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

Progetto ed implementazione di schemi di basi di dati ODMG93 con vincoli di integritá in UNISQL/X

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Filippo Sgarbi

Correlatore
Dott. Ing. Maurizio Vincini

Anno Accademico 1997 - 98

Parole chiave:

Schemi di basi di dati ad oggetti
Vincoli di integritá
ODMG93
ODB-Tools
UNISQL/X

Ai miei genitori

RINGRAZIAMENTI

Nell'ambito accademico desidero per prima cosa ringraziare il mio relatore, **Prof.ssa Sonia Bergamaschi**, per la sua grande cortesia e disponibilita', nonché per la costante e indispensabile guida nella risoluzione dei vari problemi presentatisi; un grazie anche all'**Ing. Maurizio Vincini**, per aver attivamente partecipato al progetto, fornendomi interessanti spunti di approfondimento.
Sono inoltre riconoscente nei confronti di tutte le persone che mi sono state vicine e mi hanno aiutato in questi anni di studio, anche se un grazie del tutto speciale va **ai miei genitori**, per non avermi mai fatto mancare il loro appoggio ed avermi sostenuto in ogni circostanza, e alla mia ragazza **Francesca**, per essere riuscita a sopportarmi nonostante tutto.

Indice

1 INTRODUZIONE **1**
1.1 Contenuto della tesi . . . 2

2 L' AMBIENTE ODB-TOOLS PREESISTENTE **5**
2.1 Il prototipo ODB-Tools . . . 5
2.2 Il linguaggio OLCD . . . 6
2.2.1 Sintassi OLCD: analisi delle strutture principali . . . 6
Tipi e classi . . . 6
Regole di integrità . . . 7
Metodi . . . 8
2.2.2 Sintassi . . . 9
2.3 Il linguaggio ODL . . . 12
2.3.1 Schema di esempio . . . 12
2.3.2 Estensioni di ODL . . . 14
2.3.3 tipo base `range` . . . 14
2.3.4 viste o classi virtuali (`view`) . . . 15
2.3.5 vincoli di integritá (`rule`) . . . 15
Dot notation . . . 16
Costrutti delle `rule` . . . 17
- condizioni di appartenenza ad una classe . . . 17
- condizione sul tipo di un attributo . . . 17
- condizioni sul valore di un attributo . . . 17
- condizioni su collezioni ed estensioni . . . 18
Sintassi delle regole ODL-ODMG93 . . . 19
Osservazioni sulla grammatica . . . 20
2.3.6 Restrizioni di OLCD . . . 21
2.4 Esempio: database di gestione di un' universitá . . . 21
2.5 Architettura di ODB-Tools . . . 25
2.6 OCDL-Designer . . . 27
2.7 Programma principale . . . 27
2.8 Strutture Dati . . . 30

2.8.1 Descrizione listaN . . . 30
Strutture delle liste rappresentanti le descrizioni . . . 31
2.8.2 Descrizione listaB . . . 33
Struttura degli elementi tipi base . . . 33
2.8.3 Descrizione listaO . . . 34

3 UNISQL/X **35**
3.1 Panoramica di UNISQL/X . . . 35
3.2 UNISQL/X: generalitá . . . 36
3.2.1 Il sistema dei tipi di dato . . . 36
I tipi base . . . 36
Tipi addizionali . . . 37
C-Types . . . 38
3.2.2 Il linguaggio SQL/X . . . 38
3.2.3 I Metodi . . . 39
3.2.4 I Trigger . . . 40
3.3 Interagire con UNISQL/X . . . 43
3.4 Embedded SQL/X . . . 44
3.4.1 Le *Host Variable* . . . 45
3.4.2 I cursori . . . 46
3.4.3 ESQL/X communication area (SQLCA) . . . 47
3.4.4 Sommario degli statement ESQL/X . . . 48
3.5 L' interfaccia API . . . 50
3.5.1 Istruzioni per la gestione di database . . . 50
Funzioni per l' accesso ad un database . . . 50
Funzioni per la gestione delle transazioni . . . 51
Funzioni per la gestione degli errori . . . 51
3.5.2 Istruzioni per la gestione dei valori . . . 51
Funzioni per l' accesso a contenitori di valori . . . 51
Funzioni per la costruzione di contenitori di valori . . . 52
3.5.3 Istruzioni per la gestione dello schema . . . 52
Funzioni per la definizione di classi . . . 52
3.5.4 Istruzioni per la gestione degli oggetti . . . 53
3.5.5 Istruzioni per la gestione dei trigger . . . 54
3.5.6 Istruzioni per la gestione delle collezioni . . . 54

4 IMPLEMENTAZIONE DI SCHEMI DI DATI ODMG93 IN UNISQL/X **65**
4.1 Modulo "creazione database fisico" - struttura generale . . . 66
4.2 Descrizione delle funzioni . . . 70
4.2.1 Descrizione della funzione "Scrivi_intestazione" . . . 70

4.2.2 Descrizione funzione "Scrivi_schema_API" . . . 71
4.2.3 Descrizione funzione "Scrivi_coda" . . . 91
4.3 Implementazione di rule: un esempio . . . 92

A CONFRONTO TRA LE RELEASE 1.1 E 2.0 DI ODMG93 109
A.1 Object Model . . . 109
A.1.1 Tipi e classi: interfaccia e implementazione . . . 109
A.1.2 Oggetti . . . 111
A.1.3 Literals . . . 112
A.1.4 Metadati . . . 114
L' ODL Schema Repository . . . 114
Meta Object . . . 115
Specifier . . . 124
Operand . . . 125
Struttura dello Schema repository . . . 126
Esempio: creazione di uno schema . . . 128
A.1.5 Lock e controllo della concorrenza . . . 136
Tipi di lock . . . 137
Locking implicito e esplicito . . . 137
DURATA DEL LOCK . . . 137
A.1.6 Transazioni . . . 138
A.1.7 Operazioni sul tipo "Database" . . . 138
A.2 Object Description Language (ODL) . . . 139
A.2.1 Il linguaggio OIF . . . 141
Definizione di un oggetto . . . 141
Inizializzazione di attributi . . . 143
Definizione di relazioni . . . 146
Data migration . . . 146
Command line utilities . . . 147
A.3 Object Query Language . . . 148
A.3.1 Path expressions . . . 148
A.3.2 Valori nulli . . . 148
A.3.3 Invocazione di metodi . . . 149
A.3.4 Polimorfismo . . . 149
A.3.5 Compatibilita' tra tipi . . . 150
A.3.6 Definizione di query . . . 151
A.3.7 String expressions . . . 152
A.3.8 Comparazione di oggetti e literals . . . 153
A.3.9 Quantificatori esistenziali . . . 154
A.3.10 Operatore Order-by . . . 154
A.3.11 Operatore Group-by . . . 154

A.3.12 Dichiarazione di variabili nella clausola "from" . . . 156
A.3.13 Accesso ad un elemento di un dictionary attraverso la sua chiave . . . 156
A.3.14 Inclusione tra set o bag . . . 157
A.3.15 Rimozione di duplicati . . . 157
A.3.16 Abbreviazioni sintattiche . . . 157
A.4 Java binding . . . 159
A.4.1 Java object model . . . 159
A.4.2 Java ODL . . . 160
Dichiarazione di tipi e attributi . . . 160
Dichiarazione di relazioni . . . 161
Dichiarazione di operazioni . . . 161
A.4.3 Java OML . . . 161
Persistenza di oggetti e attributi . . . 162
Cancellazione di oggetti . . . 163
Modifica di oggetti . . . 163
Nomi di oggetti . . . 163
Locking di oggetti . . . 163
Proprieta' . . . 163
Operazioni . . . 163
Transazioni . . . 163
Operazioni sul database . . . 166
A.4.4 Java OQL . . . 167
Metodi "Collection query" . . . 167
La classe *OQLquery* . . . 168

B MODELLI DI INGEGNERIA DEL SOFTWARE 171
B.1 Il modello PHOS . . . 171

Elenco delle figure

2.1 Rappresentazione grafica dello schema Università 12
2.2 Schema dell' esempio di database di gestione di un' universitá 22
2.3 Componenti ODB-Tools 26
2.4 Architettura funzionale di OCDL-Designer 28
2.5 Struttura del programma OCDL-Designer (ante tesi) 29

4.1 Struttura attuale del programma OCDL-Designer 66
4.2 Struttura del modulo per la creazione del database fisico in UNISQL/X 70
4.3 Diagramma PHOS della funzione Scrivi_schema_API 72
4.4 Schema dell'esempio con indicazione esplicita delle rules 92

A.1 Struttura del Database Schema Repository 127
A.2 Schema del database di un' universitá 130

B.1 Esempio di chiamate in sequenza 171
B.2 Esempio di chiamate in alternativa 172
B.3 Esempio di iterazione 172
B.4 Esempio di ricorsione 172

Elenco delle tabelle

3.1 Il sistema dei tipi base in UNISQL/X 56
3.2 Descrizione del tipo DB_TYPE 57
3.3 Descrizione del tipo DB_TYPE_C 58
3.4 Corrispondenza tra "DB_TYPE" e "DB_TYPE_C" 59
3.5 Descrizione dei tipi C definiti in UNISQL/X 60
3.6 Corrispondenza tra tipi UNISQL/X e tipi delle *Host variable* . 61
3.7 Corrispondenze tra istruzioni "db_get" e tipi C 62
3.8 Corrispondenze tra istruzioni "db_make" e tipi UNISQL/X . . 63

4.1 Mapping dei tipi tra OLCD e UNISQL/X 67

A.1 Mapping dei tipi tra ODMG93 e Java 161

Capitolo 1

INTRODUZIONE

Le Basi di Dati Orientate ad Oggetti, OODB (Object Oriented Database), sono attualmente oggetto di intensi sforzi di ricerca e sviluppo. La motivazione principale per questo interesse é il fatto che il paradigma orientato ad oggetti offre un ampio insieme di strutture dati e di facilitazioni alla manipolazione che lo rendono adatto per il supporto sia delle tradizionali che delle nuove applicazioni.
L' Universitá di Modena ha al proposito iniziato a sviluppare negli ultimi anni un progetto, denominato ODB-Tools[1], il quale consente di definire uno schema di base di dati ad oggetti con vincoli di integritá e di effettuare all' interno di esso alcune importanti operazioni.
Il punto di partenza é rappresentato dalla descrizione dello schema scritta conformemente alla sintassi ODMG93, opportunamente estesa per poter rappresentare anche le regole di integritá. Tale descrizione viene data in ingresso ad un componente software chiamato OCDL-Designer, il quale:

1. Acquisisce lo schema.
2. Opera su di esso la trasformazione in forma canonica al fine di controllarne la consistenza.
3. Calcola le relazioni *isa* eventualmente implicite nelle descrizioni (sfruttando l' algoritmo di sussunzione).

Obiettivo di questa tesi é quello di estendere OCDL-Designer, arricchendolo di un' interfaccia di comunicazione con un OODBMS[2] commerciale. Ció dal punto di vista pratico significa fare in modo che venga fornito in output un ulteriore file, il quale, se opportunamente compilato, linkato ed eseguito,

[1] Si veda al proposito il sito web all' indirizzo *http://sparc20.dsi.unimo.it/*.
[2] OODBMS é l' acronimo di *Object Oriented DataBase Managment System*.

permetta di generare su piattaforma UNISQL/X lo schema correntemente analizzato.

