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

Persistenza di Schemi di Basi di Dati ad Oggetti

Tesi di Laurea di
Duccio Fontana

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Correlatore
Chiar.mo Prof. Claudio Sartori

Anno Accademico 1994 - 95

Parole chiave:
Basi di dati ad oggetti
Persistenza
ODMG-93
Schemi di basi di dati
Consistenza

Ai miei genitori

RINGRAZIAMENTI

Ringrazio la Prof.ssa Sonia Bergamaschi, il Prof. Claudio Sartori, il Dott. Ing. Domenico Beneventano, l'Ing. Maurizio Vincini la Dott.ssa Alessandra Garuti e il P.H.D. Ing. Stefano Lodi per il prezioso aiuto fornito nella realizzazione della presente tesi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Sommario

La presente tesi si colloca nell'area delle basi di dati orientate agli oggetti (*OODB - Object Oriented DataBase*) e tratta il problema della descrizione e della persistenza di schemi di basi di dati.

L'obiettivo fondamentale che ci si è posti è stato quello di fornire uno strumento generale di ausilio al progettista di una base di dati ad oggetti nella fase di progettazione concettuale. A tale scopo sono stati progettati un linguaggio per la descrizione di schemi e un componente software, denominato *OCDL-PSSManager*, in grado di interpretare tale descrizione (conforme allo standard *ODMG-93*), di gestire schemi persistenti (tramite il sistema di memorizzazione di oggetti persistenti *EOS*) e di verificare la coerenza degli schemi stessi utilizzando il componente *OCDL-Designer*, sviluppato presso il Dipartimento di Scienze dell'Ingegneria di questa stessa università.

La scelta di adottare lo standard ODMG-93 è motivata dal requisito di generalità del componente OCDL-PSSManager che si vuole rendere disponibile ed utilizzabile per i diversi ODBMS (*Object DataBase Management System*) in commercio. I limiti del componente sviluppato riguardano l'espressività attuale del modello OCDL che rende possibile il controllo di coerenza di schemi di basi di dati ad oggetti solo relativamente agli aspetti strutturali. Allo stato attuale, non è quindi possibile descrivere e fornire controlli di coerenza sui metodi, i quali costituiscono un aspetto molto importante degli OODB.

D'altro lato va osservato che attraverso OCDL è possibile descrivere direttamente in uno schema, tramite regole di integrità, un insieme rilevante di

vincoli di integrità non esprimibili negli attuali ODBMS commerciali e non previsti in ODMG-93.

1.1 I modelli di dati orientati agli oggetti

Le Basi di Dati Orientate agli Oggetti, OODB (*Object Oriented DataBase*), sono attualmente oggetto di intensi sforzi di ricerca e di sviluppo. La motivazione principale per questo interesse è che il paradigma orientato agli oggetti offre un grande insieme di strutture dati e di facilità di manipolazione che lo rendono adatto per il supporto delle applicazioni che trattano grandi quantità di dati. Tra le più recenti spiccano il CAD (Computer Aided Design), il CASE (Computer Aided Software Engineering) e il CIM (Computer Integrated Manufacturing); a queste si aggiungono le applicazioni per la gestione e la presentazione, spesso costituita dall'integrazione di immagine, testo e suono, di informazioni riguardanti i più disparati campi (ambientale, sanitario, gestionale, culturale, etc.), dette applicazioni *multimediali*.

Un modello di dati orientato agli oggetti, OODM (*Object Oriented Data Model*), proposto per OODB in [A+89] e [KFL89] si basa sulle nozioni di *oggetti*, *classi*, *attributi* e *metodi*, dove le classi e gli attributi sono usati per descrivere gli aspetti *strutturali* (la conoscenza sulla struttura degli oggetti in un dominio di applicazione) mentre i metodi sono usati per rappresentare gli aspetti *comportamentali* (le funzioni degli oggetti) della realtà di interesse.

In questa tesi viene trattato il problema della definizione e della persistenza di schemi di basi di dati ad oggetti. In particolare, è stato progettato e sviluppato un interprete di comandi che permette la definizione (secondo lo standard *ODMG-93*), la memorizzazione persistente (tramite il gestore di memorizzazione di oggetti *EOS*) e la verifica di coerenza (utilizzando il componente *OCDL-Designer*) di tali schemi.

Nel seguito si riportano brevemente le principali caratteristiche di un OODBMS (*Object-Oriented Database Management System*) o anche ODBMS

- **Oggetti e identità**: Ogni entità del mondo reale è modellata come un oggetto; ogni oggetto ha associati un insieme di attributi e un identificatore unico.

- **Incapsulazione**: in ogni oggetto sono definiti e racchiusi sia le procedure (*metodi*) che l'interfaccia con la quale può essere acceduto e manipolato da altri oggetti. L'interfaccia di un oggetto è costituita

dall'insieme delle operazioni che possono essere invocate sull'oggetto. Lo stato di un oggetto (valore dei suoi attributi) è manipolato tramite l'esecuzione di metodi invocati dalle operazioni corrispondenti. Gli oggetti con struttura e comportamento comuni sono raggruppati in una *classe*.

- **Ereditarietà**: una classe può essere definita come specializzazione di una o più classi esistenti ed eredita attributi e metodi da esse.
- **Estendibilità**: un ODBMS è estendibile, cioè permette di aggiungere nuove classi e nuovi oggetti e trattarli come classi e oggetti del sistema.

Nel lavoro presentato in questa tesi si è interessati agli aspetti strutturali del modello, cioè alle modalità secondo le quali il modello permette di organizzare i dati. Pertanto non si prenderanno in considerazione gli aspetti dinamici di ODBMS, quali quelli legati all'incapsulamento, ai metodi e alla modularizzazione. Delle caratteristiche sopra citate ci interessano, quindi, esclusivamente l'oggetto e la sua identità, l'ereditarietà e l'estendibilità che vengono meglio descritte nel seguito.

Oggetti e identità

Ogni elemento componente la realtà da modellare viene rappresentato tramite un unico concetto di base: l'*oggetto*. Ogni oggetto è univocamente individuato da un identificatore di oggetto, *oid* (object identifier), ed ha associato uno stato che è costituito dal valore delle sue *proprietà* o *attributi*. Nei classici linguaggi orientati agli oggetti, come ad esempio Smalltalk [GR83], il valore associato ad un oggetto è atomico oppure è una ennupla di altri oggetti. Questa caratteristica è stata in parte ereditata da alcuni ODBMS, dove il valore di un oggetto è una tupla oppure un insieme di altri oggetti, cioè è sempre un valore piatto che può solo contenere identificatori di altri oggetti e non direttamente altri valori complessi.

Per superare questa limitazione sono stati sviluppati diversi modelli ad oggetti con valori complessi [AK89, Atz93, LR89, Bee90], indicati con CODMs (*Complex Object Data Models*). In questi modelli si trattano in modo uniforme sia oggetti con identità sia valori complessi senza identità. Un *valore complesso* o *valore strutturato* è un valore definito a partire sia da valori atomici che da identificatori di oggetti mediante l'uso ricorsivo di costruttori,

quali ad esempio il costruttore di *tupla*, di *insieme*, e di *sequenza*. Uno *schema* contiene le informazioni sulla struttura dei dati. Nei lavori citati sono presenti entrambe le nozioni di *classe* e di *tipo*. I tipi denotano una struttura e una *estensione*, intesa come insieme di valori. Anche le classi denotano un'estensione, intesa come insieme di oggetti. Tuttavia, mentre l'estensione denotata da un tipo è definita dalla sua struttura (cioè tutti gli elementi di un dominio la cui struttura coincide con quella di un dato tipo sono istanze di quel tipo), l'estensione associata ad una classe è definita dall'utente (cioè gli elementi del dominio devono essere esplicitamente inseriti fra le istanze di una classe). Ad ogni classe è normalmente associato un tipo il quale descrive la struttura degli oggetti che possono essere *istanziati* nella classe. Quindi le istanze della classe sono oggetti il cui valore associato è, a sua volta, istanza del tipo che descrive la classe.

Ereditarietà

L'*ereditarietà*, stabilita tramite la relazione *isa*, è un'importante caratteristica degli OODB. Essa costituisce un potente mezzo di modellazione, essendo in grado di dare una precisa e concisa descrizione del dominio di applicazione [A+89]. Dal punto di vista intensionale, cioè le strutture di concetti e le relazioni tra concetti, la dichiarazione di ereditarietà tra due classi A e B, cioè A *isa* B, consente la definizione della *sottoclasse* A come specializzazione della *superclasse* B. Una sottoclasse eredita le proprietà della superclasse, può avere proprietà aggiuntive e può ridefinire alcune proprietà della superclasse. Dal punto di vista estensionale, la dichiarazione A *isa* B, stabilisce che ogni oggetto di A sia anche un oggetto di B. Si parla di *ereditarietà multipla* nel caso in cui sia ammesso che una classe possa avere più di una superclasse.

Il modello considerato nella presente tesi è un modello per basi di dati orientate agli oggetti con ereditarietà multipla che permette la modellazione di valori complessi tramite la nozione di tipo e di classe.

1.2 Progetto assistito di schemi di basi di dati

Modellare un dominio di applicazione in termini di modelli concettuali è una fase importante di ogni metodologia per il progetto di una base di dati (o

database). Tale fase, detta *progetto concettuale*, fornisce come risultato uno *schema di dati contettuale* che descrive la *conoscenza intensionale*.

Una progettazione top-down di un database comincia con l'*analisi dei requisiti*. Guardando la documentazione esistente sul dominio da modellare, interpellando gli utenti finali, analizzando, per esempio, l'organizzazione di una società e il flusso delle informazioni, è possibile capire le richieste dell'utente e dividerle in contesti più o meno indipendenti. L'astrazione di queste porzioni limitate del sistema informativo porta alla rappresentazione formale degli oggetti, alle loro proprietà, alle loro correlazioni e alle operazioni che li coinvolgono attraverso un modello concettuale (il *modello E/R* tradizionalmente e, più recentemente, modelli basati su oggetti (*OODM*)). L'opportuna selezione degli aspetti da modellare dipende dalle operazioni che l'utente pensa di effettuare o, più in generale, dagli scopi del sistema per la gestione della base di dati. Il risultato della prima fase del progetto di un database è quindi un insieme di viste esterne delle caratteristiche statiche e degli aspetti dinamici principali.

Il passo successivo è l'integrazione delle varie viste esterne. La loro semplice unione non è sufficiente in quanto concetti simili possono essere descritti in più viste. È necessario capire precisamente il significato di ogni oggetto e/o attributo e raggruppare e unificare gli oggetti e gli attributi che hanno significati simili; la conoscenza, poi, di quali sono le più importanti operazioni che si ha intenzione di mettere in atto, facilita la comprensione e il confronto fra oggetti.

Si introduce un esempio per approfondire questa tematica.

1.3 Esempio: il dominio Magazzino

Consideriamo il seguente esempio riguardante la struttura di un società che si occupa della gestione di un magazzino; la realtà di interesse viene modellata ottenendo uno schema di base di dati illustrato graficamente in figura 1.1 e in una sintassi OODB-like in tabella 1.1, dove le classi sono rappresentate da ellissi e le relazioni di specializzazione da frecce in grassetto.

I materiali (`material`) sono descritti da un nome (`name`, costituito da una stringa di caratteri), un rischio (`risk`, costituito da un numero intero) e da una caratteristica (`feature`, costituita da un insieme di stringhe). I magazzini (`storage`) sono identificati da una categoria (`category`, costituita da una stringa), sono guidati (`managed-by`) da un dirigente (`manager`) e contengono (`stock`) un insieme di articoli (`item`) che sono dei materiali (`material`) per ciascuno dei quali è indicata la quantità presente (`qty`, numero intero). I dirigenti (`manager`) hanno un nome (`name`) dato da una stringa, un salario (`salary`) superiore ai 40K dollari e un livello (`level`) compreso tra 1 e 15. I massimi dirigenti (`tmanager`) sono quei dirigenti (`manager`) che hanno un livello compreso tra 8 e 12. Infine abbiamo dei magazzini marcati come "speciali" (`sstorage`) che sono un sottinsieme di magazzini e materiali "speciali" (`smaterial`) che sono un sottinsieme di materiali. Le classi descritte sono tutte di tipo base (classi *primitive*), avendo fissato solo condizioni necessarie per l'appartenenza di oggetti ad una classe.

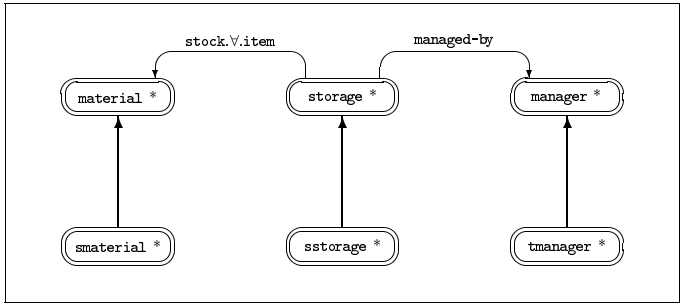

Figura 1.1: Il dominio Magazzino

Dallo studio iniziale della realtà di cui si vuole ottenere un modello, si sono ricavate alcune regole (vincoli di integrità) che vanno applicate alle classi; esse rappresentano condizioni necessarie e sufficienti per la legalità delle istanze dello schema. Questi vincoli possono essere descritti in linguaggio naturale nel seguente modo:

" **per tutti** i dirigenti (`manager`),
se il livello (`level`) è compreso tra 5 e 10,
allora il salario (`salary`) deve essere compreso tra 40K e 60K. "

" **per tutti** i materiali (`material`),
se il rischio (`risk`) è maggiore di 10,
allora devono essere dei materiali speciali (`smaterial`). "

" **per tutti** i magazzini (`storage`),
se la categoria (`category`) è "B4",
allora devono essere guidati da un massimo dirigente (`tmanager`). "

" **per tutti** i magazzini (`storage`),
se ciascun articolo (`item`) immagazzinato (`stock`) è di un materiale speciale (`smaterial`),
allora devono essere dei magazzini speciali (`sstorage`). "

" **per tutti** i magazzini (`storage`),
se ciascuna quantità (`qty`) immagazzinata (`stock`) è compresa tra 10 e 50,
allora la categoria (`category`) deve essere "A2". "

Usando un linguaggio di tipo OODB-like, lo schema del Magazzino comprensivo dei vincoli di integrità è riportato nella tabella 1.1.

1.4 Contenuto della tesi

Il componente software sviluppato, denominato OCDL-PSSManager, consente di memorizzare persistentemente le informazioni relative a uno schema di base di dati ad oggetti in modo che sia possibile:

- recuperarle in modo efficiente;
- eseguire il controllo di coerenza dello schema generato;
- apportare eventuali modifiche allo schema.

È da notare che non esiste nessun vincolo di ordinamento nell'immissione di queste informazioni. Il sistema prodotto è in grado di gestire situazioni in cui, ad esempio, nello schema della base di dati è presente la definizione di una regola senza che sia definita la classe o, più in generale, il tipo a cui

class	material	=	[name: string, risk: integer, feature : {string}]
class	smaterial	=	**isa** material
class	storage	=	[managed-by : manager, category : string, stock : {item : material, qty : $10 \div 300$]}]
class	sstorage	=	**isa** storage
class	manager	=	[name: string, salary: $40K \div \infty$, level : $1 \div 15$]
class	tmanager	=	**isa** manager **and** [level : $8 \div 12$]

Vincoli di integrità:

if manager **and** (level : $5 \div 10$)		**then**	(salary : $40K \div 60K$)
if material **and** (risk > 10)		**then**	smaterial
if storage **and** (category $= "B4"$)		**then**	(managed-by : tmanager)
if storage **and** (stock.$\forall$.item $=$	smaterial)	**then**	sstorage
if storage **and** (stock.$\forall$.qty : $10 \div 50$)		**then**	(category $= "B4"$)

Tabella 1.1: Lo schema del Magazzino in una sintassi OODB–like

si riferisce. Vedremo più avanti in dettaglio quali problemi si verificano in questa e in altre situazioni.

Per ottenere la persistenza dello schema di una base di dati si è utilizzato il gestore di memorizzazione *EOS*, presentato nel **capitolo 2**. È stato sviluppato presso gli AT&T Bell Labs ed è disponibile alle comunità accademica e scientifica per uso non commerciale (in gergo, si dice che è *free software*).

Il **capitolo 3** descrive brevemente il modello ad oggetti *OCDL* e lo standard *ODMG-93*. Per quanto riguarda OCDL (Object and Constraint Definition Language), sul quale si basa il presente lavoro, particolare attenzione è stata riservata alle definizioni di oggetti complessi, di schemi di base di dati, dell'operatore di intersezione (usato per descrivere l'ereditarietà), di istanze (possibili e legali) di uno schema e di vincoli (o regole) di integrità. Questi primi contenuti riprendono i lavori [BN91, BB93, Ben94, **?**] nei quali è possibile trovare spiegazioni ancora più approfondite. Nella stesura del liguaggio di descrizione di schemi di basi di dati, ci si è basati sullo standard ODMG-93 per basi di dati ad oggetti, presentato sempre in questo capitolo. Si tratta di una proposta nata per supportare l'affermazione dell'approccio

ad oggetti nella produzione di ODBMS. Tuttavia, il rispetto di tale standard è stato fortemente limitato dal fatto che si basa su un modello di dati (OM) sostanzialmente diverso da OCDL. Comunque, si è potuto utilizzare ODMG-93, non senza apportare osservazioni e modifiche, per la parte riguardante il linguaggio di definizione strutturale degli oggetti. La grammatica del linguaggio OCDL trasformato in base allo standard è riportata in appendice B.

Nel **capitolo 4** si presenta il componente *OCDL-PSSManager*, frutto del lavoro svolto. Un'interfaccia testuale permette all'utente di interagire con il sistema e, grazie alle potenzialità fornite da EOS, OCDL-PSSManager gestisce schemi persistenti di basi di dati. Preliminarmente si introducono sia gli strumenti di sviluppo *Lex* e *Yacc* che il modulo software *OCDL-Designer* (presentato in [**?**]) utilizzato per eseguire il controllo di coerenza. Si descrivono poi in dettaglio l'architettura e i principi adottati dal componente software prodotto.

Il manuale utente è riportato nel **capitolo 5**. Assieme alla descrizione e alla sintassi dei comandi invocabili, si riportano numerosi esempi di interazione con OCDL-PSSManager, evidenziando quali possono essere situazioni anomale che limitano il suo comportamento e suggerendo possibili soluzioni da adottare.

La grammatica di OCDL è riportata in duplice forma: in **appendice A** è descritta la versione originale; in **appendice B** si riporta la versione modificata in base a ODMG-93.

L'**appendice C** contiene tutto il codice sorgente (oltre 7000 linee) in linguaggio ANSI C++ che costituisce OCDL-PSSManager, mentre l'**appendice D** riporta il codice (quasi 2000 linee) che implementa le funzioni per l'interfacciamento, non ancora realizzato, tra ODB-QOptimizer (vedi [**?**, **?**]) e lo schema reso persistente con OCDL-PSSManager.

Capitolo 2

Il sistema EOS

EOS è un *object storage manager* sviluppato in linguaggio C++ presso gli AT&T Bell Labs negli Stati Uniti, da un gruppo guidato dai progettisti A. Biliris ed E. Panagos e reso disponibile alla comunità accademica e a quella scientifica per uso non commerciale. Per supportare il presente lavoro si è utilizzata la versione 2.2.

Poichè EOS è uno degli strumenti principali del lavoro svolto, ne vengono evidenziate le caratteristiche e l'architettura e viene descritta l'interfaccia che il sistema mette a disposizione al programmatore per poter sviluppare DBMS orientati agli oggetti caratterizzati da elevate prestazioni. EOS non è infatti propriamente un OODBMS, ma si colloca ad un gradino inferiore, fornendo al programmatore i meccanismi e le primitive, richiamabili attraverso programmi scritti in C o C++, per ottenere la persistenza degli oggetti, e creare così dei veri e propri OODBMS.

Di seguito viene fornito un breve elenco delle caratteristiche principali di tale sistema:

- accesso trasparente ad oggetti di piccole e grandi dimensioni;
- accesso agli oggetti sia tramite chiamate di funzione che svolgono operazioni a livello di byte quali read, write, append, etc., sia direttamente all'interno della memoria "cache" del programma client, senza incorrere in ulteriori costi di memoria dovuti alla necessità di copia delle informazioni dalla cache alla memoria normale;
- possibilità di creare un numero illimitato di oggetti aventi un nome: EOS è in grado di garantire integrità referenziale fra gli oggetti con

nome ed il nome stesso; ciò vuol dire che quando un oggetto con nome è rimosso dal database anche il suo nome viene rimosso;

- funzione di Hashing estensibile, in grado di supportare chiavi di lunghezza arbitraria e funzioni di hashing definite dall'utente;
- file di configurazione modificabili per personalizzare e regolare le prestazioni del sistema;
- disponibilità di file per raggruppare oggetti aventi una qualunque relazione;
- possibilità di estensione delle funzionalità stesse di EOS, al fine di migliorarle;
- EOS offre un ambiente client-server con gestione delle transazioni per le applicazioni accedono ad aree di memorizzazione condivise; in aggiunta, ogni applicazione può utilizzare aree private (o locali) senza ricorrere all'intermediazione del server;
- controllo di concorrenza basato su protocollo di locking multigranulare a due fasi (MG-2V-2PL multigranularity two version two phase lock protocol) [**?**], in grado di permettere l'accesso contemporaneo a più utenti in lettura e ad uno in scrittura;
- file di log di dimensioni contenute e rapidità nel ripristino dopo crash di sistema, poiché è necessaria una sola lettura sequenziale di tali log;
- checkpoint non bloccanti in grado di permettere alle transazioni attive di continuare ad accedere al database anche durante un checkpoint;
- EOS è scritto interamente in C++, ma può essere utilizzato da programmi scritti sia in C++ che in C, è compatibile con la maggior parte di compilatori C/C++, quali quelli di AT&T, Sun, GNU e CenterLine. In aggiunta ne esistono versioni per le principali architetture, Sun SPARC sia con sistema operativo SunOS 4.1.x che SOLARIS 2.x, SGI MIPS con sistema operativo IRIX 4.x e 5.x ed infine IBM RS6000 con sistema operativo AIX.

