec:
        SimpleTypeSpec
                |
        ConstrTypeSpec

SimpleTypeSpec:
        BaseTypeSpec
                |
        TemplateTypeSpec
                |
        ScopedName

BaseTypeSpec:
        FloatingPtType
                |
        IntegerType
                |
        CharType
                |
        BooleanType
                |
        OctetType
                |
        RangeType
                |
        AnyType


TemplateTypeSpec:
        ArrayType
                |
        StringType
                |
        CollectionType

ConstrTypeSpec:
        StructType
        |
        UnionType
        |
        EnumType

Declarators:
        Declarator
                |
        Declarator
        COMMA
        Declarators

Declarator:
        SimpleDeclarator
```

```
                    |
        ComplexDeclarator

SimpleDeclarator:
        Identifier

ComplexDeclarator:
        ArrayDeclarator

FloatingPtType:
        FLOAT
                    |
        DOUBLE
           RangeType:
        RANGE
        LPAR
        RangeSpecifier
        RPAR

RangeSpecifier:
        IntegerValue
        COMMA
        IntegerValue
                    |
        IntegerValue
        COMMA
        PLUS
        INFINITE
                    |
        MINUS
        INFINITE
        COMMA
        IntegerValue

IntegerValue:
        SignedIntegerLiteral
                    |
        Identifier


IntegerType:
        INT
                    |
        SignedInt
                    |
        UnsignedInt

SignedInt:
        SignedLongInt
                    |
        SignedShortInt

SignedLongInt:
        LONG

SignedShortInt:
        SHORT

UnsignedInt:
        UnsignedLongInt
                    |
        UnsignedShortInt

UnsignedLongInt:
        UNSIGNED
        LONG

UnsignedShortInt:
        UNSIGNED
        SHORT

CharType:
        CHAR

BooleanType:
        BOOLEAN

OctetType:
        OCTET

AnyType:
        ANY

StructType:
        STRUCT
        Identifier
        LPAR
        MemberList
        RPAR

MemberList:
        Member
                    |
        Member
        MemberList

Member:
        TypeSpec
```

```
        Declarators
        SEMI

UnionType:
        UNION
        Identifier
        SWITCH
        LRPAR
        SwitchTypeSpec
        RRPAR
        LPAR
        SwitchBody
        RPAR

SwitchTypeSpec:
        IntegerType
                  |
        CharType
                  |
        BooleanType
                  |
        EnumType
                  |
        ScopedName
                  |
        RangeType

SwitchBody:
        Case
        |
        Case
        SwitchBody

Case:   CaseLabelList
        ElementSpec
        SEMI

CaseLabelList:
        CaseLabel
        |
        CaseLabel
        CaseLabelList

CaseLabel:
        CASE
        ConstExp
        COLON
                  |
        DEFAULT
        COLON

ElementSpec:
        TypeSpec
        Declarator

EnumType:
        ENUM
        Identifier
        LPAR
        EnumeratorList
        RPAR

EnumeratorList:
        Enumerator
                  |
        Enumerator
        COMMA
        EnumeratorList

Enumerator:
        Identifier

ArrayType:
        ARRAY
        LEFT
        SimpleTypeSpec
        COMMA
        PositiveIntConst
        RIGHT
                  |
        ARRAY
        LEFT
        SimpleTypeSpec
        RIGHT

StringType:
        STRING
        LEFT
        PositiveIntConst
        RIGHT
                  |
        STRING

ArrayDeclarator:
        Identifier
        ArraySizeList

ArraySizeList:
```

```
        FixedArraySize
                    |
        FixedArraySize
        ArraySizeList

FixedArraySize:
        LEPAR
        PositiveIntConst
        REPAR

AttrDcl:
        READONLY
        ATTRIBUTE
        DomainType
        FixedArraySize
        AttributeName
                    |
        ATTRIBUTE
        DomainType
        FixedArraySize
        AttributeName
                    |
        READONLY
        ATTRIBUTE
        DomainType
        AttributeName
                    |
        ATTRIBUTE
        DomainType
        AttributeName

AttributeName:
        Identifier

DomainType:
        SimpleTypeSpec
                    |
        StructType
                    |
        EnumType

CollectionType:
        AttrCollectionSpecifier
        LEFT
        Literal
        RIGHT
                    |
        AttrCollectionSpecifier
        LEFT
        Identifier
        RIGHT
                    |
        AttrCollectionSpecifier
        LEFT
        SimpleTypeSpec
        RIGHT

AttrCollectionSpecifier:
        SET
                    |
        LIST
                    |
        BAG
                    |
        ARRAY

RelDcl:
        RELATIONSHIP
        TargetOfPath
        TraversalPathName1
        INVERSE
        InverseTraversalPath
        OptOrderBy

OptOrderBy:
        LPAR
        ORDER_BY
        AttributeList
        RPAR
                    |

TraversalPathName1:
        Identifier

TargetOfPath:
        Identifier
                    |
        RelCollectionType
        LEFT
        Identifier
        RIGHT

InverseTraversalPath:
        Identifier
        DOUBLE_COLON
        TraversalPathName2