1.1 CONTENUTO DELLA TESI

Capitolo 2: Viene brevemente analizzata l' architettura del prototipo ODB-Tools pre-esistente al presente lavoro di tesi, nonché il formalismo OLCD in esso utilizzato. Viene inoltre fornita una panoramica relativa all' estensione dell' ODL[3] di ODMG93, realizzata allo scopo di colmare le lacune espressive di quest' ultimo rispetto al linguaggio OLCD.

Capitolo 3: Viene offerta una panoramica di UNISQL/X, che rappresenta l' OODBMS commerciale scelto per implementare fisicamente gli schemi preventivamente filtrati da OCDL-Designer.

Capitolo 4: E' dedicato alla descrizione dell' estensione di ODB-Tools realizzata con il lavoro di questa tesi.
Tale descrizione si articola in due parti:

- Da un lato ho cercato di essere abbastanza preciso nella descrizione delle funzioni allo scopo di facilitare il piú possibile la comprensione delle motivazioni che hanno portato a certe scelte implementative piuttosto che ad altre.
  Siccome infatti penso che l' utilitá primaria di una descrizione di codice sia quella di guidare il lettore che voglia comprendere la struttura del codice stesso, una descrizione troppo superficiale risulta quasi inutile per chi legge, dal momento che non contiene un livello di dettaglio sufficiente da poter essere usata (anche solo in prima battuta) per accedere ed eventualmente manipolare o completare il codice. Questa considerazione assume poi a mio avviso ancora piú valore se si pensa che la mia tesi é parte di un progetto piú complesso (e perció soggetta probabilmente in futuro a modifiche ed integrazioni).
- Mi rendo tuttavia conto che la eccessiva ricchezza di dettagli potrebbe portare allo smarrimento del filo conduttore del lavoro svolto; per questo motivo ho inserito anche un paragrafo nel quale non si fa altro che dare una panoramica meno dettagliata, e perció forse preferibile per chi non sia interessato ai dettagli del codice, di quanto realizzato. Tale paragrafo in particolare é stato messo all'

[3] ODL é l' acronimo di *Object Definition Language*.

inizio del capitolo, in modo tale che anche il lettore che volesse studiare a fondo il programma possa propedeuticamente farsi un' idea di quello che in seguito andrá ad approfondire.

Appendice A: Viene presentato un confronto tra le release 1.1 e 2.0 dello standard ODMG93. Essendo infatti uscita all' inizio del 1998 la nuova versione del suddetto standard si é pensato opportuno, prima di procedere ad estendere ODB-Tools, verificare se fosse necessario o meno rivedere alcune sue parti alla luce di eventuali novitá sostanziali introdotte.
In particolare é stato fatto uno sforzo per enucleare nel modo piú preciso possibile tutte le novitá rilevate, in modo tale da fornire uno strumento utile per chi giá conosce lo standard nella vecchia versione e desideri aggiornarsi, senza doversi per questo sobbarcare l' onere dell' acquisto e della lettura dell' intero manuale ODMG93 release 2.0.

Appendice B: Viene offerta una breve descrizione dei modelli di ingegneria del software utilizzati nel presente volume.

Capitolo 2

L' AMBIENTE ODB-TOOLS PREESISTENTE

2.1 IL PROTOTIPO ODB-TOOLS

Il progetto ODB-Tools ha come obiettivo lo sviluppo di strumenti per la progettazione assistita di basi di dati ad oggetti e l'ottimizzazione semantica di interrogazioni. Gli algoritmi in esso operanti sono basati su tecniche di inferenza che sfruttano il calcolo della *sussunzione* e la nozione di *espansione semantica* di interrogazioni per la trasformazione delle query al fine di ottenere tempi di risposta inferiori. Il primo concetto è stato introdotto nell'area dell'Intelligenza Artificiale, più precisamente nell'ambito delle Logiche Descrittive, il secondo nell'ambito delle Basi di Dati. Questi concetti sono stati studiati e formalizzati in OLCD (*Object Language with Complements allowing Descriptive cycles*), una logica descrittiva per basi di dati che sará analizzata all' interno di questo stesso capitolo, piú precisamente nella sezione 2.2.

Come interfaccia verso l'utente esterno è stata scelta la proposta ODMG-93 [ODMG2.0], utilizzando il linguaggio ODL (*Object Definition Language*) per la definizione degli schemi ed il linguaggio OQL (*Object Query Language*) per la scrittura di query. Entrambi i linguaggi sono stati estesi per la piena compatibilità al formalismo OLCD.

Il capitolo è organizzato nel seguente modo: la sezione 2.2 analizza la struttura della logica descrittiva OLCD, mentre la sezione 2.3 presenta il formalismo ODL standard (tramite un esempio di riferimento), e le estensioni ad esso apportate.

Quindi il paragrafo 2.4 riporta un esempio riassuntivo di descrizione di uno schema sia in ODL esteso sia in OLCD. Nelle sezioni 2.5 e seguenti infine viene descritta l'architettura del prototipo cosí come si presentava prima dei miglioramenti apportati con il lavoro di questa tesi.

2.2 IL LINGUAGGIO OLCD

OLCD (*Object Language with Complements allowing Descriptive cycles*) é la logica descrittiva accettata ed utilizzata da ODB-Tools.

2.2.1 SINTASSI OLCD: ANALISI DELLE STRUTTURE PRINCIPALI

In OLCD la descrizione di uno schema consiste in una sequenza di definizioni di tipi, classi, regole e operazioni. Per distinguere i vari costrutti è stato introdotto un flag che specifica il tipo del termine che si sta definendo. Tale flag precede il nome e puó assumere valori diversi a seconda di quale genere di definizione introduce.

TIPI E CLASSI

Per introdurre la definizione di un tipo o di una classe posso usare uno dei seguenti flag:

`prim`: classe primitiva;

`virt`: classe virtuale;

`type`: tipo valore;

`btype` : tipo base;

La sintassi completa per la definizione di tipo o classe é invece:

$$< prefisso > < Nome_tipo > = < descrizione_del_tipo > \quad (2.1)$$

dove <*descrizione_del_tipo*> é determinata tramite la seguente grammatica (indico con S la descrizione di un tipo)):

$$
\begin{array}{lll}
S \rightarrow & B & \textit{tipo atomico} \\
 & N & \textit{nome tipo} \\
 & \{S\} & \textit{tipo insieme} \\
 & < S > & \textit{tipo sequenza} \\
 & \hat{\ } S & \textit{tipo oggetto} \\
 & [a_1 : S_1, ..., a_k : S_k] & \textit{tipo tupla} \\
 & S_1 \& S_2 & \textit{intersezione, indica ereditarieta'}
\end{array} \quad (2.2)
$$

REGOLE DI INTEGRITA'

Una regola di integrità viene denotata in OLCD tramite una coppia antecedente-conseguente, in cui sia l' antecedente sia il conseguente possono essere classi virtuali o tipi valore e perció vengono descritti attraverso la sintassi riportata in (2.1), fatto salvo per il diverso valore del flag, che nel caso di regole puó assumere i valori:

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

`antet`: antecedente di una regola di tipo valore.

`consv`: conseguente di una regola di tipo classe virtuale.

`const`: conseguente di una regola di tipo valore.

NOTE:

1. 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`.
2. Per mantenere la relazione tra antecedente e conseguente di una stessa regola occorre dare un particolare nome al tipo. Pertanto il nome della parte antecedente si ottiene dal nome della regola seguito da una a, quello della parte conseguente invece dal nome della regola seguito da una c.

   Es: Data la regola

   "Tutti i materiali con rischio maggiore di dieci sono materiali speciali SMATERIAL":

Essa in ODL ha la forma

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

In OLCD invece diventerá:

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

METODI

Un metodo (operazione) viene dichiarato in OLCD attraverso la seguente sintassi:

```
      < OpDcl >  ::=  < Operation > < ClassName > =
                      < ReturnType > f < Identifier > < ParameterDcls >
< ParameterDcls >  ::=  ( < ParamDclList > ) | ( )
 < ParamDclList >  ::=  < ParamDcl > |
                      < ParamDcl > , < ParamDclList >
     < ParamDcl >  ::=  [< ParamAttribute >] < Identifier > :
                      < SimpleTypeSpec >
< ParamAttribute > ::=  IN | OUT | INOUT
   < ReturnType >  ::=  < SimpleTypeSpec >
    < ClassName >  ::=  < Identifier >
```

In altri termini un metodo é univocamente identificato dalla parola chiave **operation**, e la sua definizione é composta da:

ClassName: Nome della classe a cui il metodo appartiene.

ReturnType: Tipo del valore di ritorno. Deve essere definito conformemente alla regola sintattica indicata in (2.2).

Identifier: Nome del metodo.

ParameterDclList: La lista dei parametri, ognuno dei quali a sua volta é composto da:

ParamAttribute: Indica se il parametro é di input (IN), di output (OUT) o di input-output (INOUT).

Identifier: Nome del parametro.

SimpleTypeSpec: Tipo del parametro. Deve essere definito conformemente alla regola sintattica indicata in (2.2).

2.2.2 SINTASSI

In questa sezione è riportata l'ultima versione disponibile della sintassi[1] OLCD riconosciuta da ODB-Tools.

```
< linguaggio >< linguaggio > ::= < def-term > |
                                 < linguaggio >< def-term >
               < def-term >  ::= < def-tipovalore > |
                                 < def-classe > |
                                 < def-regola >
         < def-tipovalore >  ::= < def-tipobase > |
                                 < def-tipo >
             < def-classe >  ::= < def-classe-prim > |
                                 < def-classe-virt >
             < def-regola >  ::= < def-antconV >
                                 < nome regola > = < classe > |
                                 < def-antconT >
                                 < nome regola > = < def-tipoT >
            < def-antconV >  ::= antev | consv
            < def-antconT >  ::= antet | const
              < def-tipoT >  ::= < tipo > | < tipobase >
               < def-tipo >  ::= type < nome tipovalore > = < tipo >
        < def-classe-prim >  ::= prim < nome classe > = < classe >
        < def-classe-virt >  ::= virt < nome classe > = < classe >
        < def-antecedente >  ::= < tipobase > |
                                 < tipo > |
                                 < classe >
        < def-conseguente >  ::= < tipobase > |
                                 < tipo > |
```