2.1 Concetti fondamentali del sistema EOS

2.1.1 Basi di dati e aree di memorizzazione

Nell'architettura di EOS un *database* è un insieme di file e di oggetti che viene creato in una delle aree di memorizzazione disponibili: tali aree possono essere semplici file Unix oppure intere partizioni di disco, e vengono create invocando dal prompt Unix il comando EOS "*eosareaformat*" specificando come argomento il nome da assegnare all'area stessa (il file di configurazione *~/.eos/formatrc* contiene i parametri, modificabili dall'utente, in base ai quali viene formattata l'area). È interessante notare come un database possa estendersi a più aree, infatti gli oggetti che appartengono a un qualsiasi database possono essere memorizzate sia nell'area in cui il database è stato creato, sia in una qualunque altra area disponibile; inoltre possono esistere aree che non contengono nessun database, ma solo oggetti appartenenti a diversi database.

Le aree di memorizzazione (o *storage areas*) possono essere sia *condivise* che *private* (o *locali*). Nella prima ipotesi si accede ad esse in un contesto *Client-Server* attraverso il server EOS, in grado di offrire il controllo di concorrenza e il recovery a seguito di crash. Nella seconda ipotesi, le aree sono private e vi accede solo l'utente che le ha create; qui non viene fornito nessun controllo di concorrenza e nessuna capacità di recupero a seguito di cadute di sistema, ma in compenso le operazioni di accesso a memoria di massa sono sensibilmente più veloci non essendo necessario effettuare chiamate al server EOS.

2.1.2 Identificatore di oggetto (OID)

Gli oggetti resi persistenti attraverso i meccanismi forniti da EOS sono riconosciuti da identificatori univoci, denominati OID (Object IDentifier); si tratta di strutture di dimensione pari ad 8 byte che contengono il numero della storage area, il numero di pagina all'interno dell'area, l'offset dell'oggetto all'interno della pagina (detto *slot number*) ed un numero per mantenere l'unicità degli oid quando lo spazio nell'area viene riutilizzato. L'identificatore di oggetto rappresenta, quindi, un *indirizzo logico* unico fra tutti quelli degli oggetti contenuti in tutte le aree di memorizzazione gestite dal server di EOS.

2.1.3 File, file scan e cluster

I *file* in EOS sono un mezzo per ragruppare oggetti aventi una qualche correlazione e forniscono la possibilità di navigare attraverso gli oggetti che contengono. Possono contenere altri file, ma ogni oggetto appartiene esattamente ad un file.

In tal modo si viene a formare un albero in cui i nodi sono i files, e le foglie sono gli oggetti ordinari: infatti ogni volta che viene creato un nuovo database, EOS automaticamente crea un file che serve come radice di tale albero. A livello fisico i file sono formati da un numero di pagine e/o di segmenti che non possono essere condivise/i tra file diversi.

È possibile scandire i file sia in avanti che retrocedendo utilizzando i *file scan*; in aggiunta si possono restringere i limiti della scansione agli oggetti appartenenti ad una determinata pagina del file.

Per migliorare le prestazioni dei programmi client è consentito esercitare un controllo sulla posizione fisica in cui gli oggetti vengono memorizzati all'interno del database: si può infatti forzare l'object manager a posizionare un nuovo oggetto vicino ad uno esistente e il sistema EOS garantisce che il nuovo oggetto verrà memorizzato nella stessa pagina del primo, se lo spazio disponibile è sufficiente, altrimenti, in una nuova pagina vicina a quella in cui l'oggetto già esistente risiede. I programmi client possono inoltre dare istruzioni al sistema EOS di collocare il nuovo oggetto:

- in una specifica area;
- alla fine di un file;
- in una nuova pagina;
- riservando una pagina esclusivamente ad esso, facendo in modo che nessun altro oggetto in futuro sia memorizzato su tale pagina.

2.1.4 Rappresentazione degli oggetti ed object handle

Ogni oggetto EOS dispone di un'intestazione (*header*) che ne contiene le principali informazioni, quali la lunghezza dell'oggetto, se esso è o meno un oggetto di grandi dimensioni, se ha nome e così via; due dei bytes nell'header non hanno alcun significato per il gestore di oggetti e sono così a disposizione del programmatore.

Per poter operare su un oggetto bisogna richiedere al gestore un "*handle*" a tale oggetto. Un *handle* è un puntatore ad una struttura che contiene, fra le varie informazioni, l'indirizzo di memoria riservato all'oggetto all'interno del *buffer* di EOS. La ricaduta in termini pratici della richiesta di un handle ad un oggetto è la creazione di un *lock* per la pagina che lo contiene, ed il trasferimento della pagina stessa nel buffer; la pagina non verrà mossa dal buffer sino a che l'handle non verrà rilasciato. Una volta che l'applicazione non necessita più dell'oggetto, è compito del programma rilasciare l'handle per fare in modo che anche la relativa pagina sia liberata dal lock.

In termini di costo la richiesta di un handle ad un oggetto è quello di un singolo accesso al disco, ovvero il costo necessario per il trasferimento in memoria di una intera pagina; successivamente, ottenuto l'handle, il costo di accesso all'oggetto è il medesimo di ogni altro accesso ad un puntatore in memoria.

2.1.5 Oggetti di grandi dimensioni

EOS è stato progettato per poter trattare oggetti di dimensione arbitraria, limitata unicamente dalla disponibilità di unità di spazio libero su memoria di massa. Formalmente un oggetto è *piccolo* se è possibile contenerlo all'interno di una sola pagina, altrimenti è *grande*.

L'accesso agli oggetti grandi è trasparente alle applicazioni perchè è il medesimo che nel caso di oggetti piccoli. Per oggetti di notevoli dimensioni, come può essere in caso di database multimediali dove la dimensione degli oggetti ha ordine di grandezza dei *gigabyte*, è possibile accedere a porzioni arbitrarie dell'oggetto. EOS fornisce infatti le primitive necessarie per leggere, scrivere, inserire e cancellare una qualsivoglia porzione di byte all'interno dell'oggetto.

2.1.6 Indici

EOS fornisce indici di hashing estensibili [**?**], le chiavi degli indici possono essere stringhe di lunghezza variabile o qualunque altra struttura di lunghezza fissata ed altrettanto possono essere i valori associati alle chiavi.

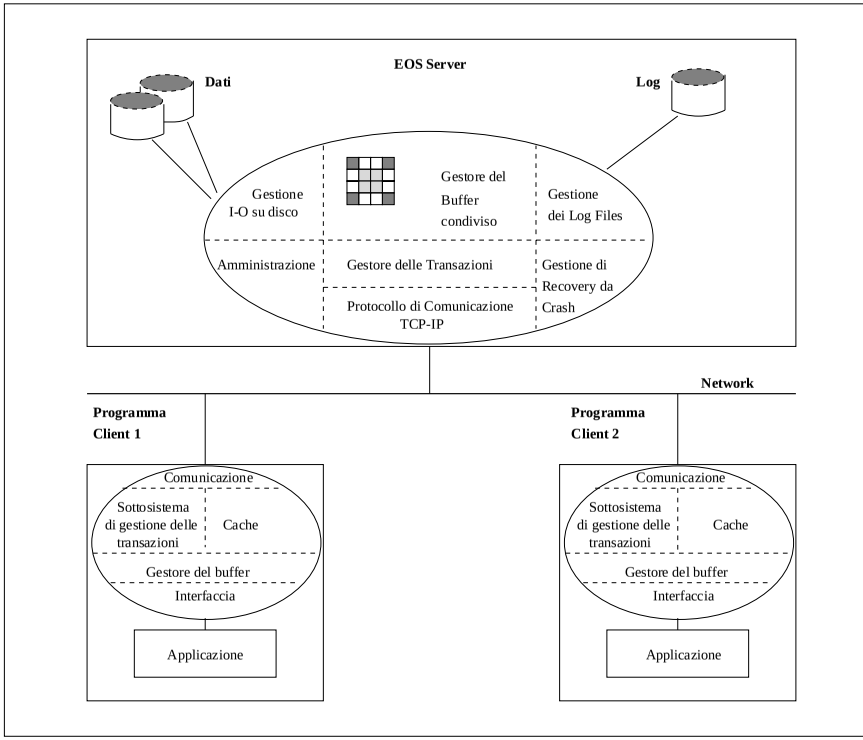

Figura 2.1: Architettura Client-Server di EOS

2.2 EOS: architettura client-server

Il server EOS è un processo demone[1] multi-threaded, dove con multi-threaded si intende un ambiente che permette la gestione concorrente di diverse esecuzioni, denominate *thread*. La differenza con il multitasking deriva dal fatto che i thread condividono gran parte del loro ambiente di esecuzione; tipicamente la condivisione dell'enviroment è ben più spinta di quanto avvenga nel caso di diversi task in un ambiente multitasking. Essi possono per questo essere attivati e disattivati in maniera estremamente rapida, dal momento che sono poche le variabili proprie di ogni thread che devono essere salvate e poi ripristinate nel passaggio da un thread all'altro [**?**]. Il server funge da tramite fra il database ed i processi che tentano di accedervi. Per evitare che le chiamate Unix di Input-Output sui dischi si rivelino bloccanti, il server EOS crea un processo separato per dialogare con le richieste di I/O per un'area di memorizzazione (*disk process)*, non appena si accede per la prima volta alla memoria di massa, rendendo così possibili operazioni di I/O in parallelo. Il server stesso gioca anche il ruolo di *area manager* permettendo la creazione e la rimozione dinamica delle aree di memorizzazione, potendone gestire sino a 6000 diverse.

Quando il server EOS viene invocato, crea i processi di checkpoint e di log ed un certo numero di aree di memoria (centrale) condivisa e di semafori, utilizzando le capacità di memoria condivisa, mappaggio di memoria e semafori dello Unix System V [2]. Il server può essere attivo come processo sia sulla stessa macchina del programma client, che su una macchina diversa, utilizzando connessioni TCP/IP attraverso socket[3] di UNIX. Lo spazio di memoria allocato per il buffer, utilizzato per portare in memoria volatile gli oggetti permanenti, si trova all'interno dell'area di memoria condivisa; i semafori si rendono necessari per controllare l'accesso a tale area dei vari thread e dei disk process, mentre code di messaggi e socket di UNIX forniscono i meccanismi di comunicazione interprocesso fra i thread e le richieste di I/O dirette ai disk process.

[1] un processo demone è un processo che ha scollegato l'I/O da console, quindi può comunicare solo con altri processi

[2] Tutti questi concetti relativi alla programmazione in ambiente Unix, sono trattati esaurientemente in [**?**] [**?**].

[3] un socket è un oggetto utilizzato per la comunicazione interprocesso; non è da confondere con la *pipe*

2.2.1 Controllo di concorrenza

Il controllo di concorrenza viene fornito a livello di pagina, attraverso un protocollo di locking multigranulare a due fasi (*MG-2V-2PL*) [**?**], che impone alle transazioni di acquisire un lock ai singoli elementi, prima di accedervi effettivamente e di rilasciarlo unicamente quando esse hanno terminato con successo (*committed*) o sono state interrotte prima del termine (*aborted*). In questo modo varie transazioni in lettura ed una in scrittura possono operare simultaneamente sullo stesso elemento; la transazione in scrittura deve attendere che tutte le transazioni in lettura terminino prima di terminare anch'essa.

2.2.2 Log

Il sistema mantiene due tipi di log: un log *globale* ed un certo numero di log *privati*; ogni log privato è associato ad una sola transazione. I record che contiene riportano il risultato delle modifiche generate dalla corrispondente transazione (sono denominati *redo record*). Il log globale, a sua volta, è formato da record che vengono generati sia dopo che una transazione ha terminato con successo (in tal caso il singolo record, *commit record*, contiene l'identificatore della transazione stessa e delle informazioni relative alla modifica apportata), sia successivamente ad un checkpoint (nel qual caso contiene gli identificatori di tutte le transazioni terminate con successo, insieme all'indirizzo, all'interno del log globale, del relativo commit record).

2.2.3 Capacità di recupero

Il recupero a seguito di malfunzionamento (o crash) di sistema è basato sul protocollo *NO-UNDO/REDO*, il cui principio di funzionamento è descritto in seguito.

Le modifiche causate da una transazione vengono memorizzate nella porzione di memoria che funge da buffer privato di tale transazione (quello che nel precedente paragrafo è stato indicato con "log privato"); solo nel momento in cui la transazione termina correttamente, le modifiche ai dati sono rese persistenti, lo stesso log privato viene memorizzato stabilmente su disco, viene inserito un commit record nel file di log globale e a sua volta portato in memoria secondaria. Se la transazione termina in una condizione di *abort*

il suo log privato viene semplicemente cancellato dalla memoria volatile e i lock relativi alla transazione vengono rilasciati.

Il procedimento è leggermente differente nel caso di oggetti di grandi dimensioni: per evitare infatti l'ingente domanda di risorse che sarebbe indotta dalla copia di un oggetto di grandi dimensioni nel buffer del server EOS, le modifiche a tali oggetti non sono bufferizzate, ma vengono direttamente memorizzate nel database, senza però sovrascrivere l'oggetto stesso; se la transazione non va a buon fine (terminazione con *abort*, le modifiche vengono cancellate e inoltre nessuna altra transazione è stata posta nella condizione di vedere i cambiamenti temporanei per via del lock mantenuto sulla pagina che li conteneva.

2.2.4 Checkpoint

Nel momento in cui il server EOS viene invocato, un suo specifico componente, detto *recovery manager*, viene attivato; il suo compito è leggere i file di log e riportare il database ad una situazione anteriore ad un eventuale crash di sistema. Per ridurre il lavoro ed il tempo necessario al recovery manager nel porre rimedio ad una situazione di malfunzionamento di sistema, vengono attivati *checkpoint* periodici durante il normale funzionamento del server. In questo modo tutte le pagine che hanno subito modifiche e che risiedono ancora nell'area di memoria condivisa, sono rese persistenti ed un record di checkpoint, che mantiene traccia della conclusione di tale processo, viene memorizzato su disco; in caso di ripristino dovuto a caduta di sistema, il recovery manager si trova a dover rifare le modifiche apportate dalle transazioni completate (correttamente) dopo il checkpoint. Al termine dell'operazione di ripristino viene effettuato un checkpoint, dopodichè il sistema è di nuovo attivo.

Queste operazioni di checkpoint non sono bloccanti, cioè durante il loro compimento nuove transazioni possono avere inizio e quelle attive possono comunque continuare ad accedere alle risorse del server.

2.2.5 Ripristino a seguito di cadute di sistema

In caso di crash il server EOS riporta il database nell'ultimo stato consistente precedente alla verifica dell'anomalia: ciò è reso possibile mediante una lettura sequenziale del file di log globale e rifacendo tutte le modifiche effettuate sia dalle transazioni andate a buon fine prima del crash che dalle

transazioni interrotte dal crash, nello stesso ordine in cui erano state originariamente eseguite. Tale procedura di *restart* si rivela sufficientemente rapida per due ragioni:

- è necessaria una sola lettura del file di log;
- il log file ha comunque dimensione contenuta, in quanto contiene solo la nuova immagine prodotta da una transazione; in più viene archiviato ogni volta che viene attivato un checkpoint oppure quando raggiunge dimensioni superiori ad una determinata soglia.

2.3 Interfaccia C++ al sistema EOS

Si vuole ora analizzare l'interfaccia C++ che il sistema EOS mette a disposizione del programmatore per lo sviluppo di sistemi di gestione di database orientati agli oggetti. Sebbene non sia qui possibile descrivere ogni singola funzione presente nelle due librerie di EOS, verranno presentate le classi che il sistema mette a disposizione, insieme ai loro principali metodi di accesso.

Un primo elenco, presentato secondo una successione che rispetta l'ordine con cui idealmente tali classi vanno pensate ed utilizzate, è il seguente:

- classe eosdatabase;
- classe eostrans;
- class eosfile;
- classe eosoid;
- classe eosobj;
- classe eos_new;
- classe eos_Ref;

2.3.1 Basi di dati: classe eosdatabase

I metodi appartenenti a questa classe permettono la creazione, la rimozione, l'apertura e la chiusura di un database.

Il seguente metodo permette l'apertura di un database esistente:

static eosdatabase open(const char *name,int rdonly=0,*
int create=0,int trunc=0)

in esso il parametro *name* identifica il database da aprire e deve essere nella forma *[nome_host:]nome_dell_area/nome_del_db*, gli altri parametri, che se non esplicitamente modificati assumono il valore di default falso [4], possono essere utilizzati secondo le seguenti modalità:

rdonly: se posto a vero, il database viene aperto solo in lettura;
create: se posto a vero, il database viene creato se non esiste;
trunc: se posto a vero, il contenuto del database viene cancellato.

La chiusura del database avviene attraverso il metodo

int eosdatabase::close(void)

Due metodi di particolare interesse sono i seguenti:

*int set_object_name(const eos_Ref_Any& objRef, char *name)*

*eos_Ref_Any lookup_object(const char *name)*

il primo assegna il nome indicato dalla variabile *name* all'oggetto puntato dal riferimento persistente *objRef*; il secondo metodo ricerca nel database l'oggetto avente nome uguale a *name* e ritorna un riferimento persistente a tale oggetto, la validità di tale riferimento deve essere controllata attraverso la funzione membro *eos_Ref_Any::is_null()*.

2.3.2 Transazioni: classe eostrans

Ogni operazione eseguita su un database, eccettuate l'apertura e la chiusura del database stesso, devono avvenire all'interno di una transazione che viene delimitata ricorrendo alle funzioni membro della classe eostrans:

static int begin(int rdonly =0)

segnala l'inizio di una transazione. Se il parametro rdonly è posto a vero la transazione viene aperta in sola lettura e non vengono permesse modifiche al database all'interno del suo blocco;

static int commit(void)

[4] In C++ e in C un valore intero uguale a zero indica la condizione booleana *falso*, mentre un qualsiasi altro valore diverso da zero indica la condizione *vero*.

segnala che la transazione é giunta al punto di commit (é completata correttamente);

static int abort(int normal =1)

termina prematuramente una transazione attiva, facendo in modo che tutte le eventuali modifiche da essa causate vengano ignorate. Se il parametro *normal* è posto uguale a falso, dopo il completamento dell'operazione di abort termina l'esecuzione anche del programma che ha originato la transazione;

static int is_active(void)

questa funzione ritorna vero se il programma ha già dato inizio ad una transazione, falso altrimenti.

2.3.3 Oggetti di tipo file: classe eosfile

Il sistema EOS tratta gli oggetti di tipo file nello stesso modo in cui tratta gli oggetti comuni; ciò è dimostrato dalla dichiarazione stessa della classe eosfile:

class eosfile: private eosobj

è un costrutto tipico del C++ che dichara la classe eosfile come classe derivata per specializzazione dalla classe eosobj, in questo modo ne eredita sia la rappresentazione interna dei dati che i metodi di accesso.

La funzione specifica degli oggetti di tipo file è quella di raggruppare oggetti fra i quali intercorre una qualsivoglia relazione; inoltre un file può essere creato all'interno di un altro file ed avere, come accade per gli oggetti comuni, esso stesso un nome.

I metodi della classe che permettono la creazione di nuovi file sono:

static eosfile create(eosdatabase *db, const char *name=0,*
int flags=0, int ano=0)

crea un nuovo file nel database identificato da *db* e se specificato, all'interno dell'area *ano* con nome *name*;

static eosfile create(eosfile *pfile, const char *name=0,*
int flags=0, int ano=0)

crea il nuovo file all'interno di un file esistente identificato da *pfile*.

Per aprire un file si possono usare i seguenti metodi:

static eosfile open(const eosoid& oid)*

restituisce un puntatore ad un file a partire dal suo object identifier e

static eosfile open(const eosdatabase *db, const char *name)*

apre il file identificato dal nome *name* all'interno del database *db*.

2.3.4 Identificatori di oggetti: classe eosoid

Questa classe fornisce il meccanismo per identificare univocamente gli oggetti persistenti; si tratta dell'identificatore di oggetto o *oid* (object identifier). Tale classe infatti incapsula le seguenti informazioni:

- *eosspid pno;* numero di pagina in cui l'oggetto risiede;
- *unsigned ano;* numero di area in cui la pagina è contenuta
- *unsigned uno;* numero utilizzato per rendere possibile l'unicità degli oid anche quando lo spazio viene riutilizzato;
- *unsigned sno;* numero di slot, all'interno della pagina, che punta all'oggetto.

La classe fornisce, inoltre, l'overloading degli operatori di confronto relazionali quali:

int operator ==(const eosoid& oid) const
int operator !=(const eosoid& oid) const
int operator >=(const eosoid& oid) const
int operator <=(const eosoid& oid) const
int operator >(const eosoid& oid) const
int operator <(const eosoid& oid) const

Tali funzioni ritornano vero se i due oid confrontati soddisfano i corrispondenti operatori relazionali.

2.3.5 Oggetti di EOS: classe eosobj

Il sistema EOS permette di manipolare gli oggetti persistenti presenti nel database facendo ricorso alla classe eosobj che incapsula tutte le informazioni necessarie al sistema per referenziare l'oggetto stesso. Inoltre, nel momento in

cui il programmatore chiede di accedere ad un oggetto (richiede un handle), la prima ricaduta in termini pratici è la copia della pagina che contiene tale oggetto nell'area di buffering condivisa, tale pagina non viene mossa dall'area di memoria condivisa sino a che l'handle all'oggetto non é esplicitamente rilasciato.