TraversalPathName2:
```

```
        Identifier

AttributeList:
        ScopedName
                    |
        ScopedName
        COMMA
        AttributeList

RelCollectionType:
        SET
                    |
        LIST

ExceptDcl:
        EXCEPTION
        Identifier
        LPAR
        OptMemberList
        RPAR

OptMemberList:
        |
        MemberList

OpDcl:
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
        ContextExpr
                    |
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
        ContextExpr
                    |
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
        ContextExpr
                    |
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
                    |
        OpTypeSpec
        Identifier
        ParameterDcls
        ContextExpr
                    |
        OpAttribute
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
                    |
        OpTypeSpec
        Identifier
        ParameterDcls
        RaisesExpr
                    |
        OpTypeSpec
        Identifier
        ParameterDcls

OpAttribute:
        ONEWAY

OpTypeSpec:
        SimpleTypeSpec
                    |
        VOID

ParameterDcls:
        LRPAR
        ParamDclList
        RRPAR
                    |
        LRPAR
        RRPAR

ParamDclList:
        ParamDcl
        |
        ParamDcl
        COMMA
        ParamDclList

ParamDcl:
        ParamAttribute
        SimpleTypeSpec
        Declarator
                    |
```

```
        ParamAttribute
        AttrCollectionSpecifier
        LEFT
        SimpleTypeSpec
        RIGHT
        Declarator

ParamAttribute:
        IN
                  |
        OUT
                  |
        INOUT

RaisesExpr:
        RAISES
        LRPAR
        ScopedNameList
        RRPAR

ScopedNameList:
        ScopedName
                  |
        ScopedName
                 COMMA
        ScopedNameList

ContextExpr:
        CONTEXT
        LRPAR
        StringLiteralList
        RRPAR

StringLiteralList:
        StringLiteral
                  |
        StringLiteral
        COMMA
        StringLiteralList

RuleDcl:
        RULE
        Identifier
        RuleAntecedente
        THEN
        RuleConseguente

RuleAntecedente:
        ForAll
        Identifier
        IN
        Identifier
        COLON
        RuleBodyList

RuleConseguente:
        RuleBodyList

RuleBodyList:
        LRPAR
        RuleBodyList
        RRPAR
                  |
        RuleBody
                  |
        RuleBodyList
        AND
        RuleBody
                  |
        RuleBodyList
        AND
        LRPAR
        RuleBodyList
        RRPAR

LiteralValue:
        SignedFloatingPtLiteral
        |
        SignedIntegerLiteral
        |
        CharacterLiteral
        |
        StringLiteral
        |
        Identifier

RuleBody:
        DottedName
        RuleConstOp
        LiteralValue
                  |
        DottedName
        RuleConstOp
        RuleCast
        LiteralValue
                  |
        DottedName
        IN
```

```
        SimpleTypeSpec
                |
        ForAll
        Identifier
        IN
        DottedName
        COLON
        RuleBodyList
                |
        EXISTS
        Identifier
        IN
        DottedName
        COLON
        RuleBodyList

RuleConstOp:
        EQUAL
                |
        GREATEQUAL
                |
        LESSEQUAL
                |
        LEFT
                |
        RIGHT


RuleCast:
        LRPAR
        SimpleTypeSpec
        RRPAR

DottedName:
        Identifier
                |
        Identifier
        DOT
        DottedName