[1] il validatore è parte di un progetto in fase di sviluppo ed è normale che la sua sintassi venga modificata.


```
                                 < classe >
                   < tipo >  ::= #top# |
                                 < insiemi-di-tipi > |
                                 < esiste-insiemi-di-tipi > |
                                 < sequenze-di-tipi > |
                                 < ennuple > |
                                 < nomi-di-tipi > |
                                 < tipo-cammino >
                 < classe >  ::= < insiemi-di-classi > |
                                 < esiste-insiemi-di-classi > |
                                 < sequenze-di-classi > |
                                 < nomi-di-classi > |
                                 ˆ < tipo > |
                                 < nomi-di-classi > & ˆ < tipo >
        < insiemi-di-tipi >  ::= { < tipo > } |
                                 { < tipo > } & < insiemi-di-tipi > |
                                 { < tipobase > }
 < esiste-insiemi-di-tipi >  ::= !{ < tipo > } |
                                 !{ < tipo > } & < esiste-insiemi-di-tipi > |
                                 !{ < tipobase > }
       < sequenze-di-tipi >  ::= ⟨ < tipo > ⟩ |
                                 ⟨ < tipo > ⟩ & < sequenze-di-tipi > |
                                 ⟨ < tipobase > ⟩
                < ennuple >  ::= [ < attributi > ] |
                                 [ < attributi > ] & < ennuple >
              < attributi >  ::= < nome attributo > : < desc-att > |
                                 < nome attributo > : < desc-att > , < attributi >
               < desc-att >  ::= < tipobase > |
                                 < tipo > |
                                 < classe >
           < nomi-di-tipi >  ::= < nome tipovalore > |
                                 < nome tipovalore > & < nomi-di-tipi >
           < tipo-cammino >  ::= (< nome attributo > : < desc-att >)
      < insiemi-di-classi >  ::= { < classe > } |
```

```
                                  { < classe > } & < insiemi-di-classi >
< esiste-insiemi-di-classi >  ::= !{ < classe > } |
                                  !{ < classe > } & < esiste-insiemi-di-classe >
< sequenze-di-classi >        ::= ⟨ < classe > ⟩
                                  ⟨ < classe > ⟩ & < sequenze-di-classi >
< nomi-di-classi >            ::= < nome classe > |
                                  < nome classe > & < nomi-di-classi >
< def-tipobase >              ::= btype < nome tipobase > = < tipobase >
< tipobase >                  ::= real |
                                  integer |
                                  string |
                                  boolean |
                                  < range-intero > |
                                  vreal < valore reale > |
                                  vinteger < valore intero > |
                                  vstring < valore string > |
                                  vboolean < valore-boolean > |
                                  < nome tipobase >
< valore boolean >            ::= true | false
< range-intero >              ::= range < valore intero > +inf |
                                  −inf < valore intero > |
                                  < valore intero > < valore intero >
< OpDcl >                     ::= < Operation > < ClassName > =
                                  < ReturnType > f < Identifier >
                                  < ParameterDcls >
< ParameterDcls >             ::= ( < ParamDclList > ) | ( )
< ParamDclList >              ::= < ParamDcl > |
                                  < ParamDcl > , < ParamDclList >
< ParamDcl >                  ::= [< ParamAttribute >] < Identifier > :
                                  < SimpleTypeSpec >
< ParamAttribute >            ::= IN | OUT | INOUT
< ReturnType >                ::= < SimpleTypeSpec >
< ClassName >                 ::= < Identifier >
```


2.3 IL LINGUAGGIO ODL

Object Definition Language (ODL) è il linguaggio per la specifica dell'interfaccia di oggetti e di classi nell'ambito della proposta di standard ODMG-93. Il linguaggio svolge negli ODBMS le funzioni di definizione degli schemi che nei sistemi tradizionali sono assegnate al Data Definition Language. Le caratteristiche fondamentali di ODL, al pari di altri linguaggi basati sul paradigma ad oggetti, possono essere così riassunte:

- definizione di classi e tipi valore;
- distinzione tra intensione ed estensione di una classe di oggetti;
- definizione di attributi semplici e multivalore (set, list, bag);
- definizione di relazioni e relazioni inverse tra classi di oggetti;
- definizione della signature dei metodi.

La sintassi di ODL estende quella dell'Interface Definition Language, il linguaggio sviluppato nell'ambito del progetto Common Object Request Broker Architecture (CORBA) [CORBA].

2.3.1 SCHEMA DI ESEMPIO

Introduciamo ora uno schema che esemplifica la sintassi ODL utilizzata dal nostro prototipo.

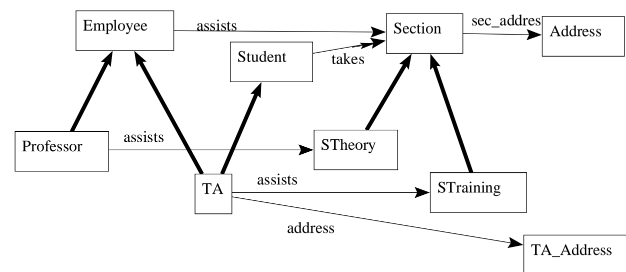

Figura 2.1: Rappresentazione grafica dello schema Università

L'esempio descrive la realtà universitaria rappresentata dai docenti, gli studenti, i moduli dei corsi e le relazioni che legano tra loro le varie classi.

In particolare esiste la classe dei moduli didattici (`Section`), distinti in moduli teorici (`STheory`) e di esercitazione (`STraining`). La popolazione universitaria è costituita da dipendenti (`Employee`) e studenti (`Student`). Un sottinsieme dei dipendenti è costituito dai professori (`Professor`) ed esiste la figura del docente assistente (`TA`) che è al tempo stesso sia un dipendente che uno studente dell'Università. I moduli vengono seguiti da studenti e sono tenuti da `TA`, o, solo per quelli teorici, da professori. In figura 2.1 viene rappresentato lo schema risultante, in cui le classi sono rappresentate da rettangoli, le relazioni di aggregazione tra classi tramite frecce (semplici per quelli monovalore, doppie per quelli multivalore), e le relazioni di ereditarietà sono rappresentate graficamente da frecce più marcate.

La definizione in ODL delle classi dello schema sará la seguente:

```
struct Address
{   string city;
    string street; };

interface Section ()
{   attribute string number;
    attribute  Address sec_address; };

interface STheory : Section()
{   attribute integer level; };

interface STraining : Section()
{   attribute string features; };

interface Employee  ()
{   attribute string name;
    attribute unsigned short annual_salary;
    attribute string domicile_city;
    attribute Section assists; };

interface Professor: Employee  ()
{   attribute string rank;
    attribute STheory assists; };

interface TA: Employee, Student ()
{   attribute STraining assists;
    attribute struct TA_Address { string city;
```



```
                                        string street;
                                        string tel_number; }
                                        address; };

interface Student ()
{    attribute string name;
     attribute integer student_id;
     attribute set<Section> takes; };
```

2.3.2 ESTENSIONI DI ODL

In questa sezione sono discusse le differenze tra ODL e il linguaggio OLCD, e conseguentemente le estensioni apportate all' ODL standard proprio in virtú di tali differenze.

Come visto nel paragrafo 2.2, OLCD prevede una ricca struttura per la definizione dei *tipi atomici*: sono presenti gli *integer*, *boolean*, *string*, *real*, *tipi mono-valore* e sottoinsiemi di tipi atomici, quali ad esempio intervalli di interi. A partire dai tipi atomici si possono definire *tipi valore*, attraverso gli usuali costruttori di tipo definiti nei CODM, quali *tuple*, *insiemi* e *tipi classe*, che denotano insiemi di oggetti con una identità ed un valore associato. Ai tipi può essere assegnato un nome, con la distinzione tra nomi per tipi-valore e nomi per tipi-classe (chiamati semplicemente *classi*). In tale assegnamento, il tipo può rappresentare un insieme di condizioni necessarie e sufficienti, o un insieme di condizioni solo necessarie. L'ereditarietà, sia semplice che multipla, è espressa direttamente nella descrizione di una classe tramite l'operatore di *congiunzione*.

Andiamo ora ad analizzare i concetti propri del formalismo OLCD che non trovano riscontro nello standard ODMG-93 e le relative estensioni da noi proposte al linguaggio ODL standard.

2.3.3 TIPO BASE `RANGE`

In OLCD è possibile esprimere intervalli di interi, utilizzati per l'ottimizzazione semantica di query in OQL con predicati di confronto. Per ovviare alla mancanza abbiamo introdotto nella sintassi ODL il costrutto `range`. Ad esempio, si può introdurre un modulo di teoria avanzato `ADVSTheory` come un modulo di teoria `STheory` il cui livello è opportunamente elevato (compreso tra 8 e 10):

```
interface ADVSTheory : STheory()
{       attribute  range {8, 10}  level;   };
```

2.3.4 VISTE O CLASSI VIRTUALI (VIEW)

OLCD introduce la distinzione tra *classe virtuale*, la cui descrizione rappresenta condizioni necessarie e sufficienti di appartenenza di un oggetto del dominio alla classe (corrispondente quindi alla nozione di vista) e *classe base (o primitiva)*, la cui descrizione rappresenta solo condizioni necessarie (corrispondente quindi alla classica nozione di classe). In altri termini, l'appartenenza di un oggetto all'interpretazione di una classe base deve essere stabilita esplicitamente, mentre l'interpretazione delle classi virtuali è calcolata sulla base della loro descrizione.

Le classi base sono introdotte tramite la parola chiave `interface`, mentre per le classi virtuali si introduce il costrutto `view` che specifica la classe come virtuale seguendo le stesse regole sintattiche della definizione di una classe base.. Ad esempio, la seguente dichiarazione:

```
view Assistant: Employee, Student ()
{ attribute Address address; };
```

introduce la classe virtuale `Assistant` che rappresenta tutti gli oggetti appartenenti sia alla classe studenti che dipendenti e che, in piú, hanno un indirizzo rappresentato dalla struttura `Address` definita dagli attributi via e città.

2.3.5 VINCOLI DI INTEGRITA' (RULE)

ODL-ODMG93 non consente di esprimere regole. Abbiamo quindi esteso ODL con regole "if ... then" che permettono di esprimere in maniera dichiarativa vincoli di integritá.

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

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

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


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

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

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

DOT NOTATION

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

Ad esempio, data la seguente dichiarazione:

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

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

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

Dato un oggetto X di tipo `Class3` si ha che:
`X.c3a1`: è di tipo `long`, fa riferimento direttamente all'attributo definito nella classe `Class3`.

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

COSTRUTTI DELLE `RULE`

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

- condizioni di appartenenza ad una classe

`identificatore_di_oggetto` **in** `nome_classe`

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

- condizione sul tipo di un attributo

`identificatore_di_attributo` **in** `nome_tipo`

Esempio:
`X.age in range {18, 25}`
ove X individua un oggetto con attributo `age` di tipo `range` o intero. La condizione è verificata se `X.age` ha un valore compreso tra 18 e 25 inclusi.
Nelle rule, i ranges sono compatibili solo con costanti di tipo intero.

- condizioni sul valore di un attributo

`identificatore_di_attributo` **operatore** `costante`

La costante può essere un letterale oppure una `const` purché dello stesso tipo dell'attributo.
Gli operatori di disuguaglianza possono essere usati solo con attributi di tipo intero, questo perché le disuguaglianze vengono tradotte condizioni sul tipo `range`.
Esempi:
`X.tass = "High"`
`X.age > 18`
`X.salary > lo_limit and X.salary < hi_limit`

NOTA BENE: La condizione puó anche essere data dal risultato dell' esecuzione di un' operazione definita nello schema.