Per esempio la seguente funzione membro permette la creazione di un nuovo oggetto

eosobj create(int size, eosdatabase *db, const void* data=0, int flags=0, int hint=0, int ano=0)*

e lo posiziona all'interno del database identificato dal puntatore *db*; l'oggetto ha dimensione pari a *size* e se non viene specificato un numero di area valido attraverso il parametro *ano*, viene fisicamente memorizzato all'interno della stessa area in cui é stato creato il database.

Un'analoga funzione permette di creare un oggetto e fare in modo che esso risulti appartenere ad uno specifico file EOS (si ricorda che un file nel sistema EOS é un oggetto esso stesso, il cui compito è raggruppare oggetti fra i quali intercorre una qualche relazione):

eosobj create(int size, eosfile *pfile, const void* data=0, int flags=0, int hint=0, int ano=0)*

Infine il seguente metodo crea un nuovo oggetto all'interno del file che contiene l'oggetto puntato da *obj*:

eosobj create(int size, eosobj *obj, const void* data=0, int flags=0, int hint=0, int ano=0)*

Il significato dei parametri non ancora visti è il seguente:

- *data*: se punta ad un'area di memoria valida, l'oggetto viene inizializzato con i primi *size* byte a partire dall'indirizzo contenuto in *data*;
- *hint*: indica che l'oggetto che viene creato può raggiungere, durante la sua vita, dimensioni maggiori di *size*. Tale parametro viene preso in considerazione dal sistema solo se è maggiore di *size*, in tal caso EOS memorizzerà fisicamente l'oggetto in una pagina sufficientemente grande da contenere un oggetto di dimensione pari a *max(size,int)*, tenendo così in considerazione la possibilità che l'oggetto aumenti di dimensione. Nel caso non fosse possibile inserire un simile oggetto all'interno di una singola pagina, il sistema passerebbe, in modo del

tutto trasparente all'applicazione, ad un metodo di memorizzazione pensato per gli oggetti di grandi dimensioni;

- *flag*: attraverso quest'ultimo parametro è possibile decidere la collocazione fisica del nuovo oggetto. I possibili valori sono:

  1. *eosobj::NEAR_LAST* Il nuovo oggetto viene posto dopo l'ultimo oggetto presente nel file;
  2. *eosobj::NEW_PAGE* Il nuovo oggetto viene collocato in una nuova pagina;
  3. *eosobj::NO_FILL* La pagina in cui l'oggetto viene memorizzato viene riservata unicamente ad esso; nessun altro oggetto sarà memorizzato in tale pagina;
  4. *eosobj::VAR_LENGHT* Viene creato un oggetto di dimensione variabile.

  Queste costanti possono essere combinate tra loro mediante gli operatori logici AND/OR.

2.3.6 Un metodo alternativo per creare nuovi oggetti: classe eos_new

In C++ l'operatore *new* fornisce il meccanismo per l'allocazione dinamica di memoria nello heap [STR91]; il sistema EOS fornisce l'overloading di tale operatore in modo che esso possa venire usato per allocare dinamicamente oggetti persistenti. Esso rappresenta quindi un ulteriore strumento a disposizione del programmatore per la creazione di oggetti persistenti, che si affianca a quello fornito dalla classe eosobj descritto in precedenza.

L'interfaccia è la seguente:

void operator new(size_t size, const eosdatabase *db)*
void operator new(size_t size, const eosfile *pfile)*
void operator new(size_t size, const eosobj *obj)*

tali metodi creano un nuovo oggetto di dimensione *size* collocandolo rispettivamente nel database indicato da *db*, nel file puntato da *pfile* o accanto all'oggetto con identificatore *obj*.

2.3.7 Primitive di accesso agli oggetti

Fino ad ora si è visto il caso di creazione di un nuovo oggetto, nel caso in cui sia invece necessario accedere ad un oggetto già esistente le possibilità offerte dal sistema EOS sono le seguenti:

static eosobj get(const eosoid& oid, int flags=0)*

ritorna un handle ad un oggetto a partire dal suo eosoid.

static eosobj get(const eosdatabase *db, const char *name,*
int flags=0)

ritorna un handle ad un oggetto il cui nome è *name* e che è memorizzato nel database indicato da *db*. La conseguenza dell'uso di una di queste due funzioni è il trasferimento dell'oggetto all'interno dell'area di memoria condivisa gestita dal server; in caso di insuccesso il valore di ritorno è NULL per entrambe.

*void *mptr(void) const*

attraverso quest'ultimo metodo si ottiene l'indirizzo di memoria dell'oggetto all'interno dell'area condivisa.

2.3.8 Riferimenti persistenti: classe eos_Ref

Questa classe è di grande rilevanza pratica; permette infatti la costruzione di oggetti complessi contenenti riferimenti persistenti ad altri oggetti. Tali riferimenti sono validi anche al di fuori dei confini di una transazione ed al di fuori dei confini di un database. Un oggetto può quindi contenere riferimenti persistenti ad oggetti appartenenti a diversi database. L'implementazione è effettuata attraverso una classe *template* cioè parametrizzata, una delle caratteristiche più interessanti introdotte nel linguaggio C++ [STR91], che permette la definizione di classi generiche e di rimandare al momento dell'effettivo uso della classe la scelta del tipo sul quale opera.

L' interfaccia della classe fornisce tutta una serie di operatori di conversione per il cui elenco si rimanda a [**?**]; qui ci si limita a presentare un esempio del loro uso (vedi figura 2.2).

```
#include<stdio.h>
#include<stdlib.h>
#include "eos.h"
#include "eos_Ref.h"
#define MAX 40
struct azioni {
   char  nome_Azione[MAX];
   int   quotazione_Odierna;
   int   beta;
   int   redditivita;
   int   quantita;
}
struct portafoglio {
   char  nome_Cliente[MAX];
   eos_Ref < eos_Ref<azioni> >      azioni_Possedute ;        // Vettore dinamico di riferimenti ad oggetti di tipo azioni
}

int nuovo_Cliente(void) {
  eosdatabase *db;
  portafoglio    *pf;
  int            azioni_Acquistate;
  db=eosdatabase::open("/databaseArea/db_Portafogli_Clienti",0,1)  // Apre il database
  pf=new(db) portafoglio;                                          // Crea nel database un nuovo oggetto di tipo portafoglio
  cout << "Inserire il numero di azioni Acquistate: ";
  cin >> azioni_Acquistate;
  pf->azioni_Possedute=   azioni_Acquistate;

  for(int i=0;i<=azioni_Acquistate;i++) {                          // Si esegue un ciclo un numero di volte uguale al
                                                                   // numero di azioni acquistate, si crea una nuova isatnza
    (pf->azioni_Acquistate)[i]=new(db) azioni;                     // persistente di azione, assegnandone il riferimento a pf
    riempi_i_Campi( (pf->azioni_Acquistate)[i] );                  // infine si chiama una specifica funzione per riempire i campi
                                                                   // della struttura azione
  }
  return(1);
}
```

Figura 2.2: Esempio di utilizzo della classe eos_Ref

Capitolo 3

OCDL e ODMG-93

Il linguaggio per la descrizione di schemi di basi di dati progettato in questa tesi è stato ottenuto applicando (parzialmente) lo standard ODMG-93 al modello OCDL; in seguito si presentano le caratteristiche principali di questi due elementi.

3.1 Il modello OCDL

OCDL (Object and Constraint Definition Language) è un modello per basi di dati orientate agli oggetti che permette la modellazione di valori complessi, dell'ereditarietà multipla e l'espressione di un numero considerevole di vincoli di integrità (vedi [BN94, **?**]). Esso è costituito dall'estensione con vincoli di integrità del modello **ODL** (*Object Definition Language*, vedi [BN94]) che integra tecniche di inferenza introdotte per le *Logiche Descrittive* (*DL*) nell'ambito dell'Intelligenza Artificiale.

3.1.1 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.

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 $\texttt{integer} \cup \texttt{string} \cup \texttt{boolean}$ (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`).

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.

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 $\texttt{employee} \preceq_\sigma \texttt{employee}$, 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.

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 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 3.1, 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 3.1 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.

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

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

Tabella 3.1: Esempio schema "Organizzazione Società"

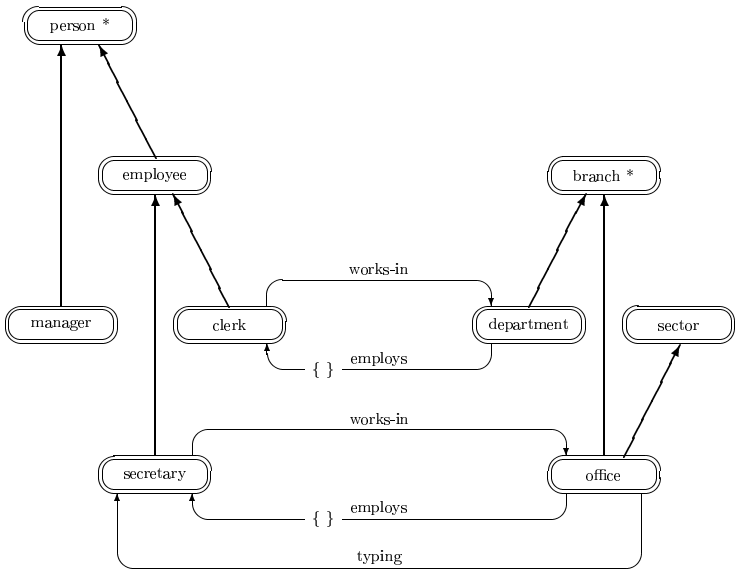

Figura 3.1: Gerarchia delle classi relativa alla struttura organizzativa di una società

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 [Car84]. 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 $N \in \mathbf{C}$.

3.1.2 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 $N \sqsubseteq_\sigma N'$.*

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.

3.1.3 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*.

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

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.

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 [CW91] su una sequenza di attributi senza essere costretti a definire ulteriori sottoclassi.
Ad esempio, con riferimento allo schema di tabella 3.1, 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:

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

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:

$$\mathcal{I}[[a_1\colon S_1, \ldots, a_p\colon S_p]] \;=\; \Big\{[a_1\colon v_1, \ldots, a_q\colon v_q] \Big| \begin{array}{r} p \leq q, v_i \in \mathcal{I}[S_i], 1 \leq i \leq p, \\ v_j \in \mathcal{V}(\mathcal{O}), p{+}1 \leq j \leq q \end{array} \Big\}$$

$$\begin{aligned}
\mathcal{I}[\forall\{S\}] &= \left\{\{v_1,\ldots,v_p\}\middle| v_i \in \mathcal{I}[S], 1 \leq i \leq p\right\} \\
\mathcal{I}[\exists\{S\}] &= \left\{\{v_1,\ldots,v_p\}\middle| \exists i, 1 \leq i \leq p, v_i \in \mathcal{I}[S]\right\} \\
\mathcal{I}[\triangle S] &= \left\{o \in \mathcal{O}\middle| \delta(o) \in \mathcal{I}[S]\right\} \\
\mathcal{I}[S \sqcap S'] &= \mathcal{I}[S] \cap \mathcal{I}[S']
\end{aligned}$$

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:

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

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

Ad esempio, il tipo–cammino ($\triangle$.`name`: *"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: `employee` $\sqcap$ ($\triangle$.`name`: *"Silvano"*). Nello stesso modo, il tipo–cammino $S = (\triangle$.`managed-by`.$\triangle$.`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 `storage` $\sqcap S$, allora σ(`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: $S' =$

($\triangle$.`managed-by`: `tmanager` $\sqcap$ ($\triangle$.`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

$$R_i : C \to S$$

Definiamo infine 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* $\mathbf{S}$.

$$
\left.
\begin{array}{ll}
\sigma & \left\{
\begin{array}{lcl}
\sigma_{\mathrm{P}}(\texttt{material}) & = & \triangle[\texttt{name: string}, \texttt{risk: integer}, \texttt{feature: \{string\}}] \\
\sigma_{\mathrm{P}}(\texttt{smaterial}) & = & \texttt{material} \\
\sigma_{\mathrm{P}}(\texttt{storage}) & = & \triangle[\texttt{managed-by: manager}, \texttt{category: string}, \\
 & & \quad \texttt{stock: \{[item: material, qty: } 10 \div 300\texttt{]\}}] \\
\sigma_{\mathrm{P}}(\texttt{sstorage}) & = & \texttt{storage} \\
\sigma_{\mathrm{P}}(\texttt{manager}) & = & \triangle[\texttt{name: string}, \texttt{salary: } 40000 \div \infty, \texttt{level: } 1 \div 15] \\
\sigma_{\mathrm{P}}(\texttt{tmanager}) & = & \texttt{manager} \sqcap \triangle[\texttt{level: } 8 \div 12]
\end{array}
\right. \\
\mathbf{R} & \left\{
\begin{array}{ll}
R_1: & \texttt{manager} \sqcap (\triangle.\texttt{level: } 5 \div 10) \rightarrow (\triangle.\texttt{salary: } 40000 \div 60000) \\
R_2: & \texttt{material} \sqcap (\triangle.\texttt{risk: } 10 \div \infty) \rightarrow \texttt{smaterial} \\
R_3: & \texttt{storage} \sqcap (\triangle.\texttt{category: ``B4''}) \rightarrow \triangle[\texttt{managed-by: tmanager}] \\
R_4: & \texttt{storage} \sqcap (\triangle.\texttt{stock.}\forall.\texttt{item: smaterial}) \rightarrow \texttt{sstorage} \\
R_5: & \texttt{storage} \sqcap (\triangle.\texttt{stock.}\forall.\texttt{qty: } 10 \div 50) \rightarrow (\triangle.\texttt{category: ``A2''})
\end{array}
\right.
\end{array}
\right\} (\sigma, \mathbf{R})
$$

Tabella 3.2: Esempio schema "Magazzino"

Schema del dominio Magazzino

Lo schema con regole mostrato in tabella 3.2 descrive una base di dati relativa a quella parte della struttura di una società che si occupa di materiali.

La regola R_1 dice che i manager con un livello compreso tra 5 e 10 devono percepire un salario maggiore di 40k e minore di 60k dollari. La regola R_2 impone che un materiale con rischio superiore o uguale a 10 sia anche nella classe dei materiali speciali (`smaterial`). In altri termini, *una condizione sufficiente* per essere un `smaterial` è quella di essere un `material` con rischio superiore o uguale a 10; pertanto per la classe base `smaterial` viene data una condizione necessaria (espressa in σ(`smaterial`)) e una condizione sufficiente (espressa tramite la regola) *differente* dalla condizione necessaria. Di conseguenza, si ottiene un diverso effetto da quello che si otterrebbe introducendo la classe `smaterial` come virtuale con la seguente descrizione

$$\sigma_{\mathrm{V}}(\texttt{smaterial}) = \texttt{material} \sqcap (\triangle.\texttt{risk: } 10 \div \infty)$$

Le regole R_3, R_4 e R_5 riguardano la classe `storage`.

Si rimanda all'appendice A per la definizione delle regole grammaticali di carattere generale che descrivono OCDL.

3.2 Lo standard ODMG-93

Lo standard **ODMG-93** (versione 1.1), presentato in [?], è nato per supportare il successo dell'approccio ad oggetti nella produzione di ODBMS (Object DataBase Management System). È il frutto del lavoro dei membri di *ODMG* (Object Database Management Group) che, raccogliendo i pregi di diverse implementazioni di ODBMS, hanno realizzato uno standard, ODMG-93 appunto, per basi di dati ad oggetti.

I membri che compongono ODMG sono:

- R. G. G. Cattel (SunSoft);
- G. Ferran (O_2 Technology);
- J. Duhl (Ontos);
- D. Wade (Objectivity);
- M. Loomis (Versant);
- T. Atwood (Object Design);

come si vede, si tratta di persone provenienti da quelle che sono tra le più importanti aziende al mondo nella produzione di software e di ODBMS in particolare. Di qui si capisce come ODMG sia posto quasi sullo stesso piano delle organizzazioni di standardizzazione ufficiali.

ODMG-93 è tuttora in via di sviluppo, quindi sono possibili modifiche future alle convenzioni che attualmente ha fissato; inoltre, non esiste nessun vincolo sul suo utilizzo poichè ODMG non è ancora un organismo di standardizzazione ufficiale; la proposta fatta da ODMG, però, se viene colta e verificata tramite implementazioni, può portare alla nascita, in tempo relativamente breve, di uno "standard di fatto" che agevolerà l'affermazione di ODBMS.

3.2.1 Perchè uno standard?

L'esigenza di pensare uno standard per basi di dati ad oggetti è nata osservando ciò che è accaduto storicamente per i DBMS relazionali. Infatti, il loro successo non si è avuto semplicemente per un maggior livello di indipendenza dei dati e per un modello dei dati più semplice rispetto ai sistemi precedenti;

la ragione principale sta nel livello di standardizzazione che hanno offerto e offrono tuttora. Avere regole di riferimento per la produzione di DBMS vuol dire poter garantire un alto grado di portabilità ed interoperabilità tra sistemi e facilitare la conoscenza e la diffusione di nuovi prodotti; la definizione dello standard per DBMS relazionali ha rappresentato una vera e propria approvazione ufficiale.

Questi fattori sono altrettanto importanti per i DBMS ad oggetti, anzi, sono ancora più importanti perchè la maggior parte dei prodotti di quest'area sono offerti da ditte giovani, perciò sconosciute, quindi la portabilità e il riconoscimento del nuovo approccio sono essenziali per il potenziale cliente. Inoltre, il raggio d'azione dei DBMS ad oggetti è di portata più ampia rispetto a quella dei DBMS relazionali in quanto integra linguaggi di programmazione e sistemi di basi di dati e include tutti i dati e tutte le operazioni di un'applicazione. Di conseguenza, uno standard è di importanza fondamentale per il successo dell'industria di basi di dati ad oggetti.

ODMG ha voluto dare uno stimolo a tale industria affinchè si ottenga uno standard in un arco di tempo breve rispetto ai molti anni che servirebbero ad un organismo di standardizzazione tradizionale. Sostanzialmente viene suggerita un'interfaccia comune per ODBMS che il programmatore userà per scrivere le proprie applicazioni.

3.2.2 Obiettivi di ODMG-93

Lo scopo è quello di produrre un insieme di regole che consentano a un produttore di software di ODBMS di scrivere applicazioni *portabili*, cioè applicazioni che, nella loro completezza, possono funzionare su diversi ODBMS. Appare chiaro che non si cerca di produrre ODBMS identici; infatti, sarebbe una meta quasi impossibile da raggiungere data la vastità del campo applicativo del paradigma ad oggetti (a seconda delle aree, gli ODBMS avranno caratteristiche diverse rispetto alle prestazioni, ai linguaggi supportati, alle funzionalità uniche per particolari segmenti di mercato, agli strumenti di costruzione di applicazioni e così via). Lo scopo primario è uno solo: la portabilità del codice.

Come detto in precedenza, il lavoro svolto dai membri di ODMG si basa principalmente sulla combinazione delle più importanti caratteristiche degli ODBMS attualmente disponibili. Questi prodotti offrono implementazioni che permettono di provare sul campo i componenti dello standard.

3.2.3 Definizioni

È importante definire la competenza degli sforzi di ODMG, poichè gli ODBMS forniscono un'architettura sostanzialmente diversa da quella degli altri DBMS (si può dire che gli ODBMS costituiscono uno sviluppo rivoluzionario piuttosto che evoluzionistico). Invece di supportare solo un linguaggio di alto livello, come il linguaggio SQL, per la manipolazione di dati, un ODBMS integra, in modo trasparente, le capacità della base di dati con il linguaggio di programmazione. Questa trasparenza rende superfluo utilizzare un DML (Data Manipulation Language) separato, non richiede la necessità di copiare e tradurre esplicitamente i dati tra base di dati e rappresentazioni del linguaggio di programmazione e favorisce vantaggi di prestazioni significativi tramite il caching dei dati nelle applicazioni (vedi figura 3.2). Un ODBMS include anche le capacità del linguaggio di interrogazione dei sistemi relazionali, perciò il modello del linguaggio di interrogazione è più potente; ad es. incorpora liste, vettori e insiemi.

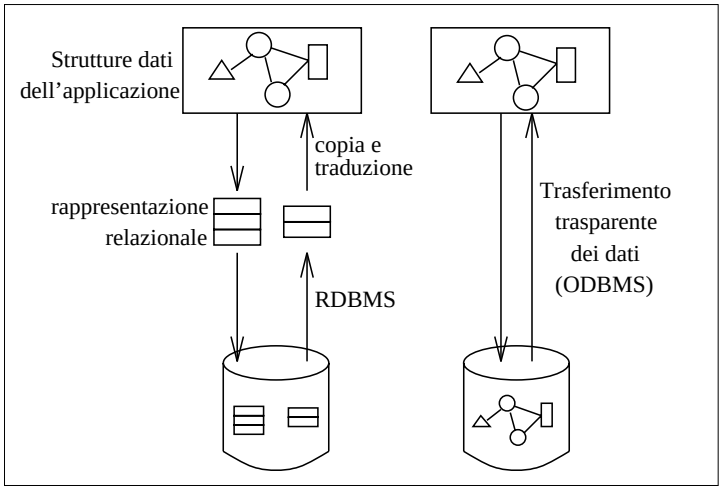

Figura 3.2: Confronto tra architetture di DBMS

Riassumendo, ODMG definisce un *ODBMS* come un DBMS con le seguenti caratteristiche:

- integra le capacità della base di dati con le capacità del linguaggio di programmazione orientato agli oggetti;
- fa apparire gli oggetti della base di dati come gli oggetti del linguaggio di programmazione (per uno o più linguaggi di programmazione esistenti);
- in modo trasparente, estende il linguaggio con la persistenza e il recupero dei dati, il controllo di concorrenza, le interrogazioni associative e altre capacità tipiche delle basi di dati.