ForAll:
        FOR
        ALL
        |
        FORALL
```

Appendice E

Sintassi OCDL

In questa sezione è riportata l'ultima una versione disponibile della sintassi[1] dell'OCDL riconosciuta dal validatore.

$$
\begin{array}{rcl}
< \textit{linguaggio} >< \text{linguaggio} > & ::= & < \text{def-term} > \ | \\
 & & < \text{linguaggio} >< \text{def-term} > \\
< \text{def-term} > & ::= & < \text{def-tipovalore} > \ | \\
 & & < \text{def-classe} > \ | \\
 & & < \text{def-regola} > \\
< \text{def-tipovalore} > & ::= & < \text{def-tipobase} > \ | \\
 & & < \text{def-tipo} > \\
< \text{def-classe} > & ::= & < \text{def-classe-prim} > \ | \\
 & & < \text{def-classe-virt} > \\
< \text{def-regola} > & ::= & < \text{def-antconV} > \\
 & & < \textit{nome regola} > = < \text{classe} > \ | \\
 & & < \text{def-antconT} > \\
 & & < \textit{nome regola} > = < \text{def-tipoT} > \\
< \text{def-antconV} > & ::= & \textbf{antev} \ | \ \textbf{consv} \\
< \text{def-antconT} > & ::= & \textbf{antet} \ | \ \textbf{const} \\
< \text{def-tipoT} > & ::= & < \text{tipo} > \ | \ < \text{tipobase} > \\
< \text{def-tipo} > & ::= & \mathbf{type} < \textit{nome tipovalore} > = \ < \text{tipo} >
\end{array}
$$

[1] il validatore è parte di un progetto in fase di sviluppo ed è normale che venga modificata.

$$
\begin{array}{rcl}
< \text{def-classe-prim} > & ::= & \mathbf{prim} < \textit{nome classe} > \; = \; < \text{classe} > \\
< \text{def-classe-virt} > & ::= & \mathbf{virt} < \textit{nome classe} > \; = \; < \text{classe} > \\
< \text{def-antecedente} > & ::= & < \text{tipobase} > \; | \\
 & & < \text{tipo} > \; | \\
 & & < \text{classe} > \\
< \text{def-conseguente} > & ::= & < \text{tipobase} > \; | \\
 & & < \text{tipo} > \; | \\
 & & < \text{classe} > \\
< \text{tipo} > & ::= & \#\mathbf{top}\# \; | \\
 & & < \text{insiemi-di-tipi} > \; | \\
 & & < \text{esiste-insiemi-di-tipi} > \; | \\
 & & < \text{sequenze-di-tipi} > \; | \\
 & & < \text{ennuple} > \; | \\
 & & < \text{nomi-di-tipi} > \; | \\
 & & < \text{tipo-cammino} > \\
< \text{classe} > & ::= & < \text{insiemi-di-classi} > \; | \\
 & & < \text{esiste-insiemi-di-classi} > \; | \\
 & & < \text{sequenze-di-classi} > \; | \\
 & & < \text{nomi-di-classi} > \; | \\
 & & \hat{} < \text{tipo} > \; | \\
 & & < \text{nomi-di-classi} > \; \& \; \hat{} < \text{tipo} > \\
< \text{insiemi-di-tipi} > & ::= & \{ < \text{tipo} > \} \; | \\
 & & \{ < \text{tipo} > \} \; \& \; < \text{insiemi-di-tipi} > \; | \\
 & & \{ < \text{tipobase} > \} \\
< \text{esiste-insiemi-di-tipi} > & ::= & !\{ < \text{tipo} > \} \; | \\
 & & !\{ < \text{tipo} > \} \; \& \; < \text{esiste-insiemi-di-tipi} > \; | \\
 & & !\{ < \text{tipobase} > \} \\
< \text{sequenze-di-tipi} > & ::= & \langle \; < \text{tipo} > \; \rangle \; | \\
 & & \langle \; < \text{tipo} > \; \rangle \; \& \; < \text{sequenze-di-tipi} > \; | \\
 & & \langle \; < \text{tipobase} > \; \rangle \\
< \text{ennuple} > & ::= & [< \text{attributi} >] \; | \\
 & & [< \text{attributi} >] \; \& \; < \text{ennuple} > \\
< \text{attributi} > & ::= & < \textit{nome attributo} > : < \text{desc-att} > \; | \\
 & & < \textit{nome attributo} > : < \text{desc-att} > , \; < \text{attributi} >
\end{array}
$$

$$
\begin{array}{rcl}
< \text{desc-att} > & ::= & < \text{tipobase} > \ | \\
 & & < \text{tipo} > \ | \\
 & & < \text{classe} > \\
< \text{nomi-di-tipi} > & ::= & < \textit{nome tipovalore} > \ | \\
 & & < \textit{nome tipovalore} > \ \& \ < \text{nomi-di-tipi} > \\
< \text{tipo-cammino} > & ::= & (< \textit{nome attributo} > : < \text{desc-att} >) \\
< \text{insiemi-di-classi} > & ::= & \{ < \text{classe} > \} \ | \\
 & & \{ < \text{classe} > \} \ \& \ < \text{insiemi-di-classi} > \\
< \text{esiste-insiemi-di-classi} > & ::= & !\{ < \text{classe} > \} \ | \\
 & & !\{ < \text{classe} > \} \ \& \ < \text{esiste-insiemi-di-classe} > \\
< \text{sequenze-di-classi} > & ::= & \langle < \text{classe} > \rangle \\
 & & \langle < \text{classe} > \rangle \ \& \ < \text{sequenze-di-classi} > \\
< \text{nomi-di-classi} > & ::= & < \textit{nome classe} > \ | \\
 & & < \textit{nome classe} > \ \& \ < \text{nomi-di-classi} > \\
< \text{def-tipobase} > & ::= & \mathbf{btype} < \textit{nome tipobase} > = \ < \text{tipobase} > \\
< \text{tipobase} > & ::= & \mathbf{real} \ | \\
 & & \mathbf{integer} \ | \\
 & & \mathbf{string} \ | \\
 & & \mathbf{boolean} \ | \\
 & & < \text{range-intero} > \ | \\
 & & \mathbf{vreal} < \textit{valore reale} > \ | \\
 & & \mathbf{vinteger} < \textit{valore intero} > \ | \\
 & & \mathbf{vstring} < \textit{valore string} > \ | \\
 & & \mathbf{vboolean} < \text{valore-boolean} > \ | \\
 & & < \textit{nome tipobase} > \\
< \text{valore boolean} > & ::= & \mathbf{true} \ | \ \mathbf{false} \\
< \text{range-intero} > & ::= & \mathbf{range} < \textit{valore intero} > \ +\mathbf{inf} \ | \\
 & & -\mathbf{inf} \ < \textit{valore intero} > \ | \\
 & & < \textit{valore intero} > \ < \textit{valore intero} >
\end{array}
$$