- condizioni su collezioni ed estensioni

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

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

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

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

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

oppure

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

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

`exists iteratore in collezione: lista_di_condizioni`

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

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

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

SINTASSI DELLE REGOLE ODL-ODMG93

Di seguito si riporta l' estensione della sintassi ODL proposta al fine di poter esprimere vincoli di integritá all' interno di schemi di database.

```
< RuleDcl >          ::= rule < Identifier >
                         < RuleAntecedente > then < RuleConseguente >
< RuleAntecedente >  ::= < Forall > < Identifier > in < Identifier > :
                         < RuleBodylist >
< RuleConseguente >  ::= < RuleBodyList >
< RuleBodyList >     ::= ( < RuleBodyList > ) |
                         < RuleBody > |
```



```
                         < RuleBodylist > and < RuleBody > |
                         < RuleBodylist > and ( < RuleBodyList > )
< RuleBody >         ::= < DottedName > < RuleConstOp > < LiteralValue > |
                         < DottedName > < RuleConstOp > < RuleCast >
                         < LiteralValue > |
                         < DottedName > in < SimpleTypeSpec > |
                         < ForAll > < Identifier > in < DottedName > :
                         < RuleBodylist > |
                         exists < Identifier > in < DottedName > :
                         < RuleBodylist > |
                         < DottedName > =
                         < SimpleTypeSpec > < FunctionDef >
< RuleConstOp >      ::= = | >= | <= | < | >
< RuleCast >         ::= ( < SimpleTypeSpec > )
< DottedName >       ::= < Identifier > |
                         < Identifier > . < DottedName >
< ForAll >           ::= for all | forall
< FunctionDef >      ::= < Identifier > ( < DottedNameList > )
< DottedNameList >   ::= [< SimpleTypeSpec >] < DottedName > |
                         [< SimpleTypeSpec >] < LiteralValue > |
                         [< SimpleTypeSpec >] < DottedName > ,
                         < DottedNameList > |
                         [< SimpleTypeSpec >] < LiteralValue > ,
                         < DottedNameList >
```

Osservazioni sulla grammatica Osservando la sintassi sopra riportata e' possibile fare alcune considerazioni interessanti:

- L'attributo del parametro (ParamAttribute), premesso al tipo del parametro e' facoltativo. Nel caso non venga indicato viene considerato IN.
- Le operazioni dichiarate nel corpo delle rule possono appartenere solamente ad un sottoinsieme delle operazioni. Esse sono considerate delle funzioni matematiche. Questo significa che sono vincolate a rispettare le seguenti regole:

  – devono restituire un risultato
  – devono possedere almeno un parametro
  – ogni parametro deve avere attributo IN. Infatti non e' permesso indicare l'attributo del parametro nel caso si tratti di funzioni.

2.3.6 RESTRIZIONI DI OLCD

Essendo un linguaggio di definizione di schemi del tutto generale, ODL contiene alcuni costrutti aggiuntivi che non sono presenti nel formalismo OLCD.

- OLCD prevede soltanto la definizione di relazioni di aggregazione unidirezionali, quindi non è rappresentata l'operazione di inversione.
- Non sono supportati i tipi `enum` e `union`.
- In OLCD la parte intensionale ed estensionale di una classe sono referenziate dallo stesso nome.
- ODL permette una suddivisione gerarchica dello schema mediante l'uso dei *moduli*, mentre lo schema OLCD è piatto.
- Nell'attuale realizzazione del traduttore, le costanti possono essere solo dei *letterali*. Non possono essere ad esempio espressioni algebriche o valori strutturati costanti.

2.4 ESEMPIO: DATABASE DI GESTIONE DI UN' UNIVERSITA'

A conclusione di quanto detto nelle sezioni 2.3.2 e 2.3 riportiamo la descrizione in ODL esteso dello schema di un database di gestione di un' universitá, e a seguire la sua traduzione in OLCD, evidenziando poi le differenze riscontrate. La rappresentazione grafica dello schema é riportata in figura 2.2 a pagina 22.

Sintassi ODL esteso:

```
interface Course ( extent courses keys name, number)
   {
   attribute string name;
   attribute string number;
   relationship list<Section> has_sections
```


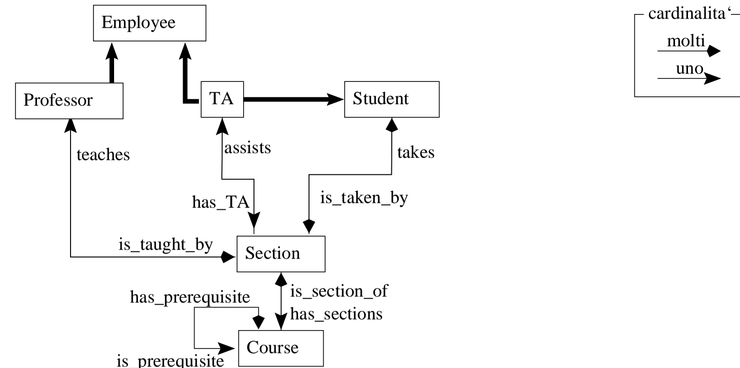

Figura 2.2: Schema dell'esempio

```
      inverse Section::is_section_of
      {order_by Section::number};
   relationship set<Course> has_prerequisites
      inverse Course::is_prerequisite_for;
   relationship set<Course> is_prerequisite_for
      inverse Course::has_prerequisites;
   boolean offer (in unsigned short semester) raises (already_offered);
   boolean drop (in unsigned short semester) raises (not_offered);
   };

interface Section ( extent sections key (is_section_of, number))
   {
   attribute string number;
   relationship Professor is_taught_by
      inverse Professor::teaches;
   relationship TA has_TA
      inverse TA::assists;
   relationship Course is_section_of
      inverse Course::has_sections;
   relationship set<Student> is_taken_by
      inverse Student::takes;
   };

interface Employee  ( extent employees key (name, id))
```

```
   {
   attribute string name;
   attribute short id;
   attribute unsigned short annual_salary;
   void fire () raises (no_such_employee);
   void hire ();
   };

interface Professor: Employee  ( extent professors)
   {
   attribute enum Rank { full, associate, assistant} rank;
   // attribute string rank;
   relationship set<Section> teaches inverse Section::is_taught_by;
   short grant_tenure () raises (ineligible_for_tenure);
   };

interface TA: Employee, Student()
   {
   relationship Section assists
      inverse Section::has_TA;
   };

interface Student ( extent students keys name, student_id)
   {
   attribute string name;
   attribute string student_id;
   attribute
      struct Address
         {
         string college;
         string room_number;
         } dorm_address;
   relationship set<Section> takes
      inverse Section::is_taken_by;
   boolean register_for_course
      (in unsigned short course, in unsigned short Section)
      raises
      (unsatisfied_prerequisites, section_full, course_full);
   void drop_course (in unsigned short Course)
      raises (not_registered_for_that_course);
   void assign_major (in unsigned short Department);
```



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

Sintassi OLCD:

```
prim Student =  ^ [ name : string ,
                    student_id : string ,
                    dorm_address : [ college : string ,
                                     room_number : string ],
                    takes : { Section }  ] ;

prim TA      =   Employee &
                 Student  &
                 ^ [  assists : Section  ] ;

prim Professor = Employee &
                 ^ [ rank : string ,
                     teaches : { Section }  ] ;

prim Employee =  ^ [ name : string ,
                     id : integer ,
                     annual_salary : integer ] ;

prim Section =   ^ [ number : string ,
                     is_taught_by : Professor  ,
                     has_TA : TA  ,
                     is_section_of : Course  ,
                     is_taken_by : { Student }  ] ;

prim Course =    ^ [ name : string ,
                     number : string ,
                     has_sections : { Section }  ,
                     has_prerequisites : { Course }  ,
                     is_prerequisite_for : { Course }  ] .

operation Student = boolean f register_for_course
```

```
                   ( in course : integer, in Section : integer ) ;
operation Student = void f drop_course ( in Course : integer ) ;
operation Student = void f assign_major ( in Department : integer ) ;
operation Student = integer f transfer ( in old_section : integer ,
                         in new_section : integer ) ;

operation Professor = integer f grant_tenure () ;

operation Employee = void f fire () ;
operation Employee = void f hire () ;

operation Course = boolean f offer ( in semester : integer ) ;
operation Course = boolean f drop ( in semester : integer ) ;
```

NOTE:

- OLCD permette la definizione di tipi valori direttamente nella dichiarazione delle classi, ad esempio l'attributo `dorm_address` nella dichiarazione della classe `Student`.
- *Differenze tra i due schemi*
  (1) L'attributo *Rank* di *Professor* in ODL esteso è di tipo *enum* ed è stato tradotto come *string* in quanto il tipo *enum* in OLCD non esiste.
  (2) Le relazioni *relationship* non sono rappresentate in OLCD. Esse appaiono come puntatori ad oggetti e perdono cosí parte del loro contenuto informativo. Non è infatti possibile esprimere la semantica di ruolo inverso ma solo esprimere ad esempio il ciclo tra `Professor` e `Section`.

2.5 ARCHITETTURA DI ODB-TOOLS

Scopo di questa sezione e' fornire una panoramica su ODB-Tools e i suoi principali componenti, al fine di chiarirne la struttura e permettere al lettore una maggiore comprensione dei miglioramenti apportati con il lavoro di questa tesi.

Come indicato in figura 2.3, ODB-Tools risulta composto da 3 moduli:

- **ODL_TRASL (il traduttore):**
  Dato uno schema di base di dati ad oggetti realizzato secondo le speci-

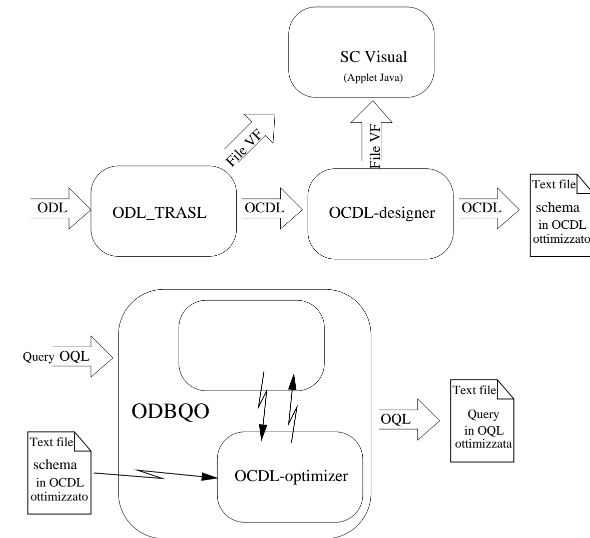

Figura 2.3: Componenti ODB-Tools

fiche ODMG93, l' ODL_TRASL si preoccupa di tradurlo sia conformemente alla sintassi OLCD, sia in formato `vf` (*Visual Form*), in modo che sia visualizzabile utilizzando l'applet Java `scvisual`.

- **OCDL-designer:**
  Ricevendo come input uno schema di base di dati ad oggetti scritto in OLCD, ne calcola la forma canonica, ne controlla la consistenza e ne determina la minimalita' rispetto alla relazione **isa**.

- **ODBQOptimizer (ODBQO):**
  E' adibito all'ottimizzazione semantica delle interrogazioni.

In particolare il lavoro di questa tesi ha coinvolto solo il modulo *OCDL-designer*; pertanto é nostro interesse premettere alla descrizione di quanto realizzato una panoramica circa lo stato dell' arte di solo questo tra i tre componenti dell' ODB-Tools, trascurando di entrare nel dettaglio circa il traduttore e l' ottimizzatore di interrogazioni.

2.6 OCDL-DESIGNER

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

- verificare che lo schema sia *consistente*, cioé che esista almeno uno stato del DB tale che ogni tipo, classe e regola abbia una estensione non vuota.
- ottenere *la minimalitá* dello schema rispetto alla relazione ISA; in altri termini per ogni tipo (classe) viene calcolata la giusta posizione all' interno della tassonomia dei tipi (classi), e cioé:
  - 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.

OCDL-Designer é stato realizzato in ambiente di programmazione C, versione standard ANSI C, su piattaforma hardware SUN SPARCstation 20, sistema operativo SOLARIS 2.3.
In figura 2.4 riportiamo l' architettura funzionale di OCDL-Designer.

Come si puó vedere il programma é diviso in due sottocomponenti funzionali principali che corrispondono a due fasi distinte:

- Il primo, denotato con OLCD-COHE, é quello che permette il controllo della coerenza dello schema, generandone la forma canonica.
- Il secondo, indicato con OLCD-SUBS, é quello che, partendo dallo schema OLCD canonico, calcola le relazioni di sussunzione e le relazioni ISA minimali che intercorrono tra i tipi (classi).