Al contrario dei sistemi di basi di dati relazionali estesi, gli ODBMS richiedono standard basati sull'integrazione tra sintassi, semantica e compilatori dei linguaggi di programmazione esistenti. ODMG ha integrato gli ODBMS con i linguaggi C++, C, Smalltalk e LISP. Ci sono alcune sovrapposizioni con i sistemi di basi di dati relazionali estesi, poichè si è voluto che il linguaggio di interrogazione mantenesse compatibilità con l'evoluzione dell'SQL standard. Comunque, affinchè il paradigma ad oggetti sia veramente vantaggioso, deve incorporare l'applicazione nella sua interezza, non solo la parte riguardante la base di dati.

Lo standard prevede la definizione di alcuni componenti principali:

- *modello ad oggetti (OM).* È stato usato il modello ad oggetti di OMG come base del modello dello standard. Il nocciolo del modello di OMG fu progettato per essere il denominatore comune tra sistemi di basi di dati ad oggetti, linguaggi di programmazione ad oggetti e altre applicazioni. Mantenendo l'architettura originale, sono state aggiunte componenti (ad es. le *relationship* tra oggetti) per supportare i bisogni del nuovo modello ad oggetti;
- *linguaggio di definizione di oggetti (ODL)*[2]. È chiamato così per distinguerlo dai linguaggi di definizione dei dati delle basi di dati tradizionali, detti DDL. Si usa il linguaggio di OMG per la definizione dell'interfaccia (IDL) come base per la sintassi;

[2]si indicherà con **ODL-93** l'ODL di ODMG-93 al fine di non creare confusione con il modello ODL predecessore di OCDL

- *linguaggio di interrogazione di oggetti (OQL).* È un linguaggio dichiarativo (non procedurale) per l'interrogazione e l'aggiornamento di oggetti della basi di dati. Ovunque possibile, è stato usato l'SQL standard relazionale come base per OQL, sebbene quest'ultimo debba supportare capacità più potenti. Non si è usato lo standard relazionale esteso che è in via di sviluppo (SQL3) a causa dei limiti nel modello dei dati che considera e a causa del suo "bagaglio" storico. Si spera che SQL3 e OQL possano convergere in futuro;
- *collegamento con il linguaggio C++.* È stato dimostrato che il più importante linguaggio di programmazione per ODBMS è il C++. Perciò, si spiega come scrivere codice C++ portabile che manipola oggetti persistenti. Questo linguaggio si chiama C++ OML (C++ Object Manipulation Language). Il collegamento con il C++ include anche una versione di ODL che usa la sintassi del C++, un meccanismo per invocare OQL e procedure per eseguire operazioni su basi di dati e su transazioni.
- *collegamento con il linguaggio Smalltalk.* Attualmente è solo abbozzato, ma si è reso necessario in quanto esistono applicazioni per le quali Smalltalk è il linguaggio di programmazione più appropriato. È possibile leggere e scrivere la stessa base di dati sia da Smalltalk che da C++ purchè il programmatore usi un sottoinsieme di tipi di dati comuni.

In futuro verranno pensati collegamenti con altri linguaggi di programmazione quali Pascal, CLOS e IDL. Si noti che, diversamente dai DBMS relazionali, i linguaggi di manipolazione dei dati degli ODBMS sono fatti su misura per gli specifici linguaggi di programmazione allo scopo di fornire un ambiente singolo e integrato per la programmazione e per la gestione dei dati. Ciò non vuol dire credere esclusivamente in una sintassi universale per DML. Si vuole andare oltre quanto fatto per i sistemi relazionali supportando sia un modello ad oggetti unificato per la condivisione dei dati tra linguaggi di programmazione, sia un linguaggio di interrogazione comune.

3.3 OCDL e ODMG-93 a confronto

OCDL e ODMG-93 presentano ciascuno caratteristiche peculiari che rendono abbastanza marcata la loro distinzione. ODMG-93, infatti, è in grado di

trattare, oltre agli aspetti strutturali, anche gli aspetti comportamentali della realtà da modellare dando la possibilità di definire *operazioni* e *relazioni* sugli oggetti; in aggiunta, consente di dichiarare attributi chiave di una classe per velocizzare il recupero delle informazioni durante un'interrogazione della base di dati. Da parte sua OCDL, grazie all'introduzione delle regole di integrità, permette di esprimere una gran classe di vincoli che sarebbero nascosti in metodi considerati da ODMG-93 direttamente nello schema di una base di dati facilitando, ad esempio, l'ottimizzazione di interrogazioni e la verifica di consistenza dello schema (vedi [?]).

Considerando, invece, l'"intersezione" tra i concetti considerati dai due modelli, si possono fare alcune osservazioni a ODMG-93. Infatti, la definizione di tipo basata esclusivamente su tipi nominati (proprio come avviene anche in C e in C++), siano essi definiti dal sistema o dall'utente, non ammette la descrizione nidificata delle parti componenti. Si supponga, ad esempio, di voler definire la classe *Persona* che ha, come informazioni caratteristiche, il nome, l'età e la residenza (via, numero e città). Una possibile definizione tale che rispetti lo standard ODMG-93 è:

```
interface Persona {
  Tuple [String           nome;
         Long             eta;
         Indirizzo        residenza;]
};
```

L'attributo *residenza* è definito di tipo *Indirizzo*, che a sua volta può essere dichiarato come una ennupla la cui definizione è la seguente:

```
Typedef Struct Indirizzo {
  String       via;
  Long         numero;
  String       citta;
};
```

La sintassi definita nella presente tesi permette di definire la classe *Persona* anche così:


```
interface Persona {
  Tuple [String                      nome;
         Range [0->150]              eta;
         Struct {String   via;
                 Long     numero;
                 String   citta;}    residenza;]
};
```

Questa seconda definizione ha il vantaggio di ammettere la descrizione nidificata dell'attributo complesso *residenza* senza dovere dichiarare il tipo supplementare *Indirizzo*. In una base di dati, che generalmente contiene tanti tipi, la mancanza di questa possibilità (come avviene in ODMG-93) darebbe origine a uno schema più complesso di quello ottenuto dalla progettazione concettuale a causa dell'introduzione di "tipi di appoggio" (nell'esempio, il tipo *Indirizzo*); questi ultimi non sono significativi per il modello della realtà che si sta costruendo e la conseguenza è quella di uno schema ampliato a causa dell'*esplosione* di nomi.

Un'altra osservazione, sebbene di rilevanza molto minore, riguarda la mancanza del tipo *intervallo di interi*: questo tipo risulta utile per agevolare l'ottimizzazione semantica delle interrogazioni (vedi [?]) in quanto permette di descrivere propriamente sottotipi (risulta facilitato il raffinamento di attributi nelle gerarchie di ereditarietà fra classi).

Si rimanda all'appendice B per la descrizione del linguaggio di definizione di schemi progettato.

Capitolo 4

Il componente OCDL-PSSManager

OCDL-PSSManager (OCDL-*Persistent Schema Storage* Manager) è un interprete in grado di interagire con un utente e di comprendere una grammatica, derivata dallo standard per basi di dati ad oggetti ODMG-93, per la definizione di uno schema di base di dati; inoltre, può eseguire una serie di comandi per la gestione di altri aspetti riguardanti le basi di dati, quali la persistenza e la verifica di coerenza dello schema.

La necessità di gestire l'interazione macchina-utente attraverso un interprete, ha subito orientato verso l'utilizzo di due strumenti, *Lex* e *Yacc*[1], originariamente nati in ambiente Unix ma disponibili ora sulla maggior parte delle piattaforme.

Il restante codice necessario per la gestione vera e propria della base di dati è stato implementato in ANSI C++; la scelta di tale linguaggio è stata fatta non solo per la sua capacità orientata agli oggetti che consente una gestione ottimale di progetti di medie-grandi dimensioni, ma anche per la necessità di interfacciarsi con le librerie del sistema EOS, anch'esse programmate e compilate in C++.

[1]in realtà si sono usati i corrispondenti moduli forniti dalla GNU (Free Software Foundation, Inc.) chiamati rispettivamente *Flex* e *Bison*; tuttavia i concetti espressi successivamente restano ugualmente validi poichè sono di carattere generale

4.1 Lex e Yacc

Lex e **Yacc** sono strumenti in grado di creare procedure in linguaggio C che analizzano ed interpretano un flusso di input. Essi consentono di produrre compilatori o interpreti a partire da moduli di dimensioni molto più ridotte di quelle dei moduli necessari per implementare, sempre in linguaggio C, gli stessi compilatori o interpreti in modo esteso.

Lex e yacc sono generalmente destinati ad operare congiuntamente. Il primo ha come input un file che contiene le regole di analisi lessicale e genera le necessarie funzioni C in grado di leggere i singoli byte del flusso di input e di convertirli in *token* (che costituiscono l'unità di informazione nella comunicazione tra i moduli prodotti da lex e quelli prodotti da yacc). Il secondo, invece, legge un file che contiene la specifica della grammatica di un linguaggio e genera le necessarie routine C per effettuare il parsing, cioè per ragruppare i token in sequenze. Il file di input di yacc specifica quali sequenze di token hanno un significato univoco e in corrispondenza ad esse invoca funzioni (codificate dal programmatore in linguaggio C o C++) che eseguono le opportune azioni.

Ad esempio, si supponga che l'utente voglia *aprire* (cioè accedere a) una base di dati di nome *azienda*. Il comando può essere dato all'interprete digitando la sequenza "*open azienda <CR>*"; il modulo prodotto da lex genera due token (uno per la parola *open* e l'altro per la parola *azienda*) e li passa al modulo prodotto da yacc che li mette in sequenza; tale sequenza di token viene riconosciuta come significativa e ha l'effetto di mandare in esecuzione la routine di apertura di una base di dati.

Per convenzione il suffisso del file che viene elaborato con lex è *.lex* o *.l*, mentre quello del file destinato ad essere processato da yacc è *.yacc* o *.y*. I due file scritti per la realizzazione dell'interfaccia macchina-utente di OCDL-PSSManager sono *syntxanlz.lex* e *gramm.yacc*. Lex crea una routine chiamata `yylex` contenuta in un file chiamato *lex.yy.c*, mentre yacc genera una routine `yyparse` e contenuta nel file *y.tab.c*. Questi due file vengono compilati e collegati con lo specifico codice fornito dal programmatore che implementa il comportamento dell'interprete o del compilatore. Si faccia riferimento alla figura 4.1 per una rappresentazione grafica di tali concetti.

In figura 4.2 è presentato un semplice esempio di un analizzatore lessicale implementato attraverso l'uso di lex; in dettaglio il significato delle istruzioni è:

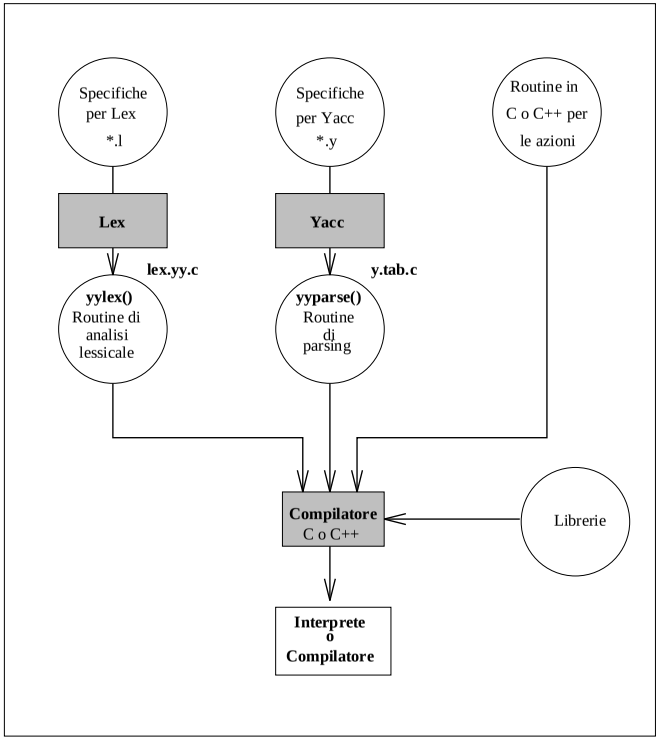

Figura 4.1: Interazione fra lex e yacc


```
1   %{
2   #define NUMERO      10
3   #define COMMENTO    11
4   #define TESTO       12
5   #define COMANDO     13
6   %}
7   %%
8   [ \t]+                          ;
9   [0-9]+                          |
10  [0-9]+\.[0-9]+                  |
11  \.[0-9]                         { return NUMERO;}
12
13  //                              { return COMMENTO;}
14
15  [a-zA-Z][a-zA-Z0-9]+            { return COMANDO;}
16
17  \n                              { return '\n';}
18
19  %%
```

Figura 4.2: Esempio di specifica Lex

- dalla riga 1 alla 6 vi sono delle normali *define* con le quali viene associato un valore ad un identificatore (che rappresenta un token);
- nella riga 8 vi è la prima regola che identifica un token; si tratta del *blank* (o carattere di spaziatura) e del carattere corrispondente al tasto <Tab>: allorchè viene individuato un blank o un <Tab> o una sequenza di questi due caratteri, la routine yylex non farà altro che ignorarli;
- dalla riga 9 alla 17 vi sono altre regole lessicali, alle quali sono associate azioni; precisamente si tratta di varie istruzioni che ritornano alla funzione che ha invocato la routine yylex (di solito yyparse) gli identificatori dei token riconosciuti. Ad esempio, nel caso di linea 15, una qualunque sequenza che inizi con una lettera e continui con un numero arbitrario di lettere e/o numeri, identifica un comando; l'azione associata è quella che ritorna al parser il valore numerico associato al token.

Yacc si aspetta di ricevere in input un file contenente la descrizione di una grammatica in una forma simile alla BNF (*Backus-Naur Form*). Si tratta di una serie di *regole di produzione* composte da una parte non terminale, quella posta a sinistra, seguita dalla sua definizione che consiste da uno o più simboli non terminali, se richiamano altre regole, o terminali, se corrispondono ad un token.

Una semplice grammatica che implementa una altrettanto semplice calcolatrice è la seguente:

$$
\begin{aligned}
< \text{calcolatore} > &::= < \text{operazione} > \mid \\
&\quad < \text{operazione} >< \text{calcolatore} > \\
< \text{operazione} > &::= < \text{operando} > \mid \\
&\quad < \text{operatore} > \\
< \text{operando} > &::= < \mathit{NUMERO} > \\
< \text{operatore} > &::= + \mid - \mid * \mid /
\end{aligned}
$$

Nell'esempio le parole scritte in maiuscolo rappresentano simboli terminali ovvero token.

scanner.lex

```
1   %%
2   [0-9]+      {
3                   sscanf(yytext,"%d",&yylval);
4                   return (INTERO);
5                   }
6
7   \n              return ('\n');
8
9   +               return PLUS;
10  -               return MINUS;
11  /               return DIV;
12  *               return MUL;
13  =               return PRINT;
14
15  quit            return 0;
16  .               printf("Errore sintattico\n");
17
18
19
20
21
22
23
24
```

parser.yacc

```
%{
int totale = 0;
%}

% token INTERO, PLUS, MINUS, MUL, DIV, PRIN

%%

input:   /* Linea vuota */
         | input  linea
         ;

linea:     '\n'
         | exp '\n'     { printf("= %d\n",totale);}
         ;

exp:     INTERO             { totale += $1; }
         | PLUS INTERO      { totale += $2; }
         | MINUS INTERO     { totale -= $2;}
         | MUL INTERO       { totale = ( totale * $2
         | DIV INTERO       { totale = (totale / $2)
         | PRINT             { totale = 0 ; }
         ;

%%
```

Figura 4.3: Specifiche Lex e Yacc per un semplice calcolatore

La riscrittura di tale grammatica in modo che sia interpretabile da yacc è operazione immediata, ed è riportata in figura 4.3 insieme con il corrispondente file contenente la specifica lex per il riconoscimento dei token.

L'interpretazione delle linee del codice è la seguente, iniziando dalla parte sinistra della figura, ovvero dal file scanner.lex:

- righe 2÷5 : la riga 2 specifica che nel caso si riconosca nell'input una sequenza arbitraria di uno o più numeri, si notifichi al parser che è stato letto un numero intero e si operi in modo tale da ritornare al parser stesso il numero che è stato letto. La variabile *yytext* contiene il flusso o *stream* di caratteri letto generalmente da standard input; la variabile *yylval* è una variabile esterna, dichiarata nella parte di codice relativa al parser ed attraverso la quale si è in grado di passare al parser il numero letto da input, dopo averlo convertito in intero. Nella riga 4 il valore di ritorno notifica al parser che è stato riconosciuto il token corrispondente ad un numero intero;

- righe 7÷13: queste istruzioni permettono riconoscere i diversi operatori matematici e notificare al parser quale di questi è stato incontrato;

- riga 15: riconosce il comando di terminazione del funzionamento della calcolatrice;

- riga 16: qualunque altra sequenza di caratteri non è significativa e viene notificato all'utente che è stato commesso un errore sintattico.

La parte destra della figura contiene la specifica della grammatica vista in precedenza:

- riga 5: dichiara i nomi dei token che corrispondono ai valori di ritorno visti nel file scanner.lex; in questo modo le *define*, che nell'esempio di figura 4.2 erano state espressamente codificate, vengono generate automaticamente da yacc;

- righe 9÷10: contengono la definizione dell'input della calcolatrice: esso può essere formato da una linea vuota e l'effetto è di ignorala, oppure da una o più linee di input vere e proprie, indicate dal simbolo non terminale *linea*;

- riga 13: al simbolo non terminale *linea* è associata un'azione: semplicemente la stampa su standard output del valore della variabile *totale*; tuttavia esso richiama, prima di effettuare la stampa, il simbolo non terminale *exp* a cui sono associate le azioni di calcolo vere e proprie;

- righe 16÷22: contengono un elenco di simboli terminali o token ad ognuno dei quali è associata una specifica azione. La notazione *' $n '* consente di referenziare le singole parole, o meglio *statement*, che compongono la regola di produzione per un simbolo, si tratta di una notazione di tipo posizionale. Ad esempio in riga 18 *' $2 '* si riferisce al token *INTERO*. L'effetto dell'azione associata è di incrementare la variabile *totale* di un valore pari a *INTERO*.

Il tema del presente lavoro non consente di descrivere in dettaglio tutte le possibilità offerte da questi due importanti strumenti di sviluppo, si rimanda quindi a [?].

4.2 OCDL-Designer

Il modulo **OCDL-Designer** presentato in [?], basandosi sul modello OCDL, è in grado di:

- operare il controllo di consistenza implementando gli algoritmi di generazione dello schema canonico e di incoerenza;

- calcolare la classificazione minimale rispetto alla relazione di ereditarietà implementando l'algoritmo di sussunzione.

OCDL-PSSManager utilizza OCDL-Designer limitatamente al controllo di coerenza di uno schema di basi di dati. È quindi opportuno descriverlo brevemente.

4.2.1 Cenni su OCDL-Designer

OCDL-Designer (*Object and Constraint Description Language* Designer) è un ambiente software per l'acquisizione e la modifica di uno schema descritto con il linguaggio OCDL e consente di:

- verificare che lo schema sia *consistente*, cioè verificare che esista almeno uno stato della base di dati tale che ogni tipo, classe e regola abbia un'estensione non vuota;
- ottenere la *minimalità* dello schema rispetto alla relazione `isa`, cioè per ogni tipo (classe) viene calcolata la "giusta" posizione nella tassonomia dei tipi (classi):
  - il tipo (classe) viene inserito "sotto" tutti i tipi (classi) che specializza;
  - il tipo (classe) viene inserito "sopra" tutti i tipi (classi) che lo specializzano.

In altre parole, il tipo viene inserito nella gerarchia di ereditarietà in posizione tale che risulti discendente di tutti i tipi (classi) che specializza e predecessore di tutti i tipi (classi) che lo specializzano.

Il programma è diviso in due sottocomponenti funzionali principali che corrispondono a due fasi distinte: il primo permette il controllo della coerenza dello schema generandone una forma semplificata in cui tutti i riferimenti a nomi di tipi sono sostituiti con le corrispondenti descrizioni (forma *canonica*). Il secondo componente, partendo dallo schema OCDL canonico, calcola le relazioni di sussunzione e le relazioni `isa` minimali che intercorrono fra i tipi (classi).

Regole di integrità

Nel paragrafo 3.1.3 una regola è stata denotata come una coppia antecedente e conseguente $R = (S^a, S^c)$ in cui sia S^a che S^c sono tipi di $\mathbf{S}(\mathbf{A}, \mathbf{B}, \mathbf{N})$. I tipi S^a e S^c vengono descritti con due definizioni separate[2]: essi possono essere solo classi virtuali e tipi-valore, in quanto si possono escludere le classi primitive (poichè non possiedono la semantica delle regole) ed i tipi-base (poichè di nessuna rilevanza concreta) senza perdere in generalità.

Si introducono cosí quattro tipologie che descrivono le regole di integrità:

`antev:` antecedente di una regola di tipo classe virtuale.

[2]N.B. si sta parlando delle convenzioni adottate da OCDL-Designer; il componente sviluppato accetta definizioni di regole solo nella loro interezza per permettere di sfruttare a pieno le potenzialità di OCDL

`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.

OCDL-Designer 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`. Inoltre, per le regole di tipo virtuale che hanno l'antecedente associato ad una classe della base di dati, questa associazione è vera anche per il conseguente, sebbene nella notazione introdotta essa non sia direttamente esplicitata: per mantenere questa semantica, quando introduciamo in OCDL-Designer conseguenti di tipo virtuale (tipo consv) occorre inserire esplicitamente tutte le eventuali classi associate all'antecedente. La relazione tra antecedente e conseguente di una stessa regola viene garantita dal nome di tipo che è lo stesso per entrambi, escluso l'ultimo carattere che vale 'a' per gli antecedenti e 'c' per i conseguenti.