2.7 PROGRAMMA PRINCIPALE

Il programma acquisisce schemi di basi di dati ad oggetti complessi espressi nel linguaggio `OLCD`, opera la trasformazione in forma canonica al fine di controllare la consistenza dello schema e calcola le relazioni `isa` eventualmente implicite nelle descrizioni.

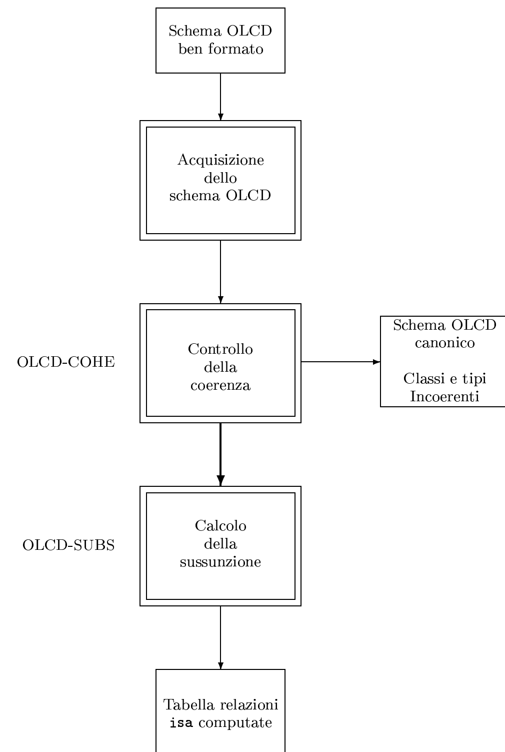

Figura 2.4: Architettura funzionale di OCDL-Designer

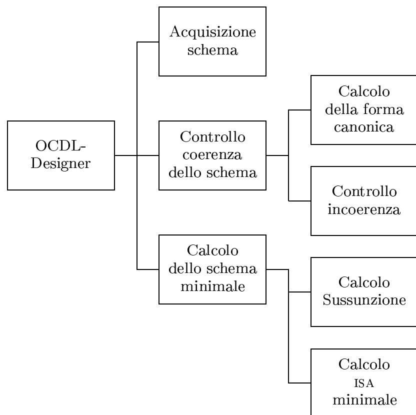

Figura 2.5: Struttura del programma OCDL-Designer

Il programma prende in ingresso un file di testo `nomefile.sc` contenente lo schema iniziale, durante le fasi successive comunica a video eventuali messaggi di errori e incoerenze rilevate, e se l'esecuzione ha termine correttamente i risultati dell'elaborazione vengono scritti in due file: `nomefile.fc` e `nomefile.sb`. Il primo contiene i risultati della trasformazione canonica, il secondo le relazioni di sussunzione e le relazioni `isa` minimali computate.
In seguito al lavoro svolto con questa tesi é stato poi aggiunto un modulo software il quale, avvalendosi dei risultati ottenuti dagli altri moduli, produce il file `nomefile.c`; tale file, scritto in linguaggio C standard, contiene istruzioni che consentono di passare dalla descrizione dello schema della base di dati, opportunamente controllata e manipolata da OCDL-Designer, alla sua realizzazione in UNISQL/X (per maggiori e piú dettagliate spiegazioni si veda il capitolo 4).
La fase di acquisizione consiste nella lettura del file contenente lo schema e la creazione delle relative strutture dinamiche rappresentanti le definizioni dei tipi (classi). Durante questa fase non viene realizzato un controllo sintattico e semantico sistematico, ma solo alcuni controlli di comodo per rilevare eventuali errori nella scrittura del file di input. Si assume infatti che lo schema preso in input sia corretto dal punto di vista sintattico e semantico, e privo di cicli rispetto alle relazioni `isa` e alle definizioni dei tipi valori (*schema ben formato*). Una volta acquisito lo schema ha inizio la fase di generazione dello schema canonico mediante l'applicazione delle funzioni ricorsive ι e μ,

definite nella teoria. Tale processo di generazione 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. Una volta determinata la forma canonica si passa all'esecuzione dell'algoritmo di sussunzione, che permette di ricalcolare tutte le relazioni `isa`, riuscendo a rilevare anche le relazioni implicite nella descrizione originale di tipi e classi e determinando i tipi e le classi equivalenti. Inoltre vengono calcolate le relazioni di sussunzione esistenti tra antecedente e conseguenti di regole. Il programma si compone di tre moduli richiamati dal programma principale:

- Acquisizione schema
- Controllo coerenza dello schema
- Calcolo dello schema minimale

Per informazioni circa questi moduli si veda [Gar95] e [Ric98]. A quelle appena viste si aggiunge poi una quarta procedura che, sulla base delle informazioni messe a disposizione da OCDL-Designer, crea il file `nomefile.c` poc'anzi menzionato (la cui descrizione é data nel capitolo 4).
Nel seguito di questa sezione viene invece data una breve panoramica sulle strutture dati utilizzate per mappare lo schema della base di dati in memoria centrale.

2.8 STRUTTURE DATI

Lo schema viene memorizzato in tre strutture a lista:

- **listaN** per i tipi valore e i tipi classe;
- **listaB** per i tipi base;
- **listaO** per le operazioni (metodi).

2.8.1 DESCRIZIONE listaN

La struttura di un elemento della prima lista è la seguente:

```
typedef struct lN {
char *name;
int type;
int new;
```

```
L_sigma *sigma;
L_sigma *iota;
L_gs *gs;
struct lN *next;
} lN;
```

I campi hanno i seguenti significati:

NAME : stringa di caratteri che descrive il nome della classe o del tipo valore;

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

- **T** = nome tipo valore
- **C** = nome classe primitiva
- **D** = nome classe virtuale
- **RA_V** = nome antecedente di una regola di tipo classe virtuale
- **RC_V** = nome conseguente di una regola di tipo classe virtuale
- **RA_T** = nome antecedente di una regola di tipo valore
- **RC_T** = nome conseguente di una regola di tipo valore
- **O** = nome operazione

NEW : campo intero che assume il valore `FALSE` se si tratta di un nome appartenente allo schema iniziale, `TRUE` altrimenti;

SIGMA : puntatore alla lista che rappresenta la descrizione originale;

IOTA: puntatore alla lista che rappresenta il risultato della forma canonica;

GS: puntatore alla lista che rappresenta l'insieme $GS(N)$;

NEXT : puntatore al successivo elemento della lista.

STRUTTURE DELLE LISTE RAPPRESENTANTI LE DESCRIZIONI

Le descrizioni delle classi e dei tipi valori vengono rappresentate come liste di elementi aventi la seguente struttura:

```
typedef struct L_sigma {
char *name;
```



```
int type;
void *field;
L_sigma *next;
struct lN *and;
} L_sigma;
```

I campi hanno i seguenti significati:

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

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
- **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
- **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
- **O** = nome tipo operazione

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 elemento di tipo `bt`;

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

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

2.8.2 DESCRIZIONE listaB

La struttura di un elemento della lista `listaB` è la seguente:

```
typedef struct lB {
char *nameB;
bt *descB;
struct lB *next;
} lB;
```

I campi hanno i seguenti significati:

NAMEB : stringa di caratteri che descrive il nome del tipo base;

DESCB: puntatore all'elemento che decrive il tipo base;

NEXT : puntatore al successivo elemento della lista.

STRUTTURA DEGLI ELEMENTI TIPI BASE

Le descrizioni dei tipi base vengono rappresentate come elementi aventi la seguente struttura:

```
typedef struct bt {
int type;
int iMin;
int iMax;
float rvalue;
char *hvalue;
} bt;
```

I campi hanno i seguenti significati:

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;

IMIN: campo utilizzato per il tipo `VINT`, il tipo `RANGE` e per il tipo `VBOOL`, nel primo caso contiene il valore intero, nel secondo caso il valore inferiore del range, infine nel caso di un valore booleano assume il valore 1 (TRUE) o il valore 0 (FALSE);

IMAX: campo utilizzato per il tipo `RANGE`, in tal caso contiene il valore superiore del range;

RVALUE: campo utilizzato per il tipo `VREAL`, in tal caso contiene il valore reale;

HVALUE: campo utilizzato per il tipo `VSTRING`, in tal caso contiene il valore stringa;

2.8.3 DESCRIZIONE listaO

Questa lista ha una struttura perfettamente identica alla lista ListaN, ma é deputata a contenere esclusivamente tutte e sole le operazioni dello schema. In altri termini tutte le classi e i tipi valore vengono memorizzati tramite listaN, le operazioni invece vengono mantenute attraverso listaO, la quale perció conterrá solo strutture di tipo **lN** che hanno nel campo *type* il valore **O**, il quale logicamente dal canto suo non potrá mai essere presente in un elemento di listaN (dal momento che quest' ultima non contiene descrizioni di metodi).

Ulteriori informazioni relative alle strutture dati appena viste si trovano in [Gar95] e [Ric98].

Capitolo 3

UNISQL/X

3.1 PANORAMICA DI UNISQL/X

Gli ultimi anni passati hanno testimoniato l'evoluzione di una nuova generazione di DBMS: c'é stata infatti una fiorente attivitá di sviluppo e esperimenti con sistemi di database che supportano o che comunque si relazionano a modelli object-oriented. Queste attivitá sono state in seguito incrementate sia a causa di necessitá (registratesi nell' ambito di un vasto spettro di applicazioni database) che i tradizionali sistemi relazionali non erano in grado di supportare sia per il crescente bisogno di effettuare un netto salto di produttivitá nello sviluppo delle applicazioni. Il risultato di questo fermento é stato il progressivo affermarsi di una gamma di prodotti che supportano un nuovo modello, object-oriented, il quale tuttavia é stato realizzato in modo da essere compatibile con il modello relazionale. Oltre a ció i nuovi database si trovano comunque a dover supportare, per ovvie ragioni, anche i linguaggi SQL-compatible. Tutta questa compatibilitá ovviamente, se da un lato puó aumentare le difficoltá di realizzazione del pacchetto software, dall' altro rappresenta sicuramente un grosso vantaggio, dal momento che:

- Minimizza il tempo di apprendimento per coloro che, sapendo giá operare nell' ambito relazionale, vogliono imparare questa nuova tecnologia.
- Risolve i problemi di conversione tra programmi. Ció significa che un database relazionale é portabile anche all' interno di un DBMS ad oggetti.

UniSQL/X in particolare si colloca proprio all' interno di questa nuova gamma di prodotti, che uniscono i modelli relazionali con quelli object-

oriented, ed é quindi un sistema *object-relational*. Per quanto riguarda il linguaggio viene usato SQL/X [1], il quale puó essere cosí suddiviso:

- Data Definition Language (DDL).
- Query Manipulation Language (QML).
- Data Manipulation Language (DML).
- Data Control Language (DCL).

3.2 UNISQL/X: GENERALITA'

3.2.1 IL SISTEMA DEI TIPI DI DATO

I TIPI BASE

Si veda la tabella 3.1 a pag. 56, in cui viene riportato l' elenco dei tipi base previsti da UNISQL/X, dando per ognuno di essi anche sia una breve descrizione, sia il nome con cui il tipo puó essere indicato in fase di programmazione.

NOTA: Ogniqualvolta é possibile viene fatto un mapping tra i tipi UNISQL/X e i tipi base C; cosí ad esempio il tipo *smallint* viene tradotto attraverso il tipo C *short*, il tipo *integer* attraverso il tipo C *int* e cosí via.

I tipi base visti in tab. 3.1, quando sono usati per specificare domini di attributi e argomenti di metodi, possono essere elencati e descritti usando il tipo SQL/X aggiuntivo **DB_TYPE**[2], il quale si basa sulle corrispondenze indicate in tabella 3.2 a pag. 57.

NOTE: Relativamente a tale tabella valgono le seguenti:

- Il tipo **DB_TYPE** é usato solo per descrivere i vari tipi all' interno dei domini di attributi o argomenti di metodo, ma non puó essere usato in fase implementativa.
- La tabella si legge nel seguente modo: un attributo di tipo float puó essere descritto attraverso *DB_TYPE_FLOAT*, uno di tipo integer attraverso *DB_TYPE_INTEGER* e cosí via.

[1] Tale linguaggio puó essere visto come una estensione (che preserva la compatibilitá) del linguaggio SQL, ampiamente usato in ambito relazionale.

[2] Per maggiori dettagli si veda il paragrafo 3.2.1.

- La tabella non é completa; mancano infatti i seguenti tipi:

  DB_TYPE_OBJECT: descrive un tipo di dato che specifica che il dominio é una classe definita dall' utente.

  DB_TYPE_ERROR: descrive un tipo speciale usato solo nelle definizioni di argomenti di metodo (e non di attributi). Serve ad indicare che si é verificato un errore.

  DB_TYPE_NULL: indica un tipo indeterminato che descrive appunto dei NULL.

Per elencare e descrivere invece i tipi dei valori usati per settare o accogliere il contenuto di una struttura DB_VALUE si fa uso del C-type (tipo C) **DB_TYPE_C**[3], basato sulle indicazioni contenute in tabella 3.3 a pag. 58, relativamente alla quale valgono le seguenti:

NOTE:

- Anche il tipo *DB_TYPE_C* (come del resto *DB_TYPE*), puó essere utilizzato solo per elencare e descrivere i vari tipi base e non in fase di implementazione.
- La tabella si legge nel seguente modo: se estraggo da una variabile di tipo DB_VALUE un valore di tipo float, lo posso descrivere tramite il tipo *DB_TYPE_C_FLOAT*.
- La tabella non é completa; mancano infatti i tipi **DB_TYPE_C_OBJECT** e **DB_TYPE_C_ERROR**, i quali hanno la stessa descrizione dei corrispondenti costrutti "DB_TYPE".

Esiste anche una corrispondenza tra i "DB_TYPE" e i "DB_TYPE_C", che riportiamo in tabella 3.4 a pag. 59.

TIPI ADDIZIONALI

UNISQL/X definisce anche una serie di tipi addizionali, tra cui ricordiamo:

DB_ERROR: E' il tipo ritornato dalle funzioni dell' interfaccia API per indicare che si é verificato un errore.

DB_OBJECT: Viene usato per indicare un puntatore ad un oggetto di una classe.

DB_TYPE: E' utilizzato per elencare e descrivere i vari tipi base SQL/X, quando si specificano domini di attributi o di argomenti di metodo.

[3] Per maggiori dettagli si veda il paragrafo 3.2.1.

DB_VALUE: E' la rappresentazione di un valore SQL/X. Si tratta della struttura primaria usata per interfacciare valori di attributi e di argomenti di metodo da un lato e variabili definite dall' utente dall' altro.

C-TYPES

UNISQL/X definisce anche una serie di tipi C, che sono riportati in tabella 3.5 a pag. 60, nella quale viene anche riportata una breve descrizione di ciascuno di essi.

3.2.2 IL LINGUAGGIO SQL/X

Come giá detto in precedenza, una delle due vie che i modelli ad oggetti mettono a disposizione per poter accedere ai dati memorizzati in un database é rappresentata dai linguaggi di interrogazione. In particolare UNISQL/X fa uso di un linguaggio compatibile con SQL (utilizzato negli RDBMS), che ha nome SQL/X e che si basa sui seguenti costrutti:

SELECT: Seleziona il valore di uno o piú attributi relativi a tutte le istanze di una classe che soddisfano le condizioni eventualmente (e non obbligatoriamente) specificate. Come negli RDBMS, il nome della classe é indicato nella clausola *FROM*, le condizioni che le istanze devono rispettare invece sono contenute nella clausola *WHERE*.
I risultati della query possono poi anche essere raggruppati o ordinati, usando rispettivamente le clausole *GROUP BY* e *ORDER BY*.
Ad es. la query

```
select x.name,x.age
from Person x
where x.profession=''Employee''
group by x.level
order by x.age;
```

seleziona il nome e l'etá di tutte le persone che svolgono la professione di impiegato, ordinate per etá crescente e raggruppate per livello.

INSERT: Crea una istanza della classe specificata. E' possibile inserire in questo statement anche delle query e/o sottoquery, in modo tale da estrarre, da una o piú classi dello schema, dei dati che possono essere cosí inseriti come istanze in altre classi.
Ad es. l' istruzione

```
insert into Person(name,works-in)
values(''John'',select x.identity
                from Department x
                where x.location=''Sunsweet bouleward 12'');
```

inserisce nella classe *Person* una persona di nome "John", la quale lavora nel dipartimento che si trova in "Susweet boulevard 12".

UPDATE: Consente di modificare il valore di uno o piú attributi di tutte le istanze che soddisfano le eventuali condizioni specificate (nella clausola *WHERE*).
Ad es. l' istruzione

```
update Person
set status=''adult''
where age>=18;
```

modifica il nome di tutte le persone aventi etá maggiore o uguale a 18 anni, ponendo il loro stato uguale ad "adult".

DELETE: Permette di cancellare tutte le istanze di una classe che soddisfano le eventuali condizioni specificate (nella clausola *WHERE*). Ad es. l' istruzione

```
delete
from class Person
where name=''Walter'';
```

elimina tutte le istanze relative alle persone che hanno nome pari a "Walter".

Per maggiori dettagli si veda l' UNISQL/X User Manual.

3.2.3 I METODI

UNISQL/X, essendo basato su un modello ad oggetti, consente di associare alle varie classi di un database dei metodi, che, come visto in precedenza, non sono altro che routine software, le quali implementano tutte e sole le operazioni che é possibile svolgere sulle istanze della classe a cui il metodo stesso é riferito.
In particolare per poter scrivere tali routine é necessario tenere presente le seguenti convenzioni:

1. Le funzioni che implementano i metodi ritornano sempre ***void***. I risultati da restituire al chiamante devono essere posti nella struttura di tipo DB_VALUE (che d' ora in poi indicheremo brevemente anche con il termine "*contenitore*") che occupa il secondo posto nella lista dei parametri.

2. Il primo parametro della lista degli argomenti della funzione é sempre un puntatore al tipo DB_OBJECT, il quale ha il compito di referenziare l' OID dell' oggetto relativamente al quale é stato chiamato il metodo.

3. Il secondo parametro, come giá visto sopra, é sempre riservato al return value del metodo.

4. Il terzo parametro e tutti i seguenti sono invece i parametri veri e propri passati al metodo[4].

5. Tutti gli argomenti del metodo, fatta eccezione per il primo, sono di tipo DB_VALUE, e pertanto all' interno della funzione dovranno essere manipolati sfruttando opportune macro che UNISQL/X mette a disposizione e che possono essere raggruppate nel modo seguente:

   - Per settare il contenuto di variabili di tipo DB_VALUE viene fornito il set di macro aventi nome pari a "**DB_MAKE_**" seguito dal nome del tipo del valore da inserire nel contenitore.
     Ad es. la macro DB_MAKE_STRING consente di memorizzare in una variabile di tipo DB_VALUE un valore di tipo stringa.
   - Per estrarre valori da un contenitore vengono fornite macro aventi nome pari a "**DB_GET_**" seguito dal nome del tipo a cui appartiene la variabile in cui voglio memorizzare quanto estratto.
     Ad es. la macro DB_GET_INTEGER consente di estrarre un intero dal DB_VALUE passatole come parametro.
   - Viene infine fornita la macro DB_VALUE_TYPE, la quale puó essere utilizzata per determinare il tipo del valore memorizzato in un contenitore.

3.2.4 I TRIGGER

I trigger sono oggetti di sistema che implementano un meccanismo per l' inserimento di operazioni definite dall' utente all' interno dell' esecuzione di una sessione di lavoro sul database.

[4] E' ammesso un numero massimo di 12 parametri per ogni metodo.

In altre parole un trigger permette di invocare l' esecuzione di un' azione in risposta ad una specifica attivitá riscontrata all' interno della base di dati.
Possono essere usati per:

- Realizzare vincoli di integritá relativi ad una o piú classi.
- Implementare aggiornamenti paralleli del database.
- Assegnare riferimenti circolari.
- Mantenere statistiche.
- Realizzare cancellazioni in cascata.
- Prevenire accessi non autorizzati a una classe.
- Avvisare che si é verificato un evento.

I principali componenti di un trigger sono:

nome: Serve per distinguere i trigger tra loro. Deve essere unico all' interno del database e non puó coincidere con una Keyword di UNISQL/X.

stato: Consente di specificare se il trigger é attivo o meno (uso al proposito le parole riservate *ACTIVE* e *INACTIVE*). In caso di omissione della clausola il trigger viene considerato attivo.

prioritá: Permette di specificare la prioritá di esecuzione del trigger. E' utile nei casi in cui sia necessario eseguire in uno specifico ordine piú trigger relativi alla stessa classe.

tempo dell' evento: Determina quando la condizione specificata nel trigger deve essere valutata. In particolare l' uso delle parole chiave *BEFORE* o *AFTER* indica che la condizione va verificata rispettivamente prima o subito dopo che l' evento che ha scatenato il trigger é stato processato.

tipo di evento: Specifica quale é l' attivitá eseguita sul database che scatena il trigger. In particolare vi sono due categorie di eventi che possono provocare l' esecuzione di un trigger:

- Eventi applicati ad ogni singola istanza (detti *istance events*). Si dividono in:
  - **INSERT**

  - **UPDATE**
  - **DELETE**

  L' esecuzione di una di queste istruzioni su di una istanza qualunque della classe specificata scatena il trigger.
- Eventi applicati ad ogni statement (detti *statement events*). Si dividono in:
  - **STATEMENT INSERT**
  - **STATEMENT UPDATE**
  - **STATEMENT DELETE**

  Tali eventi sono un metodo per provocare l' esecuzione di un trigger non in corrispondenza ad ogni evento di insert, update o delete che si verifica relativamente alla classe specificata, ma solo in corrispondenza della prima esecuzione dello statement stesso.
  In altre parole scrivere ad es. *"AFTER STATEMENT INSERT ON Person"* significa che il trigger in cui tale clausola é inserita viene attivato solo dopo il primo (e non dopo ogni) insert fatto sulla classe *Person*.

Nel primo caso si parlerá per analogia anche di *istance trigger*, nel secondo invece di *statement trigger*.
Esistono infine altri due eventi che possono provocare l' esecuzione di un trigger che sono:

- **COMMIT**
- **ROLLBACK**

target dell' evento: Permette di specificare la classe relativamente alla quale deve verificarsi l' evento per scatenare il trigger. Si noti che:

- UNISQL/X non consente di riferire trigger ad una classe virtuale.
- Nel caso in cui il tipo di evento che provoca l' esecuzione del trigger sia un' istruzione di *delete*, é possibile specificare in questa clausola non solo il nome della classe, ma anche dell' attributo a cui il delete deve essere riferito per far sí che il trigger venga attivato.

condizione: Si tratta di una espressione, valutata solo dopo che il trigger é stato attivato, che produce un risultato booleano e che deve restituire TRUE affinche l' azione associata al trigger possa essere eseguita.

azione: Rappresenta quello che viene fatto nel caso la condizione sia verificata.
Le azioni possibili sono:

REJECT: Provoca il rollback dell' attivitá che ha scatenato il trigger (ad es. annulla la modifica fatta attraverso uno statement update).