Ad esempio, se si vuole definire la regola

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

occorre inserire nello schema due definizioni, una per l'antecedente e una per il conseguente:

`antev` r2a = `material` $\sqcap \triangle$ [`risk` : $10 \div \infty$]
`consv` r2c = `material` $\sqcap$ `smaterial`

Tipo-cammino

Il linguaggio OCDL, descritto nella teoria, prevede, fra gli altri, il tipo-cammino con la notazione $p = e_1 . e_2 . \ldots . e_n$, con $e_i \in \mathbf{A} \cup \{\langle\rangle, \triangle, \forall, \exists\}$.
In fase di implementazione il tipo-cammino viene tradotto nella notazione tradizionale applicando le seguenti regole:

$$\begin{aligned} \rho(S) &= S \text{ if } S \in \mathbf{B} \cup \overline{\mathbf{C}} \\ \rho(\triangle S) &= \triangle \rho(S) \\ \rho([a_1 : S_1, \ldots, a_n : S_n]) &= [a_1 : \rho(S_1), \ldots, a_n : \rho(S_n)] \end{aligned}$$

$$\begin{aligned}
\rho(S \sqcap S') &= \rho(S) \sqcap \rho(S') \\
\rho(\{S\}) &= \{\rho(S)\} \\
\rho((\epsilon\colon S)) &= \rho(S) \\
\rho((\forall\colon S)) &= \{\rho(S)\} \\
\rho((\exists\colon S)) &= \{\exists\colon \rho(S)\} \\
\rho((a\colon S)) &= [a\colon \rho(S)] \\
\rho((e.p\colon S)) &= \rho((e\colon (p\colon S)))
\end{aligned}$$

Ad esempio la definizione

$$(\triangle.\texttt{stock}.\forall.\texttt{item}\colon \texttt{smaterial})$$

viene tradotta così:

$$\triangle[\texttt{stock}\colon \{[\texttt{item}\colon \texttt{smaterial}]\}]$$

Classi fittizie

Nella teoria una *classe fittizia* $\overline{C}$ è stata definita come un nuovo nome di classe base $\overline{C}$ la cui descrizione è $\triangle\top$, ossia:

$$\sigma(\overline{C}) = \triangle\top$$

A livello di implementazione, quando viene creato un elemento *atomo fittizio*, non viene inserito come nome dello schema, ma può comparire solo come elemento di una descrizione.

L'esecuzione inizia con la fase di generazione dello schema canonico mediante l'applicazione delle funzioni ricorsive ι e μ definite in [?], le quali operano alternativamente fino al raggiungimento del punto fisso della trasformazione canonica. La generazione di tale schema permette inoltre di determinare quali sono i tipi (classi) incoerenti, quelli, cioè, la cui descrizione è inconsistente e che di conseguenza avranno sempre estensione vuota. Durante l'elaborazione vengono comunicati a video eventuali messaggi di errore e incoerenze rilevate.

Non si approfondisce ulteriormente l'argomento poichè esula dalle tematiche trattate nella presente tesi; per una descrizione molto più dettagliata del componente OCDL-Designer si rimanda a [?].

4.3 OCDL-PSSManager: architettura e funzionalità

Il componente **OCDL-PSSManager** è stato sviluppato in linguaggio ANSI C++ su piattaforma hardware SUN Sparc 20, con sistema operativo Solaris 2.3 (Sun OS 5.3). I sorgenti sono stati compilati con gcc (compilatore C e C++) realizzato dalla GNU, (Free Software Foundation, Inc.) versione 2.6.3.

In figura 4.4 è rappresentata l'architettura interna di OCDL-PSSManager e la sua interazione con il componente ODB-QOptimizer, di cui fa parte OCDL-Designer. La linea sottile e tratteggiata rappresenta i confini del componente OCDL-PSSManager.

Il blocco "INTERPRETE" costituisce il cuore del sistema ed è formato dalle routine `yylex()` e `yyparse()`. La prima ha il compito di filtrare i comandi immessi dall'utente per eseguire l'analisi sintattica e generare token; la seconda riceve i token, ne effettua l'analisi semantica ed invoca le funzioni contenute negli altri moduli per compiere le opportune azioni.

La modifica degli schemi di memoria centrale e di memoria di massa può avvenire solo, rispettivamente, attraverso i moduli "grammfunct" ed "eos_funct". Per quanto riguarda lo schema persistente, anche la semplice lettura deve essere eseguita tramite "eos_funct", mentre lo schema di memoria centrale è accessibile invocando i metodi facenti parte della sua interfaccia.

Lo "schema di memoria centrale" è costituito da 5 oggetti (alberi binari bilanciati, vedi paragrafo 4.3.7); in figura è rappresentato come un'entità a sè stante poichè ogni oggetto è corredato da metodi che eseguono sia operazioni elementari di accesso ai dati, sia operazioni di livello superiore (ad es. visualizzazione parziale o totale dei dati, traduzione delle informazioni nel formato di memorizzazione (vedi freccia verso il modulo "eos_funct"), etc.).

La scelta per OCDL-PSSManager di un linguaggio (il C++) diverso da quello utilizzato da OCDL-Designer (il C) ha reso necessaria la definizione di un' "interfaccia" di comunicazione tra i due programmi per la conversione delle strutture di memoria centrale.

Il modulo "primitive di accesso" implementa le routine che permetteranno a ODB-QOptimizer di accedere allo schema persistente su memoria di massa, una volta che verrà creato il collegamento indicato dalle frecce tratteggiate.

Nei prossimi paragrafi si riporta la descrizione dettagliata del software

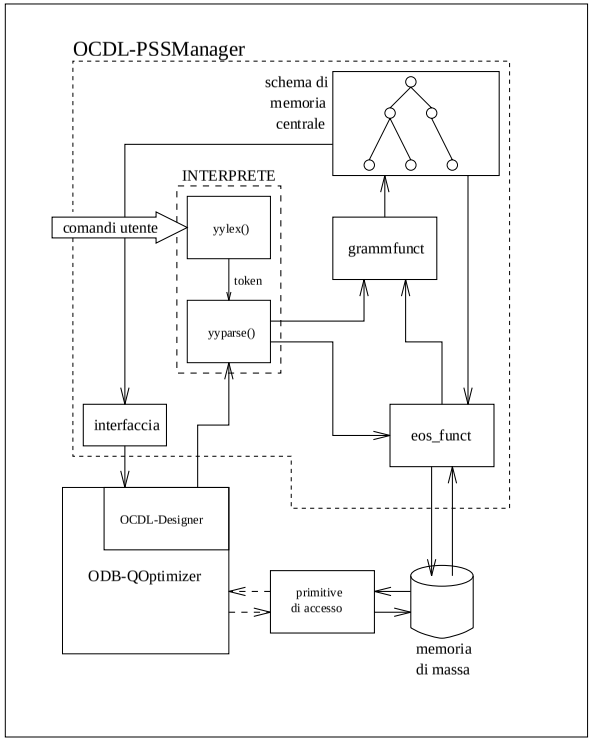

Figura 4.4: Architettura di OCDL-PSSManager

che compone OCDL-PSSManager utilizzando, per la parte riguardante le funzionalità, il modello *DFD* (*Data Flow Diagram*).

4.3.1 Il modello DFD

È uno degli strumenti più utilizzati nell'ingegneria del software per supportare la fase fondamentale di *analisi dei requisiti* del ciclo di vita del software; in questa fase si pongono in forma ordinata le conoscenze ottenute dallo studio preliminare del problema che ci si propone di risolvere.

I modelli si dividono in:

- **modelli informali**. Organizzano la conoscenza mettendo un po' di ordine;
- **modelli semi-formali**. Sono basati su costrutti grafici con i quali è possibile schematizzare, in modo non ambiguo, una certa quantità di conoscenza, mentre la parte restante rimane espressa in modo informale (servono descrizioni fornite in altra maniera);
- **modelli formali**. Strumenti a base matematico-logica (formalismi o linguaggi) che producono una forma della conoscenza che può essere processata direttamente.

Il modello DFD si colloca nella categoria dei modelli semi-formali. I grossi pregi di cui gode sono la semplicità di realizzazione/comprensione e la possibilità che offre di dettagliare i problemi finchè si vuole (di solito il dettaglio arriva fino alla rappresentazione delle procedure o funzioni; oltre non è significativo andare). A fronte di questi pregi, però, evidenzia alcune lacune: non rappresenta l'ordine di esecuzione delle funzioni; presenta solo la situazione di regime del software; non descrive i dati che usa; infine, anche giunti al massimo dettaglio, occorre spiegare a parte il compito svolto da ciascuna funzione. Tuttavia, per l'importanza fondamentale dei suoi pregi, l'utilizzo del modello DFD è di basilare importanza nell'industria per la produzione del software, tant'è che è usato in quasi tutti gli strumenti di CASE (Computer Aided Software Engineering).

Tutte le volte che si fa uso di DFD occorre rispettare la seguente legge: i flussi di dati d'ingresso e d'uscita di una funzione/processo devono essere sempre mantenuti quando si aumenta il grado di dettaglio del diagramma. In figura 4.5 si riportano i costrutti fondamentali con i quali si può disegnare un diagramma.

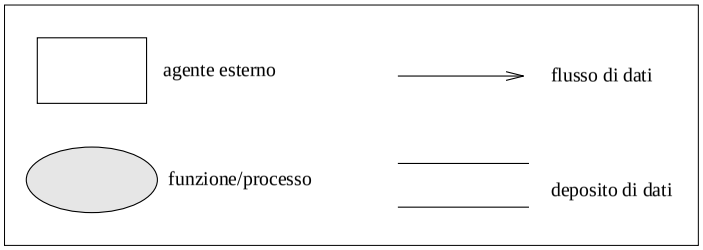

Figura 4.5: Elementi di un DFD

4.3.2 L'architettura del software di OCDL-PSSManager

OCDL-PSSManager interagisce con OCDL-Designer; il sistema che risulta dal loro uso combinato svolge essenzialmente due funzionalità:

- comprende la descrizione e garantisce la persistenza di uno schema di base di dati (OCDL-PSSManager);
- permette il controllo di coerenza dello schema stesso (OCDL-Designer).

In figura 4.6 è rappresentato il DFD globale che evidenzia l'interazione dei due componenti sopra citati. Si nota che OCDL-PSSManager riceve comandi dalla console, li interpreta ed esegue le opportune azioni. A seconda del comando, OCDL-PSSManager può agire sullo schema persistente di una base di dati (contenuto nel deposito "SCHEMA PERSISTENTE"), può registrare lo stesso schema in formato testo (sul deposito "ARCHIVIO DICHIARAZIONI") e può fare intervenire OCDL-Designer per il controllo di coerenza. I flussi di dati rappresentati hanno i seguenti significati:

- *comandi*: comandi immessi dall'utente (comprendono anche le dichiarazioni di tipi dello schema);
- *risultati*: messaggi visualizzati da OCDL-PSSManager;
- *def. tipo e nome*: descrizione e nome di un tipo dello schema;
- *dichiarazioni*: sequenza delle dichiarazioni immesse dall'utente per generare lo schema della base di dati;

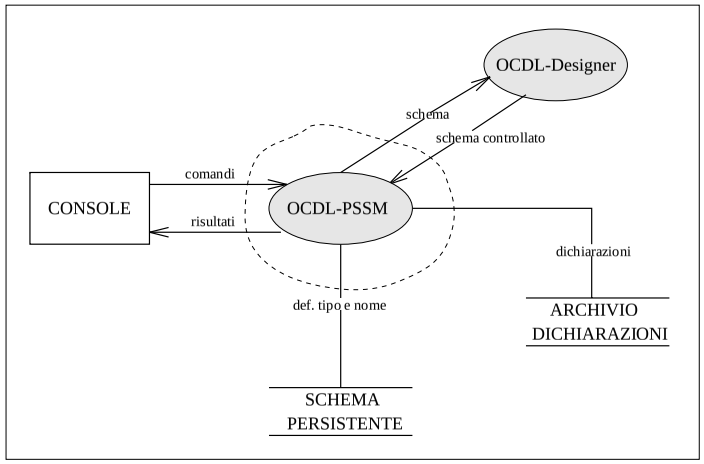

Figura 4.6: architettura funzionale del sistema con OCDL-PSSManager e OCDL-Designer

- *schema*: descrizione completa dello schema della base di dati su strutture di memoria tipiche di OCDL-Designer;
- *schema controllato*: descrizione dello schema dopo che è stato manipolato dal modulo di verifica della consistenza.

Dettagliando il processo "OCDL-PSSManager", si ottiene la figura 4.7; la linea tratteggiata aiuta a notare che sono stati mantenuti i flussi di dati in ingresso e in uscita, rispetto al diagramma precedente.

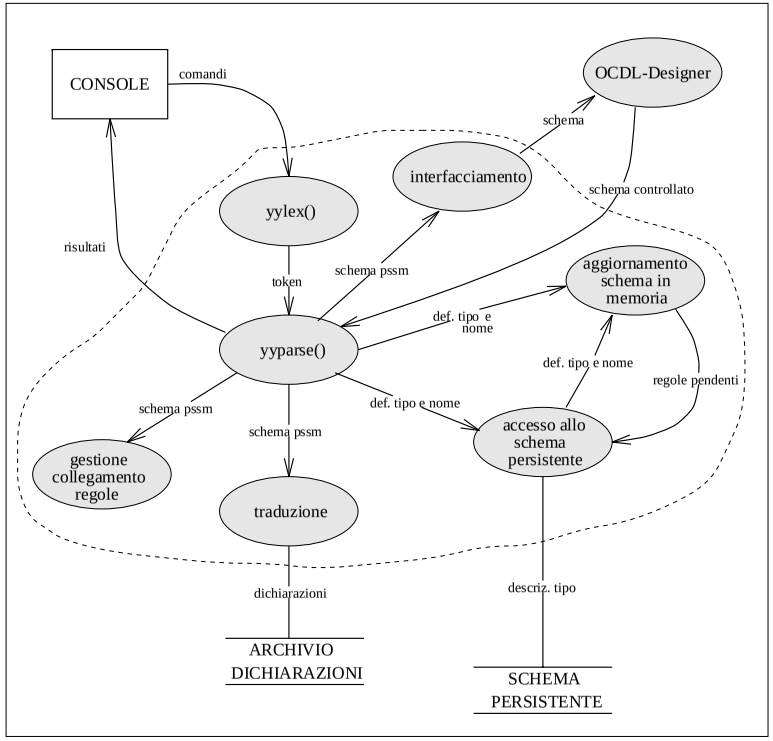

Figura 4.7: Visione dettagliata di OCDL-PSSManager

I flussi di dati aggiuntivi sono:

- *token*: elementi di comunicazione tra le funzioni di analisi sintattica e semantica;
- *schema pssm*: descrizione completa dello schema sulle strutture di memoria utilizzate da OCDL-PSSManager;
- *regole pendenti*: nomi delle regole che sono state classificate grazie all'immissione del tipo corrente; se sono presenti anche nello schema persistente, le loro descrizioni devono essere aggiornate.

La funzione "gestione collegamento regole" crea o elimina un collegamento (solo a livello di memoria centrale) fra i tipi e le rispettive regole. Viene eseguita all'invocazione dei comandi *attach rules* o *detach rules*. Non si riportano le funzioni elementari di gestione della base di dati quali apertura, chiusura, etc. poichè sono eseguite semplicemente invocando le corrispondenti primitive del sistema EOS.

Procedendo a dettagliare, si focalizza l'attenzione su un processo alla volta (prima "aggiornamento schema di memoria (centrale)" e poi "accesso allo schema persistente") al fine di non creare diagrammi troppo complicati. La figura 4.8 presenta il primo dei due sottocomponenti.

Non si notano flussi di dati aggiuntivi; il flusso di dati in ingresso *def. tipo e nome*, è stato separato nei flussi *nome* e *def. tipo*. All'atto dell'immissione di un tipo sia da parte dell'utente che da parte della funzione di caricamento dello schema persistente in memoria, viene eseguito il controllo di unicità del nome; se il controllo dà esito positivo, la funzione *inserimento* invoca i metodi per immettere il nuovo tipo nello schema di memoria centrale. Successivamente si cercano regole pendenti da classificare grazie all'immissione del nuovo tipo (in seguito verrà presentata dettagliatamente questa situazione).

In figura 4.9 è espansa la funzione *accesso allo schema persistente* che contiene le chiamate alle primitive di EOS per la gestione della base di dati.

I flussi di dati introdotti sono:

- *descriz. regole*: rappresenta l'insieme delle descrizioni delle regole memorizzate come pendenti, ma che ora possono essere classificate;
- *descriz. modificate*: contiene le nuove descrizioni delle regole classificate.

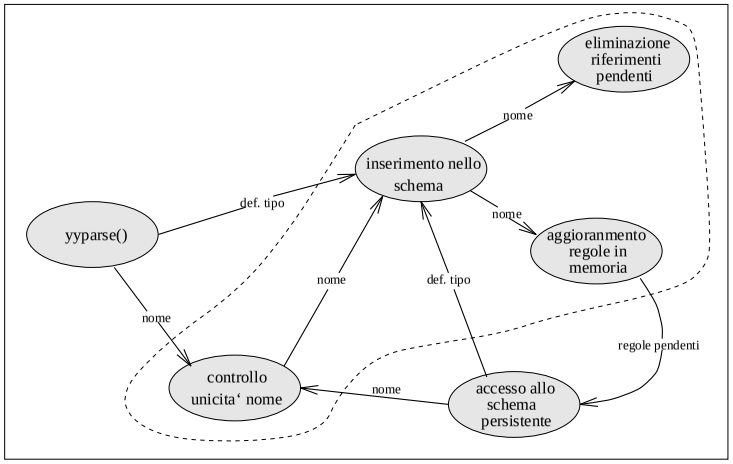

Figura 4.8: Aggiornamento dello schema in memoria centrale

Le funzioni "lettura" e "memorizzazione" eseguono la conversione delle strutture di memorizzazione delle descrizioni dei tipi da e per la memoria di massa; la tecnica di memorizzazione è spiegata in seguito. La funzione "aggiornamento regole in memoria" di fig.4.8 classifica, in memoria centrale, le regole pendenti associate all'ultimo tipo immesso dall'utente; se tra queste regole ne esistono di quelle che sono state rese persistenti in precedenza, è necessario che siano aggiornate anche le loro descrizioni su memoria di massa. Questo compito viene svolto dalla funzione "aggiornamento descrizioni regole pendenti.

4.3.3 Tipi e regole pendenti

Nella gestione di basi di dati occorre trattare descrizioni di tipi che fanno riferimento ad altri tipi non ancora definiti. Ad esempio, si può voler introdurre un tipo `X` la cui definizione è

```
Typedef Set<Y> X;
```


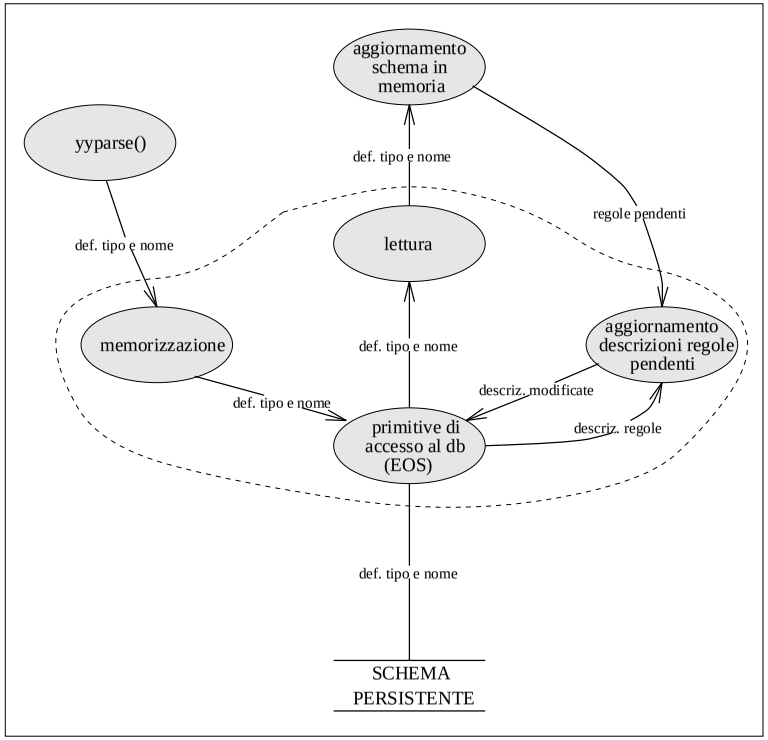

Figura 4.9: Aggiornamento dello schema persistente

cioè `X` è un insieme formato da elementi di tipo `Y`, senza avere ancora definito il tipo `Y`; il riferimento da `X` a `Y` è detto *pendente*. Quando verrà immessa la descrizione del tipo `Y`, il riferimento sarà risolto.

Analogamente, una regola si definisce *pendente* quando si applica a un tipo non ancora definito, cioè quando ha un riferimento pendente.

OCDL-PSSManager rende persistenti anche riferimenti e regole pendenti e usa particolari accorgimenti per la loro gestione che verranno illustrati nei paragrafi successivi.

4.3.4 Interazione tra OCDL-PSSManager e OCDL-Designer

Quando viene immesso il comando *compile* per la verifica di coerenza, se non esistono riferimenti pendenti all'interno dello schema, la routine di interfaccia `AVLtreetoLsigma` traduce le strutture di memoria di OCDL-PSSManager in quelle di OCDL-Designer; successivamente viene invocato il componente OCDL-Designer che calcola la forma canonica dello schema. Una volta che il controllo torna ad OCDL-PSSManager, viene esaminato il risultato ottenuto visualizzando gli eventuali tipi incoerenti.

La verifica di coerenza non ha senso se esistono riferimenti pendenti all'interno dello schema perchè non è possibile risalire alle descrizioni dei tipi mancanti; in tal caso si assume lo schema intrinsecamente incoerente ed OCDL-PSSManager comunica quali sono i riferimenti pendenti da eliminare prima che possa avere senso eseguire il controllo di coerenza.