INVALIDATE TRANSACTION: Invalida la transazione che ha attivato il trigger, impedendone (anche successivamente) il commit. Tale transazione dovrá perció essere necessariamente abortita.

PRINT: Mostra semplicemente un messaggio di testo sullo schermo. Si usa soprattutto in fase di test del database.

INSERT: Aggiunge una o piú ulteriori istanze a una classe.

UPDATE: Modifica il valore di uno o piú attributi all' interno di una specificata classe.

DELETE: Rimuove una o piú istanze da una data classe.

CALL (o EVALUATE): Esegue la chiamata ad uno dei metodi definiti all' interno del database.

Vediamo, per chiarire le idee, il seguente esempio:

```
CREATE TRIGGER new_cost_hotel
STATUS ACTIVE
PRIORITY 1
BEFORE UPDATE ON hotel(cost)
IF new.cost > 200
EXECUTE REJECT;
```

Si tratta di un trigger di nome *new_cost_hotel*, stato attivo, prioritá 1, il quale viene eseguito prima di ogni modifica fatta all' attributo *cost* di una qualunque istanza della classe *hotel*, solo peró nel caso in cui il valore che si intende inserire sia maggiore di 200. In caso tale condizione non sia verificata l' update viene rifiutato.

3.3 INTERAGIRE CON UNISQL/X

Cominciamo a questo punto ad analizzare le caratteristiche principali di UNISQL/X, dal momento che questo é l' OODBMS che é stato scelto per implementare fisicamente gli schemi giá preventivamente filtrati dall' ODB-Tools.
Esistono quattro modalitá diverse per interagire con UNISQL/X:

1. Attraverso l' *Interactive SQL/X Processor* (ISQL/X Processor).
2. Attraverso il *Command Line SQL/X Processor*.
3. Attraverso l' *Application Programming Interface* (API).
4. Tramite codice *Embedded SQL/X*.

Non prenderemo in esame la prime due delle sopracitate modalitá, dal momento che esse non sono state utilizzate per la realizzazione del programma relativo alla presente tesi. Sia l' interactive sia il Command Line SQL/X Processor sono infatti utili soprattutto se si vogliono eseguire delle query su un database giá creato, mentre risultano abbastanza inefficienti per l' utente (rispetto alle altre due modalitá) nella fase di creazione della base di dati. Ci limitiamo semplicemente a sottolineare che si tratta di due modalitá di accesso a UNISQL/X molto simili, distinte quasi unicamente per il fatto che ISQL/X é *Menu Driven*, mentre invece l' SQL/X Processor prevede la scrittura delle varie query da eseguire in un opportuno file[5].

3.4 EMBEDDED SQL/X

Consente di inserire all' interno di un programma scritto in un linguaggio di alto livello (come ad es. il C) delle istruzioni SQL/X, dette anche istruzioni *"embedded"*, le quali permettono di estrarre, modificare e aggiungere informazioni a database UNISQL/X.
Grazie infatti al preprocessore **ESQL/X** (Embedded SQL/X), é possibile tradurre statement SQL/X in corrispondenti sequenze di istruzioni C, producendo cosí in uscita un programma che puó essere compilato tramite un semplice compilatore ANSI C.
Gli statement di tipo embedded si riconoscono per il fatto che sono introdotti dal prefisso **EXEC SQLX** (o **EXEC SQL**).
Volendo ora entrare un pó piú nel dettaglio dell' attivitá di analisi di un programma ESQL/X, possiamo affermare che essa si articola in tre fasi distinte:

1. **PREPROCESSING:** durante questa fase, vengono analizzate tutte le istruzioni ESQL/X e vengono sostituite con il corrispondente codice C. Il preprocessore non é tuttavia ancora in comunicazione con il database e perció non é possibile valutare se gli statement ESQL/X

[5] Ció risulta preferibile nel caso di applicazioni che devono essere periodicamente eseguite. Per maggiori informazioni si veda [Unisql].

sono anche semanticamente e non solo sintatticamente corretti. In altre parole a questo livello possono essere considerate valide ad esempio delle "select" formalmente corrette ma riferite a classi non esistenti nel database considerato. Errori di questo tipo non verranno rilevati finché il programma non viene eseguito.
In questa prima fase inoltre viene fatto un primo parziale controllo sulle dichiarazioni delle varie strutture dati C utilizzate nel programma, evidenziando in output eventuali errori trovati.
Ad ogni modo, gran parte del codice viene semplicemente copiato in output, affidandone l' analisi alle fasi successive.

2. **COMPILAZIONE:** durante questa fase viene eseguita la compilazione vera e propria, la quale consente perció di evidenziare eventuali errori semantici e sintattici contenuti nella porzione del programma scritta originariamente in linguaggio C (dal momento che il codice C generato dal precompilatore non puó mai produrre errori).
Solo quando tutti gli errori sono stati rimossi il file prodotto in output viene "*linkato*" con le opportune librerie, generando cosí un file eseguibile.

3. **ESECUZIONE:** Il file risultante dalla fase di compilazione viene caricato ed eseguito.
Solo in questa fase é possibile validare anche semanticamente i vari statement SQL/X riscontrati nella fase di preprocessing. Ogni istruzione che interagisce col database viene perció nuovamente analizzata, e nel caso venga rilevato un errore per un qualsiasi motivo, un opportuno codice viene memorizzato in una variabile speciale consultabile dal programma ESQL/X stesso, il quale puó perció differenziare il suo comportamento a seconda dei casi, scegliendo l' azione piú opportuna da eseguire.

3.4.1 LE *HOST VARIABLE*

Le *Host Variable* (variabili ambiente) costituiscono il principale legame tra il linguaggio C ed ESQL/X, dal momento che rappresentano il mezzo per estrarre o inserire valori nel database. E' infatti in tali variabili che ESQL/X memorizza i valori estratti dal database (tramite ad es. una select) o che nel database vuole inserire (tramite ad es. un insert).
Il loro uso é regolato dalle seguenti norme:

1. Tutte le *Host Variable* devono essere preventivamente dichiarate in una sezione speciale delimitata dagli statement **BEGIN DECLARE SECTION** e **END DECLARE SECTION**.

2. Ogniqualvolta si usa una *Host Variable* all' interno di uno statement ESQL/X é necessario indicarne il nome preceduto dal prefisso "**:**" (due punti).

3. E' possibile inizializzare le *Host Variable* al momento della loro dichiarazione, esattamente come si fa per le variabili C.

4. L' uso di variabili di tipo ambiente pone in primo piano il problema della corrispondenza tra tipi UNISQL/X e tipi C, la quale é regolata dalla tabella 3.6 a pag. 61. Tale tabella indica semplicemente quale é il tipo a cui deve appartenere una *Host Variable* per potersi interfacciare con ciascun tipo UNISQL/X.

5. Come si puó notare dalla tabella 3.6 stessa, é possibile in linea di principio utilizzare il tipo DB_VALUE per trasferire da e verso il database un qualunque tipo di dato. Chiaramente in questo caso per manipolare le *Host Variable* sará necessario utilizzare le funzioni apposite che l' interfaccia API[6] mette a disposizione, dal momento che una variabile di tipo DB_VALUE non puó essere acceduta direttamente.

3.4.2 I CURSORI

Consentono di accedere, una tupla alla volta, ai dati estratti dal database attraverso uno statement di *SELECT*.
Per utilizzare un cursore sono necessari i seguenti passi:

1. **DICHIARAZIONE** del cursore.

2. **APERTURA** del cursore.

3. **FETCHING** delle varie tuple individuate. In questa fase vengono utilizzate delle *Host Variable* per memorizzare i valori dei vari attributi relativi agli oggetti estratti dal database.

4. **CHIUSURA** del cursore.

Vediamo per chiarire un esempio:

[6] Si veda al proposito il paragrafo 3.5 a pag. 50

```
EXEC SQLX DECLARE c CURSOR FOR
     SELECT name
     FROM Person
     WHERE age < :cutoff;

EXEC SQLX OPEN c;

for (;;)
{
  EXEC SQLX FETCH c INTO :name;
  if (SQLCODE == 100)
     break;

  printf("NAME= %s\n",name);
}

EXEC SQLX CLOSE c;
```

In questo caso viene dichiarato un cursore di nome "c", il quale seleziona il nome di tutti gli oggetti della classe *Person* aventi etá inferiore al contenuto di una *Host Variable* di nome *cutoff*, la quale deve essere perció opportunamente e preventivamente dichiarata all' interno di una "Declare Section" precedente la definizione del cursore.
In seguito il cursore stesso viene aperto e vengono recuperati, tupla per tupla, i nomi degli oggetti precedentemente estratti, i quali vengono semplicemente stampati su std-output. Infine si procede alla chiusura del cursore.
Si noti come il loop che consente di analizzare una alla volta le tuple selezionate é realizzato attraverso un ciclo for infinito, che si interrompe solo quando la variabile di sistema SQLCODE vale 100, il che come vedremo nel prossimo paragrafo corrisponde al caso in cui non vi siano piú oggetti da analizzare.

3.4.3 ESQL/X COMMUNICATION AREA (SQLCA)

La *ESQL/X communication area* é la struttura in cui vengono memorizzati i risultati dell' esecuzione dei vari statement SQL/X; al suo interno distinguiamo tra gli altri i seguenti campi:

SQLCODE: Contiene un codice che indica il risultato dell' ultimo statement di SELECT, FETCH, INSERT, UPDATE o DELETE.
Tale codice puó avere valore:

'0' (zero): Lo statement é stato eseguito con successo.

'100' (cento): Lo statement é stato eseguito con successo ma non sono stati trovati dati da processare. Ció ha significati diversi a seconda di quale é l' ultima istruzione SQL/X eseguita:

- Se si tratta di un' istruzione di FETCH significa che non ci sono piú tuple da estrarre perché il cursore corrente é stato completamente analizzato.
- Se si tratta di uno statement SELECT significa che esso non ha selezionato alcun oggetto.
- Se si tratta di un' istruzione di INSERT significa che ho cercato di inserire in un attributo il contenuto di una *Host Variable* vuota.
- Se si tratta di uno statement CALL significa che il metodo invocato non ha ritornato alcun valore.

'<0' (minore di zero): Si é verificato un errore.

SQLFILE e SQLLINE: Identificano rispettivamente il nome del file sorgente e la posizione al suo interno in cui é contenuto l' ultimo statement SQL/X eseguito.

SQLERRMC e SQLERRML: Se il valore di SQLCODE é <0, SQLERRMC referenzia una stringa contenente la descrizione dell' errore verificatosi, SQLERRML invece contiene un intero indicante la lunghezza di tale stringa.

SQLERRD: Si tratta di un array contenente informazioni addizionali relative all' ultimo statement SQL/X eseguito.

3.4.4 SOMMARIO DEGLI STATEMENT ESQL/X

Viene riportato a seguire un elenco dei principali statement SQL/X che posso inserire all' interno di un programma C, ricordando che ciascuno di essi, per poter essere utilizzato in modo "Embedded", deve essere preceduto dal prefisso "EXEC SQLX" (o "EXEC SQL")[7]:

BEGIN DECLARE SECTION: Introduce una sezione del programma in cui sono dichiarate le *Host Variable*.

CALL: Invoca un metodo relativamente ad un dato oggetto.

CLOSE: Chiude un cursore e libera le risorse di sistema ad esso associate.

[7] Per maggiori dettagli si veda [Unisql].