4.3.5 Modifica dello schema della base di dati

La modifica intesa come aggiunta di nuovi tipi, non comporta particolari problemi. L'utente può immetterli direttamente da tastiera oppure farli leggere da un file di testo contenente tutte le descrizioni volute; il comando *read* serve a questo scopo. Invocandolo, il processo in esecuzione genera un processo figlio e si mette in stato di attesa; il processo figlio (che inizialmente è una copia del processo padre) esegue la redirezione dell'input dal file indicato dall'utente, legge tutte le definizioni, le rende persistenti e termina la sua esecuzione. A quel punto il processo originale si risveglia e leggendo dalla memoria di massa, aggiorna lo schema della base di dati; l'acquisizione dei tipi da file ha cosí termine.

N.B. La persistenza realizzata dal processo figlio può essere di due tipi:

- **temporanea**: l'utente è entrato nell'ambiente di editing disattivando il salvataggio automatico. La persistenza, allora, è solo un mezzo di comunicazione tra processi; quando il processo padre ha terminato l'aggiornamento delle strutture di memoria centrale, i tipi acquisiti dal processo figlio sono rimossi da memoria di massa. Per renderli persistenti occorre usare il comando *save*[3];

- **permanente**: è abilitato il salvataggio automatico. I tipi acquisiti dal processo figlio rimangono persistenti.

La variazione e la cancellazione di uno o più tipi sono situazioni più delicate. In questi casi è utile il comando *dump*: a partire dalle descrizioni di tipi presenti nelle strutture di memoria centrale, OCDL-PSSManager esegue il passaggio inverso rispetto a quello fatto dal comando *read*. Risale, cioè, a tutte le definizioni (rispettando la sintassi presentata al paragrafo B) servite per ottenere lo schema nella situazione corrente e le scrive, in modalità testo, sul file specificato. In questo modo l'utente può apportare le modifiche opportune alle definizioni usando un qualsiasi editor di testo; successivamente, dopo avere azzerato la base di dati con il comando *clear*, può dare in ingresso a OCDL-PSSManager le definizioni modificate ed ottenere un nuovo schema sfruttando il comando *read*. Se l'obiettivo è quello di eliminare le incoerenze, si può effettuare il controllo di consistenza e iterare i passi se necessario.

Questo metodo di modifica appare eccessivamente macchinoso se si pensa che una sola variazione, anche minimale, della descrizione di un tipo comporta la riacquisizione completa dell'intero schema della base di dati. Tuttavia, si deve tenere presente che la modifica di un tipo può portare al ridisegno completo dello schema; di conseguenza sarebbe necessario uno studio approfondito per risolvere il problema della propagazione delle modifiche. Siccome questo argomento esula dagli scopi della presente tesi, si lascia all'utente il compito di eseguire questa propagazione. In casi come questo l'utente può trarre notevoli vantaggi dall'utilizzo combinato dei comandi *dump* e *read*.

4.3.6 Persistenza dello schema

Durante la fase di creazione di una base di dati vengono creati, tra gli altri, anche i file di EOS chiamati *private_table* e *PEND_table* che OCDL-

[3] per la spiegazioni di questi ed altri concetti legati al funzionamento di OCDL-PSSManager si rimanda al capitolo 5

PSSManager usa per rendere persistente lo schema (comprensivo dei riferimenti pendenti). La persistenza può essere ottenuta *automaticamente* (ogni definizione corretta di tipo viene immediatamente resa persistente) oppure *manualmente* (il salvataggio dello schema avviene solo se l'utente invoca il comando *save*). Ad ogni immissione viene verificata l'unicità del nome di tipo, per rispettare la definizione di schema di base di dati (vedi paragrafo 3.1.1); eventuali duplicati di nomi di tipi vengono rifiutati.

Il file *private_table* contiene oggetti della classe `metaclass` la cui definizione completa è nel file *metaclass.hh*; qui si riportano solo la struttura dati nascosta e l'interfaccia della classe:

```
class metaclass {
  int        hidden_type;
  char       hidden_sigma[AVG_LEN + 1];
  eosoid     hidden_next;
 public:
  metaclass(int, char *); // costruttore

  int type();
  char *sigma();
  eosoid next();
  void set_type(int);
  void set_sigma(char *);
  void set_next(eosoid);
};
```

Come si può notare, si tratta di una struttura molto semplice che riduce al minimo l'occupazione di spazio su memoria di massa (ciascuno dei metodi che fa parte dell'interfaccia è costituito al più da due istruzioni).

Per rendere persistente una descrizione, la si trasforma in stringa seguendo una certa sintassi (la stessa che è utilizzata anche per visualizzare lo schema in risposta ai comandi *show*). Una tecnica di memorizzazione che fa uso di liste, rende variabile la lunghezza della stringa allo scopo di evitare sia sprechi di memoria persistente dovuti a sovradimensionamento, sia problemi di overflow della stringa dovuti a sottodimensionamento.

La stringa contenente la descrizione completa viene frazionata in moduli di dimensione pari ad `AVG_LEN` caratteri (solo l'ultimo modulo potrà avere lunghezza minore); ciascun modulo viene inserito in un'istanza di `metaclass` che a sua volta viene collegata opportunamente alle altre (si crea una lista persistente per ogni descrizione). In fase di apertura della base di dati, quando si acquisiscono le descrizioni, vengono di nuovo ricomposte le stringhe nella loro interezza proprio grazie ai collegamenti tra istanze.

Sarebbe necessario uno studio approfondito per determinare il valore ottimo di `AVG_LEN` affinchè non sia troppo grande (spreco di memoria dovuto a sovradimensionamento), ma nemmeno troppo piccolo (due svantaggi: 1) spreco di memoria dovuto allo sbilanciamento tra spazio destinato all'informazione utile e spazio destinato ai metodi e agli altri campi della classe; 2) dilatazione dei tempi in fase di apertura della base di dati a causa del maggior numero di oggetti da leggere, con conseguente aumento del numero di accessi alla memoria persistente). Empiricamente è stato assegnato ad `AVG_LEN` il valore 32, che è sembrato essere un valore soddisfacente rispetto ai criteri citati.

Il file *PEND_table* dei riferimenti pendenti, contiene oggetti della classe `PENDmetaclass`; l'informazione utile contenuta in ogni oggetto non è altro che un nome di tipo sul quale esiste almeno un riferimento pendente. Il file viene aggiornato tutte le volte che si chiude la base di dati e viene letto in fase di apertura (le funzioni utilizzate sono, rispettivamente, `save_pending` e `load_pending`).

La classe `OCDLmetaclass` sarà usata per memorizzare in modo persistente i tipi originati dall'esecuzione del modulo ODB-QOptimizer una volta che verrà integrato con OCDL-PSSManager; le primitive `getschemaN()`, `getschemabt()`, `getschemacalc()` e `putschema()` servono per l'interfacciamento e sono incluse con il codice di OCDL-PSSManager. Brevemente, le prime due funzioni permettono a ODB-QOptimizer di acquisire lo schema della base di dati che è stato immesso con OCDL-PSSManager, la terza memorizza persistentemente i tipi calcolati sul file di EOS *OCDL_table* (anch'esso creato assieme alla base di dati) e l'ultima li riacquisisce.

I risultati prodotti a livello fisico da OCDL-PSSManager si possono visualizzare utilizzando il programma *eosfsview* fornito con la distribuzione di EOS. Si veda il capitolo 5 per l'invocazione del comando.

4.3.7 Strutture dati

Poichè uno dei limiti del modello DFD è quello di non descrivere i dati usati nell'applicazione, si descrivono le strutture dati principali.

Per facilitare l'interfacciamento tra OCDL-PSSManager e OCDL-Designer, si sono scelte strutture analoghe per la rappresentazione di descrizioni di tipi; ovviamente, l'analogia riguarda le strutture dati, le quali, in OCDL-PSSManager, sono mantenute nascoste dai metodi di accesso agli oggetti.

Lo schema completo di una base di dati è mantenuto su 5 alberi binari bilanciati, istanze della classe `AVL_tree`. I nodi che formano gli alberi sono istanze della classe `node`; le definizioni complete delle due classi si possono trovare nel file *AVL_tree.hh* riportato in appendice con il resto del software; quelle che seguono sono solo le definizioni delle strutture dati:

```
class AVL_tree {
  int        type;
  node      *root;
  int        not_balanced;
 public:
   .
   .  // metodi
   .
};

class node {
  char      *key;
  void      *data;
  node      *left;
  node      *right;
 public:
   .
   .  // metodi
   .
};
```

Ciascun campo ha il seguente significato:

type: costante che indica la categoria dei tipi contenuti nell'albero;

root: puntatore alla radice dell'albero;

not_balanced: indica se l'albero è bilanciato o meno; serve per controllare la fase di inserimento di un nuovo nodo.

key: puntatore al nome del tipo;

data: puntatore alla testa della lista che contiene la descrizione del tipo;

left: puntatore al sottoalbero di sinistra del nodo (contenente tutti i tipi con nomi che precedono questo tipo, secondo l'ordine alfabetico);

right: puntatore al sottoalbero di destra del nodo (contenente tutti i tipi con nomi che seguono questo tipo, secondo l'ordine alfabetico).

Si è scelto di utilizzare un albero binario bilanciato per potere effettuare ricerche rapide tra i nomi dello schema anche nel caso di basi di dati molto complesse. Ognuno dei 5 alberi contiene i tipi di una delle categorie sottoelencate:

- tipi base;
- tipi valore;
- tipi classe primitiva;
- tipi classe virtuale;
- regole pendenti;

Le prime 4 categorie riguardano tipi di dati già trattati nel capitolo 3 e più dettagliatamente in [?], mentre l'ultima categoria si è resa necessaria per gestire le regole pendenti. Questa categoria di tipi costituisce una sorta di "anticamera"; infatti una regola deve essere considerata come tipo valore o tipo classe virtuale a seconda che il tipo a cui si riferisce sia, rispettivamente, un tipo valore o un tipo classe (sia primitiva che virtuale). Tuttavia, se la definizione di una regola precede quella del tipo, non è possibile stabilire la sua categoria; allora la si lascia in attesa sull'albero delle regole. Nel momento in cui si inserisce un tipo valore o una classe, si classificano opportunamente anche le sue eventuali regole pendenti.

Ogni nodo di un albero corrisponde a un tipo dello schema; nella parte contenente l'informazione, si trova il puntatore alla testa della lista di oggetti che rappresenta la sua descrizione. Nel caso di tipi base, gli oggetti

della lista sono istanze della classe `Ebt`, mentre per tutte le altre categorie di tipi, gli oggetti che compongono la descrizione sono istanze della classe `El_sigma`. Entrambe le classi di oggetti hanno la struttura dati che coincide, rispettivamente, con i tipi `bt` e `L_sigma` di OCDL-Designer. Questa scelta è motivata dal fatto che tali strutture di memoria sono in grado di trattare efficientemente qualsiasi descrizione di tipo, comunque complessa; inoltre, poichè OCDL-PSSManager va interfacciato con OCDL-Designer, la traduzione delle strutture di memoria risulta facilitata.

Le definizioni complete delle classi `Ebt` ed `El_sigma` si trovano rispettivamente nei file *Ebt.hh* ed *El_sigma.hh*, ma è significativo spiegare le loro strutture dati.

La classe `Ebt` contiene le descrizioni dei tipi base.

```
class Ebt {
  int       hidden_type;
  int       hidden_iMin;
  int       hidden_iMax;
  float     hidden_rvalue;
  char     *hidden_hvalue;
 public:
   .
   .  // metodi
   .
};
```

dove i vari campi hanno i seguenti significati:

hidden_type: costante che identifica il tipo; può assumere i seguenti valori:

- `INT` = tipo intero;
- `STRING` = tipo stringa;
- `REAL` = tipo reale;
- `RANGE` = tipo range;
- `BOOL` = tipo booleano;
- `VINT` = tipo valore intero;
- `VSTRING` = tipo valore stringa;

- `VREAL` = tipo valore reale;
- `VBOOL` = tipo valore booleano;

hidden_iMin: campo utilizzato dai tipi `VINT`, `RANGE` e `VBOOL`: nel primo caso contiene il valore intero, nel secondo caso il valore inferiore del range, infine nel terzo assume il valore 1 (TRUE) o 0 (FALSE);

hidden_iMax: campo utilizzato dal tipo `RANGE`; in tal caso contiene il valore superiore del range;

hidden_rvalue: campo utilizzato dal tipo `VREAL`; in tal caso contiene il valore reale;

hidden_hvalue: campo utilizzato dal tipo `VSTRING`; in tal caso contiene il puntatore al valore stringa.

Le descrizioni delle classi e dei tipi valore vengono rappresentate come liste di oggetti della classe `El_sigma`.

```
class El_sigma {
  char        *hidden_name;
  int          hidden_type;
  void        *hidden_field;
  El_sigma    *hidden_next;
  El_sigma    *hidden_and;
 public:
   .
   .  // metodi
   .
};
```

dove i vari campi hanno i seguenti significati:

hidden_name: campo utilizzato per i nomi di tipi o di classi;

hidden_type: costante che identifica il tipo può assumere i seguenti valori:

- `B` = tipo base
- `T` = nome tipo valore

- `C` = nome classe primitiva
- `D` = nome classe virtuale
- `PENDING` = nome di un tipo non ancora definito nello schema
- `RA_V` = antecedente di una regola di tipo classe virtuale
- `RC_V` = conseguente di una regola di tipo classe virtuale
- `RA_T` = antecedente di una regola di tipo valore
- `RC_T` = conseguente di una regola di tipo valore
- `RA_P` = antecedente di una regola pendente
- `RC_P` = conseguente di una regola pendente
- `CB` = nome classe fittizia
- `SET` = insieme
- `ESET` = tipo insieme con quantificatore esistenziale
- `SEQ` = sequenza
- `EN` = ennupla
- `ITEM` = attributo di un'ennupla
- `DELTA` = tipo oggetto
- `NB` = nome tipo base
- `TMP_CAT` = elemento di collegamento temporaneo tra la descrizione di un tipo e la descrizione di una sua regola;

hidden_field: puntatore a elementi di strutture contenenti informazioni aggiuntive; attualmente viene utilizzato solo per i tipi base e quindi si tratta di un puntatore a un oggetto di tipo `Ebt`;

hidden_next: puntatore ad un elemento successivo della lista utilizzato per la rappresentazione di descrizioni che hanno una struttura annidata;

hidden_and: puntatore ad un elemento della lista stessa utilizzato per rappresentare le intersezioni nelle descrizioni.

Per ogni tipo si illustrano le convenzioni adottate per la rappresentazione

TIPO BASE

HIDDEN_NAME	
HIDDEN_TYPE	B
HIDDEN_FIELD	$\longrightarrow$ Ebt
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME		nome non definito
HIDDEN_TYPE	=	costante che identifica un tipo base
HIDDEN_FIELD	=	puntatore ad un elemento di tipo `bt`
HIDDEN_AND		non rilevante
HIDDEN_NEXT		il livello di annidamento ha termine

NOME TIPO BASE

HIDDEN_NAME	stringa
HIDDEN_TYPE	NB
HIDDEN_FIELD	$\longrightarrow$ Ebt
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME	=	nome del tipo base
HIDDEN_TYPE	=	costante che identifica un nome tipo base
HIDDEN_FIELD	=	puntatore ad un elemento di tipo `bt`
HIDDEN_AND		non rilevante
HIDDEN_NEXT		il livello di annidamento ha termine

NOME TIPO VALORE

HIDDEN_NAME	stringa
HIDDEN_TYPE	T
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME = nome del tipo valore
HIDDEN_TYPE = costante che identifica un nome tipo valore
HIDDEN_FIELD campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT il livello di annidamento ha termine

NOME CLASSE VIRTUALE

HIDDEN_NAME	stringa
HIDDEN_TYPE	D
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME = nome della classe virtuale
HIDDEN_TYPE = costante che identifica un nome di classe virtuale
HIDDEN_FIELD campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT il livello di annidamento ha termine

NOME CLASSE PRIMITIVA

HIDDEN_NAME	stringa
HIDDEN_TYPE	C
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME = nome della classe base
HIDDEN_TYPE = costante che identifica un nome di classe base
HIDDEN_FIELD campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT il livello di annidamento ha termine

RIFERIMENTO PENDENTE

HIDDEN_NAME	stringa
HIDDEN_TYPE	PENDING
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME = nome del tipo mancante dallo schema
HIDDEN_TYPE = costante che identifica un riferimento pendente
HIDDEN_FIELD campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT il livello di annidamento ha termine

ATOMO FITTIZIO

HIDDEN_NAME	stringa
HIDDEN_TYPE	CB
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME = nome dell'atomo fittizio
HIDDEN_TYPE = costante che identifica un atomo fittizio
HIDDEN_FIELD campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT il livello di annidamento ha termine

NOME ANTECEDENTE DI REGOLA

HIDDEN_NAME	stringa
HIDDEN_TYPE	RA_T o RA_D o RA_P
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME = nome dell'antecedente di regola
HIDDEN_TYPE = costante che identifica un antecedente
HIDDEN_FIELD campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT il livello di annidamento ha termine

NOME CONSEGUENTE DI REGOLA

HIDDEN_NAME	stringa
HIDDEN_TYPE	RC_T o RC_D o RC_P
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	NULL

HIDDEN_NAME = nome del conseguente di regola
HIDDEN_TYPE = costante che identifica un conseguente
HIDDEN_FIELD campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT il livello di annidamento ha termine

TIPO OGGETTO $\triangle$S

HIDDEN_NAME	
HIDDEN_TYPE	DELTA
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	$\longrightarrow$ S

HIDDEN_NAME nome non definito
HIDDEN_TYPE = costante che identifica un tipo oggetto
HIDDEN_FIELD = campo non utilizzato
HIDDEN_AND non rilevante
HIDDEN_NEXT = puntatore all'elemento che descrive S

TIPO INSIEME {S}

HIDDEN_NAME	
HIDDEN_TYPE	SET o ESET
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	$\longrightarrow$ S

HIDDEN_NAME — nome non definito
HIDDEN_TYPE = costante che identifica un set o un set con esistenziale
HIDDEN_FIELD = campo non utilizzato
HIDDEN_AND — non rilevante
HIDDEN_NEXT = puntatore all'elemento che descrive S

TIPO SEQUENZA <S>

HIDDEN_NAME	
HIDDEN_TYPE	SEQ
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	$\longrightarrow$ S

HIDDEN_NAME — nome non definito
HIDDEN_TYPE = costante che identifica una sequenza
HIDDEN_FIELD = campo non utilizzato
HIDDEN_AND — non rilevante
HIDDEN_NEXT = puntatore all'elemento che descrive S

TIPO ENNUPLA [$a_1 : S_1$, ..., $a_p : S_p$]

HIDDEN_NAME	
HIDDEN_TYPE	EN
HIDDEN_FIELD	NULL
HIDDEN_AND	
HIDDEN_NEXT	$\longrightarrow a_1$

HIDDEN_NAME — nome non definito
HIDDEN_TYPE = costante che identifica un'ennupla
HIDDEN_FIELD — campo non utilizzato
HIDDEN_AND — non rilevante
HIDDEN_NEXT = puntatore all'elemento che descrive il primo attributo

TIPO ATTRIBUTO DI ENNUPLA a_1

HIDDEN_NAME	stringa
HIDDEN_TYPE	ITEM
HIDDEN_FIELD	NULL
HIDDEN_AND	$\longrightarrow a_2$
HIDDEN_NEXT	$\longrightarrow S_1$

HIDDEN_NAME = nome dell'attributo
HIDDEN_TYPE = costante che identifica un tipo attributo di ennupla
HIDDEN_FIELD — campo non utilizzato
HIDDEN_AND = puntatore attributo successivo
HIDDEN_NEXT = puntatore all'elemento che descrive il dominio dell'attributo

COLLEGAMENTO TEMPORANEO TIPO–REGOLA

HIDDEN_NAME	stringa
HIDDEN_TYPE	TMP_CAT
HIDDEN_FIELD	NULL
HIDDEN_AND	$\longrightarrow$ regola
HIDDEN_NEXT	

HIDDEN_NAME — campo non utilizzato
HIDDEN_TYPE = costante che identifica un collegamento temporaneo
HIDDEN_FIELD — campo non utilizzato
HIDDEN_AND = non rilevante
HIDDEN_NEXT = puntatore alla descrizione della regola collegata

ESEMPI DI CODIFICA

Data la seguente definizione di classe

$$\sigma_{\mathrm{P}}(\texttt{tmanager}) \;=\; \texttt{manager} \sqcap \triangle\Big[\texttt{level}: 8 \div 12\Big]$$

la sua dichiarazione, secondo la grammatica di OCDL-PSSManager, è

```
interface tmanager: manager {
  Tuple [Range [8->12]    level;]
};
```

il nodo dell'albero che contiene la classe avrà i seguenti valori:

KEY = tmanager
DATA = puntatore alla lista sotto definita
LEFT = puntatore al sottoalbero di sinistra
RIGHT = puntatore al sottoalbero di destra

La sua descrizione è una lista di oggetti di tipo `El_sigma` rappresentata in figura 4.10.

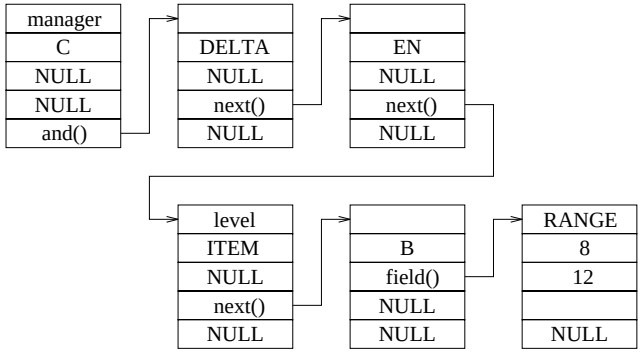

Figura 4.10: descrizione del tipo `tmanager`

N.B. Con i simboli `next()`, `and()` e `field()` si indicano i nomi dei metodi che permettono di accedere rispettivamente ai campi `hidden_next`, `hidden_and` e `hidden_field` della struttura dati nascosta. La parte rimanente riporta la rappresentazione delle sole strutture dati.

Data la seguente definizione di classe

$$\sigma_{\mathrm{P}}(\texttt{material}) \;=\; \triangle\Big[\texttt{name}: \texttt{string}, \texttt{risk} : \texttt{integer}, \texttt{feature} : \{\texttt{string}\}\Big]$$

la sua dichiarazione, secondo la grammatica di OCDL-PSSManager, è

```
interface material {
   Tuple[ String        name;
          Long          risk;
          Set<String>   feature;]
};
```

il nodo dell'albero che contiene la classe avrà i seguenti valori:

KEY = material
DATA = puntatore alla lista sotto definita
LEFT = puntatore al sottoalbero di sinistra
RIGHT = puntatore al sottoalbero di destra

La descrizione di `material` è una lista di oggetti di tipo `El_sigma` rappresentata in figura 4.11.

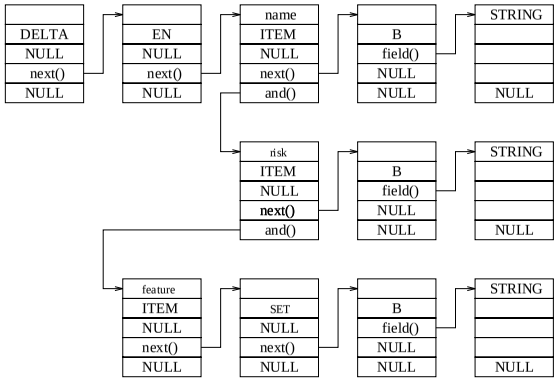

Figura 4.11: descrizione del tipo `material`

Si consideri la regola R1 dello schema del dominio Magazzino (figura 3.1); la sua definizione espressa secondo la sintassi di OCDL-PSSManager è:

```
Ruledef (manager) & (Range [5->10] ^.level) -> (Range
        [40000->60000] ^.salary) R1;
```

Il nodo dell'albero che contiene la regola avrà i seguenti valori:

`KEY`	=	R1
`DATA`	=	puntatore alla lista sotto definita
`LEFT`	=	puntatore al sottoalbero di sinistra
`RIGHT`	=	puntatore al sottoalbero di destra

La sua descrizione è una lista di oggetti di tipo El_sigma rappresentata in figura 4.12 (si noti che gli elementi terminali delle descrizioni di antecedente e conseguente sono di tipo Ebt e non El_sigma come accade per tutti gli altri).

Nella routine di interfaccia tra OCDL-PSSManager e OCDL-Designer le regole vengono scisse in due tipi distinti; con la convenzione adottata il

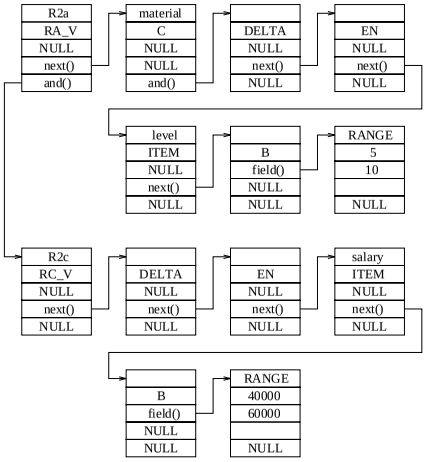

Figura 4.12: descrizione della regola `R1`

passaggio è pressochè immediato poichè si tratta di spezzare il collegamento tra antecedente e conseguente e di sostituire la testa di ciascuna lista con un elemento di tipo lN, mentre le parti rimanenti delle descrizioni rimangono invariate (vedi [?] per maggiori dettagli sulle strutture usate da OCDL-Designer).

Come ultimo esempio si illustra in figura 4.13 la situazione che si viene a creare nella descrizione di un tipo dopo l'esecuzione del comando *attach rules*. La descrizione della regola `R1` (già vista in figura 4.12 e che qui non si riporta) dello schema del dominio Magazzino risulta collegata a quella del tipo `manager`.

Se venisse definita un'altra regola sul tipo `manager`, verrebbe agganciata, tramite un altro elemento TMP_CAT, al campo and() del primo TMP_CAT e così via per successive regole.

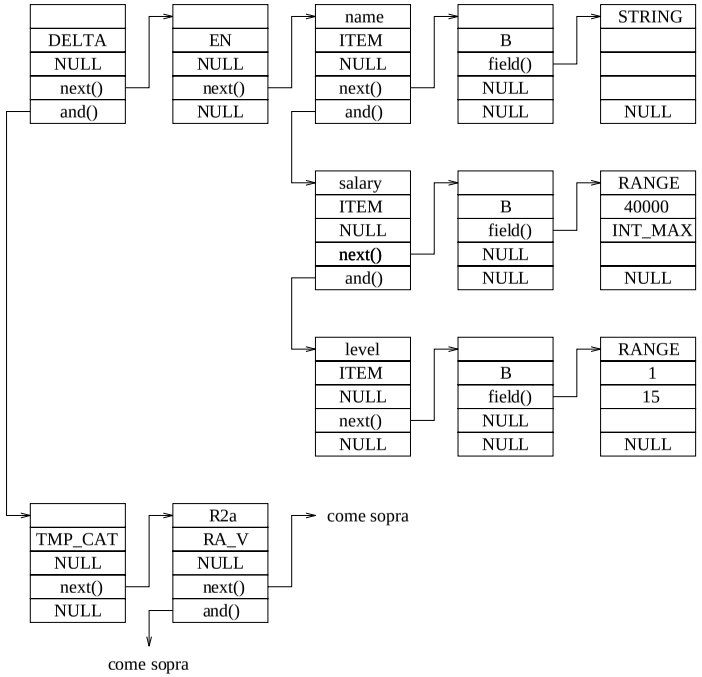

Figura 4.13: esempio di collegamento tipo-regola

Capitolo 5

Manuale utente

Per potere eseguire OCDL-PSSManager occorre preventivamente formattare un'area di EOS con il comando *eosareaformat*. Si genera così un file di Unix che costituisce l'area di memorizzazione. Per ottenere un'area *locale* di dimensioni contenute, il comando da usare dal prompt di Unix è il seguente:

```
eosareaformat nome_assoluto_area_locale -l -s 8 -n 64 -d 2 -o
```

Il significato delle varie opzioni è spiegato in [?]; ci si limita a sottolineare che l'area così creata ha le seguenti caratteristiche principali:

- è di tipo locale;
- il file di Unix che la contiene ha una dimensione di circa 4MB;
- può contenere al massimo 2 basi di dati.

Un'area *locale/privata* è utilizzata da un singolo utente, perciò su di essa si può escludere il server di EOS; ciò significa disabilitare tutti i componenti per il controllo di concorrenza, il logging e il ripristino, ottenendo un consistente miglioramento delle prestazioni in termini di velocità di esecuzione (per ulteriori dettagli si rimanda al paragrafo 2.1.1 e a [?]). OCDL-PSSManager può operare su qualsiasi area privata di EOS; in più, è stata predisposta un'area di default il cui nome assoluto è

```
/home/fontana/eos_area
```

Se si vuole permettere ad OCDL-PSSManager di operare su un'area *condivisa*, cioè accessibile a più utenti contemporaneamente, occorre modificare il

file *makefile* sostituendo, nel comando che esegue il link tra i vari file oggetto, la libreria

```
/home/fontana/eos2.2.0/lib/private.o
```

con la libreria

```
/home/fontana/eos2.2.0/lib/client.o
```

che attiva il server di EOS.
Lanciando il comando

```
eosfsview nome_assoluto_area_eos
```

dal prompt di Unix, si entra in un ambiente interattivo (*EOS File System Viewer*) che permette la visualizzazione di informazioni riguardo all'area di memorizzazione di EOS specificata e alle basi di dati, file e oggetti che essa contiene. L'ambiente presenta un'interfaccia primitiva, tuttavia l'utilizzo è talmente elementare che se ne trascura la descrizione dettagliata.

5.1 Utilizzo di OCDL-PSSManager

OCDL-PSSManager consiste di un doppio ambiente interattivo:

- **ambiente esterno** per la gestione di basi di dati considerate nella loro interezza: permette di creare, aprire, chiudere, distruggere e azzerare basi di dati, visualizzarne lo schema , verificarne la consistenza; non è ammessa nessuna modifica dello schema se non la sua completa rimozione (si può aprire solo una base di dati alla volta);
- **ambiente di editing** per la modifica dello schema di una base di dati.

Dopo aver digitato il comando

```
pssm [nome_assoluto_area_eos]
```

dal prompt di Unix, appare la seguente schermata:

```
   OCDL-PSSManager   Object Oriented Database Schema Interpreter
-------------------------------------------------------------------

pssm>
```

L'interprete è pronto ad accettare i comandi dell'ambiente esterno (contraddistinto dal prompt "`pssm>`"). Il parametro sulla linea di comando, che specifica l'area di EOS da considerare, è opzionale; in sua mancanza, OCDL-PSSManager utilizza l'area di default.

Nei prossimi paragrafi si descrivono i vari comandi in maniera dettagliata.

5.2 Comandi di validità generale

Esistono alcuni comandi che possono essere eseguiti indifferentemente dai due ambienti; ad eccezione del comando *help*, danno lo stesso risultato.

help - aiuto in linea

Elenca tutti i comandi, con relative brevi descrizioni, che si possono eseguire nell'ambiente da cui viene invocato.

Sintassi

help[;]

clear - azzeramento

È il comando che permette l'azzeramento di una base di dati; ciò significa che vuota la base di dati aperta, se esiste, altrimenti non ha effetto.

Sintassi

clear[;]

Esempio

Si supponga che il nome della base di dati attualmente aperta sia *Magazzino* e la si voglia azzerare; se l'esecuzione del comando avviene correttamente, il risultato a video è il seguente:


```
pssm> clear
Clearing database 'Magazzino'...  done
pssm>
```

N.B. Agisce sempre a livello persistente, anche nel caso che sia invocato dall'ambiente di editing impostato con il salvataggio non automatico.

compile - verifica di coerenza

Questo comando attiva il componente OCDL-Designer per verificare la consistenza dello schema della base di dati attualmente aperta.

Sintassi

compile[;]

Esempio

Se la verifica può essere eseguita, può dare due esiti:

- lo schema è coerente. Il risultato a video dell'operazione è il seguente:

```
pssm> compile
Compiling database schema...  done
Database schema is coherent
pssm>
```

- lo schema non è coerente. Esistono tipi incoerenti che vengono segnalati:

```
pssm> compile
Compiling database schema...  BOTTOM:intersezione tra tipi non
possibile
BOTTOM:intersezione errata tra due range
done
Database schema is NOT coherent



NON COHERENT TYPES:
-------------------
```

```
storage2


pssm>
```

Note, problemi e soluzioni
Se esistono riferimenti pendenti, vengono visualizzati, ma non viene eseguita nessuna verifica di coerenza. Ad esempio, se una descrizione contiene un riferimento pendente sul tipo TIPO_NON_DEFINITO:

```
pssm> compile
CANNOT COMPILE!


PENDING REFERENCES:
-------------------

'TIPO_NON_DEFINITO'


pssm>
```

comandi 'show' - visualizzazione schema

È un insieme di comandi che permette di visualizzare, sia parzialmente che integralmente, lo schema della base di dati aperta. Questi comandi richiedono tutti, come ultimo carattere della riga di invocazione, il punto e virgola.

show base

Comando che serve per visualizzare tutti i tipi base presenti nello schema.

Sintassi

show base;

Esempio

```
pssm> show base;
BASE TYPES
----------

B1 ->  boolean
F1 ->  floating point
FF -> −1200000000.000000
L1 ->  integer
S1 -> "bmvwb dnbamc Z<b ; cvanv"
S2 -> string
livello3 -> [−1..10]
livelloMin -> 1


pssm>
```

show value

Comando che serve per visualizzare tutti i tipi valore dello schema.

Sintassi

show value;

Esempio

```
pssm> show value;
VALUE TYPES
-----------

Set1 -> T1
Set2 -> E{Set1 & {string}}
Set3 -> E{livMin1 & livMin2}
T1 -> [a:  integer, b:  floating point, c:  string, d:  boolean]
T2 -> [a:  TRUE, b:  T1, c:  Set2, d:  Set1 & Set2]
T3 -> T2
livMin1 -> livelloMin
```

```
livMin2 -> livelloMin


pssm>
```

show class

Comando che serve per visualizzare tutti i tipi classe (primitiva e virtuale) definiti nello schema.

Sintassi

show class;

Esempio

```
pssm> show class;
PRIMITIVE CLASS TYPES
---------------------

smaterial -> material
sstorage -> storage
tmanager -> manager & ^[level:  [8..12]]
manager -> ^[name:  string, salary:  [40000..100000], level:  [1..15]]
material -> ^[name:  string, risk:  integer, feature:  {string}]


VIRTUAL CLASS TYPES
-------------------

storage2        ->  manager & ^[managed_by: manager, category: "B",
stock: {[item: material, qty: [10..300]]}, salary:
[-2147483648..39999]]

R1 -> (rule) manager & ^[level:  [5..10]] ==> ^[salary:  [40000..60000]]
R2 -> (rule) material & ^[risk:  [10..21]] ==> smaterial


pssm>
```


show rules

Comando che serve per visualizzare tutte le regole pendenti definite fino all'istante di invocazione del comando.

Sintassi

show rules;

Esempio

```
pssm> show rules;
PENDING RULES
-------------

R3 -> (rule) storage & ^[category:  "B4"] ==> ^[manager:  tmanager]
R4 -> (rule) storage & ^[stock:  [item:  smaterial]] ==> sstorage
R5 -> (rule) storage & ^[stock:  [qty:  [10..50]]] ==> ^[category:
"A2"]


pssm>
```

show schema

Visualizza l'intero schema suddividendo i tipi nelle 5 categorie di cui si è parlato nel paragrafo 4.3.

Sintassi

show schema;

Esempio

Digitando il comando

```
pssm> show schema;
```

viene prodotto in sequenza l'output di tutti i *comandi show* appena visti.

show types

Permette la visualizzazione delle descrizioni dei tipi elencati, se esistono.

Sintassi

show types [nome_tipo1[, nome_tipo2[, ... [, nome_tipoN]]]];

Esempio

```
pssm> show types tmanager, R4, S, material, TIPO_FALSO, Set3;



(prim class) tmanager -> manager & ^[level:  [8..12]]
(pending) R4 -> (rule) storage & ^[stock:  [item:  smaterial]]
==> sstorage
(base type) S -> "bmvwb dnbamc Z<b ; cvanv"
(virt class) material -> ^[name:  string, risk:  integer, feature:
string]
type 'TIPO_FALSO' is NOT defined
(value type) Set3 -> E{livMin1 & livMin2}



pssm>
```

which - nome della base di dati

Comunica il nome della base di dati aperta.

Sintassi

which[;]

Esempio

Se la base di dati aperta è Magazzino, l'esecuzione del comando dà il seguente output:

```
pssm> which
Currently open database is 'Magazzino'
pssm>
```

Se, invece, non è aperta nessuna base di dati:


```
pssm> which
NO OPEN DATABASE! type 'open db_name'
pssm>
```

attach rules - collegamento regole

Crea un collegamento temporaneo tra le regole (NON pendenti) e i tipi (valore o classe virtuale) ai quali esse si applicano. Successivamente l'utente può visualizzare l'effetto di questo comando utilizzando i comandi 'show'.

Sintassi

attach rules[;]

Esempio

Il comando dà il seguente output:

```
pssm> attach rules
Attaching rules to types...  done
pssm>
```

Note, problemi e soluzioni

Ovviamente opera solo se esiste una base di dati aperta; in aggiunta, non ha nessun effetto persistente. Se lo schema viene aggiornato dopo aver dato questo comando, i collegamenti tra nuovi tipi e nuove regole non viene eseguito automaticamente. **Sol.** Scollegare le regole con il comando *detach* e poi ricollegarle.

detach rules - separazione regole

Elimina il collegamento creato in precedenza con il comando *attach*.

Sintassi

detach rules[;]

Esempio

```
pssm> detach rules
Separating rules from types...  done
pssm>
```

Note, problemi e soluzioni
Viene eseguito automaticamente chiudendo la base di dati.

dump - salvataggio definizioni

Permette di scaricare su file di testo tutte le definizioni, scritte con la stessa sintassi usata per l'immissione (vedi sezione B), che hanno portato lo schema della base di dati nella situazione corrente. È un comando particolarmente utile nel caso in cui si vogliano apportare modifiche allo schema. L'utente può, infatti, eseguire questo comando, operare sul file generato per modificare le definizioni mediante un qualunque editor di testo, azzerare la base di dati e poi acquisire nuovamente lo schema dal file modificato (vedi il comando *read*).

Sintassi

dump *nome_file*[**;**]

Esempio
(Invocazione del comando da ambiente di editing, come si nota dal prompt):

```
# dump appoggio
Dumping database schema definition on file 'appoggio'...  done
#
```

Note, problemi e soluzioni
Attenzione! Non è ammesso il carattere '.' nel nome del file, per non generare interpretazioni errate da parte della routine di scansione dell'input prodotta da lex.

Il file indicato, se già esistente, viene sovrascritto. Se il nome di file non è dato in forma assoluta, viene creato nel direttorio da dove è stato invocato OCDL-PSSManager.

quit - uscita

Chiude la sessione di lavoro con OCDL-PSSManager; i passi che esegue sono:

- se esiste una base di dati aperta, la chiude dando la possibilità di salvarne lo schema nel caso sia stato modificato;
- termina l'esecuzione del programma.

Sintassi

quit[**;**]

5.3 Comandi dell'ambiente esterno

Come già detto in precedenza, in questo ambiente le basi di dati si considerano nella loro interezza non essendo possibile modificarne lo schema.

create - creazione di base di dati

Permette la creazione e l'apertura di una base di dati avente il nome specificato.

Sintassi

create *nome_base_di_dati*[**;**]

Esempio

```
pssm> create Magazzino
Creating and opening database 'Magazzino'...  done
pssm>
```

Ora esiste la base di dati *Magazzino* ed è già aperta (accessibile); la fase di creazione comporta anche la creazione di tutti i file di EOS utilizzati da OCDL-PSSManager.

Note, problemi e soluzioni
Il comando non ha effetto se è verificata almeno una delle condizioni seguenti:

- esiste una base di dati aperta. **Sol.** Occorre prima chiuderla usando il comando *close*;
- si tenta di creare una base di dati con lo stesso nome di una già esistente. Le possibili soluzioni sono:
  - cambiare il nome della base di dati che si vuole creare;
  - cambiare l'area di memorizzazione su cui agisce OCDL-PSSManager;
  - cancellare la base di dati esistente tramite il comando *destroy*;

- è stato raggiunto il numero massimo di basi di dati che possono essere contenute nell'area di memorizzazione su cui sta operando OCDL-PSSManager. Soluzioni:
  - cambiare area di memorizzazione;
  - rimuovere una base di dati non più necessaria;
  - riformattare l'area di memorizzazione per aumentare il numero massimo di basi di dati che può contenere. Per recuperare gli schemi delle basi di dati esistenti, utilizzare opportunamente i comandi *dump* e *read*.

open - apertura di base di dati

Il comando *open* permette di accedere allo schema di una base di dati esistente.

Sintassi

open *nome_base_di_dati*[;]

Esempio

```
pssm> open Magazzino
Opening database 'Magazzino'...  done
pssm>
```

Note, problemi e soluzioni
L'effetto di questa operazione è il trasferimento dello schema da memoria di massa a memoria centrale; non va a buon fine nelle seguenti situazioni:

- esiste già una base di dati aperta. **Sol.** Occorre prima chiuderla usando il comando *close*;
- non esiste una base di dati con nome uguale a quello specificato. **Sol.** Verificare la correttezza del nome ed eventualmente creare la base di dati con il comando *create*.

close - chiusura di base di dati

Serve per chiudere la base di dati aperta; per chiuderla occorre specificare anche il nome. Se lo schema della base di dati risulta modificato dopo l'ultimo salvataggio, viene data la possibilità di rendere persistenti anche gli ultimi cambiamenti.

Sintassi

close *nome_base_di_dati*[;]

Esempio
Chiusura senza salvataggio:

```
pssm> close Magazzino
Closing database 'Magazzino'...  done
pssm>
```

Chiusura con salvataggio:

```
pssm> close Magazzino
Database schema was modified since last saving.  Save (y/n)?  y
Saving database schema...  done
Closing database 'Magazzino'...  done
pssm>
```

Note, problemi e soluzioni
I principali motivi che non permettono un corretto funzionamento di questo comando e le possibili corrispondenti soluzioni sono:

- è aperta una sessione di editing. **Sol.** Chiuderla con il comando *exit*;
- la base di dati aperta non ha il nome specificato sulla linea di comando. **Sol.** Verificare il nome della base di dati tramite il comando *which*;

destroy - rimozione di base di dati

Rimuove l'intera base di dati avente il nome fornito sulla linea di comando; la base di dati su cui opera può essere sia chiusa che aperta (purchè il nome indicato coincida con il nome della base di dati aperta).

Sintassi

destroy *nome_base_di_dati*[;]

Esempio

```
pssm> destroy Magazzino
Destroying database 'Magazzino'...  done
pssm>
```

Note, problemi e soluzioni
Il comando è ininfluente se non esiste la base di dati col nome specificato oppure se la base di dati aperta non è quella che si vuole rimuovere.

edit - modifica dello schema

Con questo comando viene eseguita la transizione dall'ambiente esterno a quello di editing; prima che inizi la sessione di editing (che permette la modifica dello schema della base di dati aperta) è chiesta quale modalità di salvataggio deve essere utilizzata (salvataggio automatico o manuale). Dopo la risposta, il prompt cambia e inizia la vera e propria sessione di modifica dello schema.

Sintassi

edit[;]

Esempio

```
pssm> edit
OCDL-PSSManager  EDIT  ENVIRONEMENT
-----------------------------------


Autosaving (y/n)?  n
Autosaving not enabled


#
```

Note, problemi e soluzioni
Il comando non ha effetto se non è aperta nessuna base di dati.

5.4 Comandi dell'ambiente di editing

Per modificare lo schema di una base di dati occorre aprirla e poi entrare in questo ambiente; oltre ai comandi globali illustrati in precedenza, sono ammessi altri comandi.
N.B. Solo in questo ambiente è possibile immettere le definizioni di tipi secondo la sintassi introdotta nel paragrafo B e che qui non viene ripetuta. L'effetto dell'introduzione di ogni definizione è quello di aggiornare le strutture di memoria centrale (schema non persistente); se è abilitato il salvataggio automatico, viene anche aggiornato lo schema persistente.

Esempi di definizioni di tipi

Si riportano alcune definizioni per l'introduzione di tipi nello schema (ciò che segue i caratteri '\\' e fino alla fine della linea è interpretato come commento):

```
\\*** definizioni di tipi base e valore ***

Typedef Long L1, L2;

Typedef String S1, S2;

Typedef Float F1, F2;

Typedef Boolean B1;

Typedef F1 copiaF1;

const String S = "bmvwb dnbamc Z<b ; cvanv";

const Double FF = -.12E+10;

const Float GG = +19.001e-1;

const Float HH = 1.65701e3;

Typedef HH copiaHH;
```

```
Typedef Struct T1 {
  Long           a;
  Float          b;
  String         c;
  Boolean        d;
  Range [-5 ->]  e;
};

Typedef Set<T1> Set1;

Typedef E_set<(Set1) & Set<String>> Set2;

Typedef Struct T2 {
  a = TRUE;
  T1 b;
  Set2 c;
  (Set1 & Set2) d;
};

Typedef T2 T3;

Typedef Range [10->-1] livello1, livello2, livello3; \\ gli estremi
vengono scambiati

Typedef Range [1->1] livelloMin; \\ interpretato come valore intero

Typedef (livelloMin) livMin1, livMin2; \\ tipo valore come nome di
tipo base

Typedef E_set<(livMin1 & livMin2)> Set3;


\\*** definizioni di tipi classe (sia primitive che virtuali) ***

interface material {
  Tuple[ String name;
         Long risk;
         Set<String> feature;]
};
```



```
interface smaterial:  material {
};

interface manager {
  Tuple[ String name;
         Range [40000->] salary;
         Range [1->15] level;]
};

interface tmanager:  manager {
  Tuple[Range [8->12] level;]
};

interface storage {
  Tuple[ manager managed_by;
         category = "B4";
         Set<Struct {
             material item;
             Range [10->300] qty;}> stock;]
};

interface sstorage:  storage {
};

interface storage2:  manager (virtual) {
  Tuple[ manager managed_by;
         category = "B";
         Set<Struct {
             material item;
             Range [10->300] qty;}> stock;
         Range [->39999] salary;]
};


\\*** definizione di regole su tipi classe virtuale ***

Ruledef (manager) & (Range [5->10] ^.level) -> (Range [40000 -> 60000]
^.salary) R1;
```

```
Ruledef (material) & (Range [10->] ^.risk) -> smaterial R2;

Ruledef (storage) & (^.category = "B4") -> Tuple[tmanager manager;]
R3;

Ruledef (storage) & (smaterial ^.stock.Set.item) -> sstorage R4;

Ruledef (storage) & (Range [10 -> 50] ^.stock.Set.qty) -> (^.category
= "A2") R5;


\\*** definizione di tipi cammino ***

Typedef Long ^.stock.Set.item PATH1, PATH2, PATH3;

Typedef PATH2 PATH4;
```

read - lettura di un file

Comando che permette di acquisire le definizioni di tipi (ma, in generale, qualunque comando dell'ambiente di editing) da un file di testo.

Sintassi

read *nome_file*[;]

Esempio

```
# read definizioni
Reading file 'definizioni'...  done
#
```

Note, problemi e soluzioni

Attenzione! Non è ammesso il carattere '.' nel nome del file, per non generare interpretazioni errate da parte della routine di scansione dell'input prodotta da lex.

Se il nome del file non è dato in modo assoluto, la ricerca riguarda il

direttorio da dove è stato invocato OCDL-PSSManager. Se il file non viene trovato, viene segnalato un messaggio di errore.

save - salvataggio dello schema

Rende persistenti le modifiche apportate allo schema dopo l'ultimo salvataggio. È un comando che rivela la sua utilità nelle sessioni di editing nelle quali non è attivato il salvataggio automatico.

Sintassi

save[;]

Esempio

```
# save
Saving database schema...  done
#
```

Note, problemi e soluzioni

Il salvataggio non viene eseguito se non è stata apportata nessuna modifica allo schema.

exit - terminazione modifica dello schema

Termina la sessione di editing effettuando la transizione verso l'ambiente esterno; se lo schema è stato modificato dopo l'ultimo salvataggio, permette di salvare i cambiamenti.

Sintassi

exit[;]

Esempio

```
# exit
Database schema was modified since last saving. Save (y/n)? y
Saving database schema... done
    OCDL-PSSManager   Object Oriented Database Schema Interpreter
 ------------------------------------------------------------------------


pssm>
```

Appendice A

Sintassi originale di OCDL

Il linguaggio è descritto dalle seguenti regole grammaticali di carattere generale:

$$
\begin{array}{rcl}
< \text{linguaggio} > & ::= & < \text{def-term}_1 > \cdots < \text{def-term}_n > \\
< \text{def-term} > & ::= & < \text{def-tipovalore} > \mid \\
 & & < \text{def-classe} > \mid \\
 & & < \text{def-regola} > \mid \\
< \text{def-tipovalore} > & ::= & < \text{def-tipobase} > \mid \\
 & & < \text{def-tipo} > \\
< \text{def-classe} > & ::= & < \text{def-classe-prim} > \mid \\
 & & < \text{def-classe-virt} > \\
< \text{def-regola} > & ::= & < \textit{nome regola} >: \\
 & & < \text{def-antecedente} > \rightarrow \\
 & & < \text{def-conseguente} > \\
< \text{def-tipo} > & ::= & < \textit{nome tipovalore} > \\
 & & = < \text{tipo} > \\
< \text{def-classe-prim} > & ::= & < \textit{nome classe} > \leq < \text{classe} > \\
< \text{def-classe-virt} > & ::= & < \textit{nome classe} > = < \text{classe} > \\
< \text{def-antecedente} > & ::= & < \text{tipobase} > \mid \\
 & & < \text{tipo} > \mid \\
 & & < \text{classe} >
\end{array}
$$

$$
\begin{array}{rcl}
< \text{def-conseguente} > & ::= & < \text{tipobase} > \mid \\
 & & < \text{tipo} > \mid \\
 & & < \text{classe} > \\
< \text{tipo} > & ::= & < \top > \mid \\
 & & < \text{insiemi-di-tipi} > \mid \\
 & & < \text{esiste-insiemi-di-tipi} > \mid \\
 & & < \text{sequenze-di-tipi} > \mid \\
 & & < \text{ennuple} > \mid \\
 & & < \text{nomi-di-tipi} > \mid \\
 & & < \text{tipo-cammino} > \\
< \text{classe} > & ::= & < \text{insiemi-di-classi} > \mid \\
 & & < \text{esiste-insiemi-di-classi} > \mid \\
 & & < \text{sequenze-di-classi} > \mid \\
 & & < \text{nomi-di-classi} > \mid \\
 & & \triangle < \text{tipo} > \mid \\
 & & < \text{nomi-di-classi} > \sqcap \triangle < \text{tipo} > \\
< \text{insiemi-di-tipi} > & ::= & \forall \left\{ < \text{tipo} > \right\} \mid \\
 & & \forall \left\{ < \text{tipo} > \right\} \sqcap < \text{insiemi-di-tipi} > \mid \\
 & & \forall \left\{ < \text{tipobase} > \right\} \\
< \text{esiste-insiemi-di-tipi} > & ::= & \exists \left\{ < \text{tipo} > \right\} \mid \\
 & & \exists \left\{ < \text{tipo} > \right\} \sqcap < \text{esiste-insiemi-di-tipi} > \mid \\
 & & \exists \left\{ < \text{tipobase} > \right\} \\
< \text{sequenze-di-tipi} > & ::= & \langle < \text{tipo} > \rangle \mid \\
 & & \langle < \text{tipo} > \rangle \sqcap < \text{sequenze-di-tipi} > \mid \\
 & & \langle < \text{tipobase} > \rangle \\
< \text{ennuple} > & ::= & \left[< \text{attributi} > \right] \mid \\
 & & \left[< \text{attributi} > \right] \sqcap < \text{ennuple} > \\
< \text{attributi} > & ::= & < \textit{nome attributo} > : < \text{desc-att} > \mid
\end{array}
$$

$$
\begin{array}{rcl}
 & & < \textit{nome attributo} > : < \text{desc-att} > , < \text{attributi} > \\
< \text{desc-att} > & ::= & < \text{tipobase} > \mid \\
 & & < \text{tipo} > \mid \\
 & & < \text{classe} > \\
< \text{nomi-di-tipi} > & ::= & < \textit{nome tipovalore} > \mid \\
 & & < \textit{nome tipovalore} > \sqcap < \text{nomi-di-tipi} > \\
< \text{tipo-cammino} > & ::= & (< \text{cammino} > : < \text{desc-att} >) \\
< \text{cammino} > & ::= & \epsilon \mid \\
 & & < \text{el} > \mid \\
 & & < \text{el} > . < \text{cammino} > \\
< \text{el} > & ::= & < \textit{nome attributo} > \mid \\
 & & \exists \mid \\
 & & \forall \mid \\
 & & \triangle \mid \\
 & & \langle\rangle \mid \\
< \text{insiemi-di-classi} > & ::= & \forall \left\{ < \text{classe} > \right\} \mid \\
 & & \forall \left\{ < \text{classe} > \right\} \sqcap < \text{insiemi-di-classi} > \\
< \text{esiste-insiemi-di-classi} > & ::= & \exists \left\{ < \text{classe} > \right\} \mid \\
 & & \exists \left\{ < \text{classe} > \right\} \sqcap < \text{esiste-insiemi-di-classe} > \\
< \text{sequenze-di-classi} > & ::= & \langle < \text{classe} > \rangle \\
 & & \langle < \text{classe} > \rangle \sqcap < \text{sequenze-di-classi} > \\
< \text{nomi-di-classi} > & ::= & < \textit{nome classe} > \mid \\
 & & < \textit{nome classe} > \sqcap < \text{nomi-di-classi} > \\
< \text{def-tipobase} > & ::= & < \textit{nome tipobase} > \\
 & & = < \text{tipobase} > \\
< \text{tipobase} > & ::= & \textit{real} \mid \\
 & & \textit{integer} \mid \\
 & & \textit{string} \mid \\
 & & \textit{boolean} \mid
\end{array}
$$

$$
\begin{array}{rcl}
 & & < \text{range-intero} > \mid \\
 & & < \textit{valore reale} > \mid \\
 & & < \textit{valore intero} > \mid \\
 & & < \textit{valore string} > \mid \\
 & & < \textit{valore boolean} > \mid \\
 & & < \textit{nome tipobase} > \\
< \text{range-intero} > [1] & ::= & < \textit{valore intero} > - \textit{max} \mid \\
 & & \textit{min} - < \textit{valore intero} > \mid \\
 & & < \textit{valore intero} > - < \textit{valore intero} >
\end{array}
$$

[1] Quando si definisce un intervallo è possibile definire, nell'ordine, solo il limite inferiore, quello superiore o entrambi i limiti.

Appendice B

Grammatica standardizzata di OCDL

Il componente OCDL-PSSManager accetta definizioni di oggetti (i tipi dello schema) che rispettano, laddove ciò è possibile, la sintassi di ODL-93, mentre, per quanto riguarda i concetti peculiari di OCDL, si sono introdotte nuove regole grammaticali aventi struttura omogenea con quella delle regole standardizzate. La sintassi originale di OCDL presentata nel capitolo 3, è stata modificata nella seguente grammatica:

⟨definizioni⟩ ::= ⟨definizione⟩**;** |
⟨definizione⟩**;** ⟨definizioni⟩

⟨definizione⟩ ::= ⟨def-tiponon classe⟩ |
⟨def-tipoclasse⟩ |
⟨def-regola⟩

⟨def-tiponon classe⟩ ::= **Typedef** ⟨dichiaraz-tipo⟩ |
Const ⟨dichiaraz-costante⟩

⟨def-tipoclasse⟩ ::= **interface** ⟨dichiaraz-classe⟩

⟨def-regola⟩ ::= **Ruledef** ⟨dichiaraz-regola⟩

⟨dichiaraz-tipo⟩ ::= ⟨tipobase-sysdef⟩ ⟨IDs⟩ |
⟨tipo-userdef⟩ ⟨IDs⟩ |
⟨dichiar-tipovalore⟩

⟨tipo-userdef⟩ ::= ⟨ID⟩

⟨tipobase-sysdef⟩ ::= **Float** |
Long |
Boolean |
String |
Range [⟨descr-interv⟩]

⟨descr-interv⟩ ::= ⟨min⟩ **->** ⟨max⟩ |
::= ⟨min⟩ **->** |
::= **->** ⟨max⟩

⟨min⟩ ::= ⟨numero-intero⟩

⟨max⟩ ::= ⟨numero-intero⟩

⟨IDs⟩ ::= ⟨ID⟩ |
⟨ID⟩ **,** ⟨IDs⟩

⟨ID⟩ ::= ⟨lettera⟩ |
⟨lettera⟩ ⟨caratteri⟩

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

⟨caratteri⟩ ::= ⟨lettera⟩ | ⟨cifra⟩ | _ |
⟨lettera⟩ ⟨caratteri⟩ |
⟨cifra⟩ ⟨caratteri⟩ |
_ ⟨caratteri⟩

⟨dichiaraz-costante⟩ ::= **Float** ⟨ID⟩ = ⟨numero-reale⟩ |
Long ⟨ID⟩ = ⟨numero-intero⟩ |
Boolean ⟨ID⟩ = ⟨valore-booleano⟩ |
String ⟨ID⟩ = ⟨“⟩ ⟨*stringa*⟩ ⟨“⟩

⟨numero-intero⟩ ::= ⟨segno⟩ ⟨cifre⟩ |
⟨cifre⟩

⟨segno⟩ ::= **+** | **-**

⟨cifre⟩ ::= ⟨cifra⟩ |
⟨cifra⟩ ⟨cifre⟩

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

⟨numero-reale⟩ ::= ⟨segno⟩ ⟨reale⟩ |
⟨segno⟩ ⟨reale-esponenziale⟩ |
⟨reale⟩ |
⟨reale-esponenziale⟩

⟨reale⟩ ::= ⟨cifre⟩ **.** ⟨cifre⟩ |
. ⟨cifre⟩

⟨reale-esponenziale⟩ ::= ⟨reale⟩ ⟨letteraE⟩ ⟨numero-intero⟩ |
⟨reale⟩ ⟨letteraE⟩ ⟨segno⟩ ⟨numero-intero⟩

⟨letteraE⟩ ::= **e** | **E**
⟨valore-booleano⟩ ::= **TRUE** | **FALSE**
⟨dichiar-tipovalore⟩ ::= ⟨tipo-sequenza⟩ ⟨IDs⟩ |
⟨tipo-insieme⟩ ⟨IDs⟩ |
Struct ⟨ID⟩ **{** ⟨attributi⟩ **}** |
⟨descr-tipo⟩ ⟨cammino⟩ ⟨IDs⟩ |
(⟨tipo-userdef⟩ **)** ⟨IDs⟩ |
⟨2o+nomi⟩ ⟨IDs⟩ |
⟨1o+nomi⟩ **&** ⟨tipo-sequenza⟩ ⟨IDs⟩ |
⟨1o+nomi⟩ **&** ⟨tipo-insieme⟩ ⟨IDs⟩ |
⟨1o+nomi⟩ **&** **Struct** ⟨ID⟩ **{** ⟨attributi⟩ **}**
⟨2o+nomi⟩ ::= **(** ⟨ID⟩ **&** ⟨listanomi⟩ **)**
⟨1o+nomi⟩ ::= **(** ⟨tipo-userdef⟩ **)** |
⟨2o+nomi⟩
⟨listanomi⟩ ::= ⟨ID⟩ |
⟨ID⟩ **&** ⟨listanomi⟩
⟨attributi⟩ ::= ⟨dichiaraz-attributo⟩ **;** |
⟨dichiaraz-attributo⟩ **;** ⟨attributi⟩
⟨dichiaraz-attributo⟩ ::= ⟨descr-tipo⟩ ⟨ID⟩ |
⟨ID⟩ **=** ⟨val-costante⟩
⟨val-costante⟩ ::= ⟨numero-reale⟩ |
⟨numero-intero⟩ |
⟨valore-booleano⟩ |
⟨stringa⟩
⟨descr-tipo⟩ ::= ⟨dichiaraz-esplicita⟩ |
⟨dichiaraz-implicita⟩ |
⟨descr-cammino⟩
⟨dichiaraz-esplicita⟩ ::= **Sequence** < ⟨descr-tipo⟩ > |
Set < ⟨descr-tipo⟩ > |
E_set < ⟨descr-tipo⟩ > |
Struct **{** ⟨attributi⟩ **}**
⟨specif-corpo⟩
⟨dichiaraz-implicita⟩ ::= ⟨tipo-sysdef⟩ |
⟨tipo-userdef⟩ |
⟨2o+nomi⟩ |
⟨1o+nomi⟩ **&** ⟨dichiaraz-esplicita⟩
⟨tipo-sequenza⟩ ::= **Sequence** < ⟨descr-tipo⟩ >

⟨tipo-insieme⟩ ::= **Set** < ⟨descr-tipo⟩ > |
E_set < ⟨descr-tipo⟩ > |
⟨descr-cammino⟩ ::= **(** ⟨descr-tipo⟩ ⟨cammino⟩ **)** |
(⟨cammino⟩ **=** ⟨val-costante⟩ **)** |
< ⟨descr-tipo⟩ >
⟨cammino⟩ ::= ⟨simbolo⟩ **.** ⟨ID⟩ |
⟨simbolo⟩ **.** ⟨ID⟩ **.** ⟨cammino⟩
⟨simbolo⟩ ::= ^ |
Set |
E_set |
Sequence
⟨def-tipoclasse⟩ ::= ⟨ID⟩ **{** ⟨corpo⟩ **}** |
⟨ID⟩ **(virtual)** **{** ⟨corpo⟩ **}** |
⟨ID⟩ **:** ⟨listanomi⟩ ⟨specif-corpo⟩ |
⟨ID⟩ **:** ⟨listanomi⟩ **(virtual)** ⟨specif-corpo⟩
⟨specif-corpo⟩ ::= **{** ⟨corpo⟩ **}**
⟨corpo⟩ ::= ⟨classe-ennupla⟩ |
⟨classe-insieme⟩
⟨classe-ennupla⟩ ::= **Tuple** [⟨corpo-classeenn⟩]
⟨corpo-classeenn⟩ ::= ⟨export⟩ |
⟨export⟩ ⟨corpo-classeenn⟩
⟨classe-insieme⟩ ::= **Set** < ⟨tipo-insieme⟩ >
⟨export⟩ ::= **attribute** ⟨dichiaraz-attributo⟩ **;** |
⟨dichiaraz-attributo⟩ **;**
⟨dichiaraz-regola⟩ ::= ⟨antecedente⟩ **->** ⟨conseguente⟩ ⟨ID⟩
⟨antecedente⟩ ::= ⟨1o+nomi⟩ **&** ⟨descr-cammino⟩ |
⟨1o+nomi⟩ **&** **(** ⟨dichiaraz-esplicita⟩ **)** |
⟨1o+nomi⟩ **&** **(** ⟨dichiaraz-implicita⟩ **)**
⟨conseguente⟩ ::= ⟨descr-tipo⟩

Appendice C

Il sorgente di OCDL-PSSManager

Appendice D

Accesso allo schema persistente

Bibliografia

[A+89] M. Atkinson et al. The object-oriented database system manifesto. In *1nd Int. Conf. on Deductive and Object-Oriented Databases*. Springer-Verlag, 1989.

[AK89] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *SIGMOD*, pages 159–173. ACM Press, 1989.

[Atz93] P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects*. Springer-Verlag: LNCS n. 701, Heidelberg - Germany, 1993.

[BB93] J.P. Ballerini and S. Bergamaschi. Automatic building and validation of complex object database schemata. Technical Report 5/124, CNR - Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, July 1993.

[Bee90] C. Beeri. Formal models for object-oriented databases. In W. Kim, J.M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, page 405:430. Elsevier Science Publisher, B.V. - North-Holland, 1990.

[Ben94] D. Beneventano. *Uno strumento di inferenza nelle basi di dati ad oggetti: la sussunzione*. PhD thesis, Dip. di Elettronica, Informatica e Sistemistica, Università di Bologna, Bologna, 1994.

[BN91] S. Bergamaschi and B. Nebel. Theoretical foundations of complex object data models. Technical Report 5/91, CNR, Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Roma, January 1991.

[BN94] S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.

[Car84] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types - Lecture Notes in Computer Science N. 173*, pages 51–67. Springer-Verlag, 1984.

[CW91] N. Coburn and G.E. Weddel. Path constraints for graph-based data models: Towards a unified theory of typing constraints, equations and functional dependencies. In *2nd Int. Conf. on Deductive and Object-Oriented Databases*, pages 312–331, Heidelberg, Germany, December 1991. Springer-Verlag.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: the language and its implementation*. Addison∼Wesley, 1983.

[KFL89] W. Kim and editors F. Lochovsky. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, 1989.

[LR89] C. Lecluse and P. Richard. Modelling complex structures in object-oriented databases. In *Symp. on Principles of Database Systems*, pages 362–369, Philadelphia, PA, 1989.

[STR91] M.R. Scalas, P. Tiberio, and M. Rolli. Il tempo nelle basi di dati: aspetti architetturali. *Rivista di informatica*, 1991.