CONNECT: Connette il programma a cui appartiene al database specificato.

COMMIT WORK: Esegue il commit di tutte le transazioni correnti.

CREATE: Crea una classe.

DECLARE CURSOR: Dichiara un cursore.

DELETE: Elimina uno o piú oggetti da una data classe.

DISCONNECT: Elimina la connessione tra il programma a cui appartiene e il database specificato.

DROP: Elimina una classe.

END DECLARE SECTION: Chiude la sezione del programma in cui sono dichiarate le *Host Variable*.

FETCH: Estrae dal cursore specificato il record successivo all' ultimo analizzato e pone i dati ad esso relativi in opportune variabili ambiente.

FETCH OBJECT: Estrae da un oggetto specificato il valore di uno o piú attributi.

GRANT: Concede privilegi di accesso ad un dato database.

INSERT: Aggiunge un nuovo oggetto ad una classe.

OPEN: Apre un cursore.

RENAME: Permette di cambiare il nome ad una classe.

REVOKE: Revoca privilegi di accesso ad un dato database.

ROLLBACK WORK: Esegue il rollback di tutte le transazioni che non hanno ancora raggiunto il commit.

UPDATE: Modifica il valore di uno o piú attributi relativi a tutti gli oggetti di una classe che soddisfano le condizioni specificate. Se tali condizioni mancano vengono modificate tutte le istanze della classe.

UPDATE OBJECT: Modifica uno o piú attributi dell' oggetto specificato.

3.5 L' INTERFACCIA API

Si tratta di un' interfaccia che consente di accedere a UNISQL/X senza utilizzare il linguaggio SQL/X. Puó essere pertanto utilizzata in programmi applicativi che non si appoggiano a tale linguaggio per interagire col database. Per questo motivo l' estensione di ODB-Tools realizzata con la presente tesi consente di produrre come output di OCDL-Designer un file con estensione ".c", il quale é scritto in linguaggio C e contiene istruzioni API che permettono (compilando ed eseguendo il file suddetto) di creare fisicamente tutte le strutture dati descritte nello schema di database correntemente analizzato. Di seguito vengono riportate le principali istruzioni utilizzate, limitandosi per ognuna di esse a descriverne gli effetti, senza soffermarsi su dettagli quali tipo e numero dei parametri utilizzati in input, oppure codici di errore e valori forniti in output, per i quali rimandiamo alla lettura dell' *API Reference Manual* di UNISQL/X.

3.5.1 ISTRUZIONI PER LA GESTIONE DI DATABASE

Si tratta di istruzioni che consentono operazioni di gestione di un database, quali il restart, il trattamento delle transazioni e il controllo dello spazio di lavoro. Tra queste ricordiamo:

FUNZIONI PER L' ACCESSO AD UN DATABASE

db_login: Precede il restart del database e serve a impostare user name e password che in seguito dovranno essere inseriti per accedere al database[8].

db_restart: Effettua il restart di un database esistente[9].

db_shutdown: Chiude la connessione al database precedentemente costituita. Tutte le transazioni in corso (che non hanno ancora raggiunto il commit) vengono abortite.

[8] E' chiaramente possibile anche non specificare alcuna password; in tal modo si consente l' accesso ad un qualsiasi utente.

[9] Il database deve perció essere stato preventivamente creato tramite l' istruzione *createdb*.

FUNZIONI PER LA GESTIONE DELLE TRANSAZIONI

db_abort_transaction: Abortisce tutte le transazioni che non hanno ancora raggiunto il commit.

db_commit_transaction: Esegue il commit di tutte le transazione correnti.

FUNZIONI PER LA GESTIONE DEGLI ERRORI

db_error_code: Fornisce in output un intero che identifica l' ultimo errore verificatosi all' interno del database.

db_error_string: Restituisce una stringa che descrive a parole l' ultimo errore verificatosi all' interno del database.

3.5.2 ISTRUZIONI PER LA GESTIONE DEI VALORI

Sono istruzioni utilizzate per esaminare e modificare i valori degli attributi, l' accesso ai quali é sempre realizzato attraverso una struttura di comunicazione chiamata **DB_VALUE**. In particolare ai fini della presente tesi si sono rivelate utili:

FUNZIONI PER L' ACCESSO A CONTENITORI DI VALORI

Ogni volta che estraggo, tramite le opportune istruzioni, un valore da un attributo, esso viene sempre messo in una variabile di tipo DB_VALUE che poi andrá riconvertita per poter essere utilizzata nel modo tradizionale. Tale riconversione é effettuata attraverso istruzioni il cui nome é dato da **db_get_<*nometipo*>**; a seconda del valore di *nometipo* la struttura di tipo DB_VALUE viene trasformata in una variabile di tipo opportuno, seguendo lo schema riportato nella tabella 3.7 a pag. 62:

NOTE: Dall' analisi di tale tabella si puó inoltre osservare che:

- Il recupero di una collezione di valori é sempre effettuato attraverso la funzione *db_get_collection* (sia che si tratti di un set, un multiset o una sequenza), la quale pone il contenuto del DB_VALUE passatole come parametro in una variabile di tipo DB_COLLECTION.
- La funzione *db_get_string_size* ritorna un intero che é la dimensione della stringa passata come parametro.

FUNZIONI PER LA COSTRUZIONE DI CONTENITORI DI VALORI

Ogni volta che voglio inserire un valore all' interno di un attributo devo prima trasformarlo in un dato di tipo DB_VALUE, il quale verrá poi dato in pasto alle opportune istruzioni che effettuano l' inserimento vero e proprio. Questo compito é svolto da un insieme di istruzioni il cui nome é dato da **db_make_<*nometipo*>**; a seconda del tipo della variabile da mettere all' interno della struttura di comunicazione dovró utilizzare una diversa funzione, seguendo la tabella 3.8 a pag. 63:

NOTE: Relativamente a tale tabella si puó inoltre osservare che:

- L' istruzione *db_make_null* non corrisponde a nessun tipo base definito in UNISQL/X, dal momento che ha come unica funzione quella di inizializzare la struttura di tipo DB_VALUE specificata a NULL.
- L' istruzione *db_make_error* invece trasporta all' interno di una variabile di tipo DB_VALUE un codice di errore, che in UNISQL/X é perció di tipo DB_ERROR. Tale istruzione si usa in genere per settare il valore di ritorno di un metodo in modo da indicare che si é verificato un errore.
- L' istruzione *db_make_object* non corrisponde a nessuno dei tipi base visti in tabella 3.1 (pag. 56), dal momento che serve per memorizzare in strutture di tipo DB_VALUE degli OID di oggetti, che perció saranno sempre di tipo DB_OBJECT.

3.5.3 ISTRUZIONI PER LA GESTIONE DELLO SCHEMA

Si tratta di funzioni usate per creare e modificare classi all' interno di un database UNISQL/X. Distinguiamo:

FUNZIONI PER LA DEFINIZIONE DI CLASSI

db_add_attribute: Aggiunge un attributo ad una classe.

db_add_argument: Descrive il dominio di un argomento relativo ad un metodo specificato. Si puó usare anche per definire il tipo del "return value" del metodo stesso.

db_add_method: Definisce un metodo relativamente ad una classe dello schema.

db_add_method_file: Specifica il nome del file in cui deve essere ricercato il body del metodo specificato[10].

db_add_super: Aggiunge una superclasse alla classe specificata. In particolare quest' ultima eredita dalla superclasse tutti gli attributi e i metodi, a meno che questo non comporti la violazione delle regole di ereditarietá o l' ereditarietá stessa venga modificata a causa dei meccanismi di risoluzione dei conflitti.

db_create_class: Crea una nuova classe avente il nome specificato, a meno che non esista giá nello schema una classe omonima, nel qual caso viene riportato in output un opportuno codice d' errore.

3.5.4 ISTRUZIONI PER LA GESTIONE DEGLI OGGETTI

Sono istruzioni che operano su istanze di classi definite all' interno del database, consentendone la creazione, la modifica e la cancellazione. Si tratta di una categoria di funzioni molto ampia, di cui peró é stata utilizzata, ai fini della presente tesi, solo la seguente:

db_get: Si tratta della funzione base usata per estrarre il valore di un attributo dell' oggetto specificato[11].

NOTA: Questa funzione, al contrario di quelle viste finora, é stata utilizzata non in fase di creazione dello schema (nel file con estensione ".c"), bensí all' interno del body dei vari metodi creati (come si vedrá in seguito) per implementare le varie rule definite sullo schema stesso.

[10] Si noti che é necessario specificare il nome assoluto del file, il quale deve essere un file oggetto o comunque deve avere estensione "**.o**". Tale file non deve esistere al momento in cui viene eseguita questa istruzione, ma é sufficiente che venga creato prima di invocare per la prima volta il metodo in questione (momento in cui viene creato il link dinamico tra il metodo ed il suo body).

[11] Tale valore viene messo in una struttura di tipo DB_VALUE, per il cui trattamento si veda il paragrafo 3.5.2 a pag. 51.

3.5.5 ISTRUZIONI PER LA GESTIONE DEI TRIGGER

Queste funzioni consentono di creare, eseguire e manipolare trigger.
Un trigger come sappiamo é un oggetto di sistema che é creato per agganciare a determinati eventi che si possono verificare all' interno del sistema l' esecuzione di opportune operazioni o comandi definiti dall' utente[12].
Tra le istruzioni appartenenti a questa categoria ricordiamo:

db_create_trigger: Crea un nuovo trigger, specificandone obbligatoriamente nome, stato, prioritá ed eventi collegati.

db_drop_trigger: Elimina il trigger specificato.

db_rename_trigger: Assegna un nuovo nome al trigger specificato.

3.5.6 ISTRUZIONI PER LA GESTIONE DELLE COLLEZIONI

Sono funzioni usate per creare e manipolare oggetti di tipo *collection*. Tali oggetti possono essere raggruppati in tre tipologie diverse:

SET: insieme non ordinato di valori dello stesso tipo priva di duplicati.

SEQUENCE: insieme ordinato di valori dello stesso tipo con possibilitá di duplicati.

MULTISET: insieme non ordinato di valori (non necessariamente tutti dello stesso tipo) con possibilitá di duplicati.

UNISQL/X in particolare mette a disposizione un unico tipo (puntatore a *DB_COLLECTION*) per gestire tutte le collezioni; in altre parole ogni insieme di valori viene memorizzato in una struttura di tipo DB_COLLECTION, lasciando poi alle varie funzioni che manipolano tale struttura il compito di distinguere se si tratta di un set, di una sequence o di un multiset.
Per questo motivo possiamo distinguere:

COLLECTION FUNCTIONS: gestiscono tutte le collezioni senza distinguere la loro tipologia.
Tra queste ricordiamo:

db_col_add: Aggiunge un elemento ad una data collezione.

[12] Per maggiori informazioni si veda il paragrafo 3.2.4 a pag. 40.

db_col_create: Costruisce una nuova collezione.

db_col_drop: Elimina un elemento dalla collezione specificata.

db_col_find: Ricerca all' interno di una collezione un valore, ritornandone, se lo trova, la posizione all' interno della collezione stessa.

db_col_get: Estrae da una collezione l' elemento che occupa la posizione specificata.

db_col_ismember: Verifica se un particolare valore appartiene o meno ad una data collezione.

db_col_put: Inserisce nella posizione specificata un elemento.

db_col_size: Determina la dimensione di una collezione.

SET FUNCTIONS: si occupano dei set e dei multiset.

SEQUENCE FUNCTIONS: gestiscono le sequence.

NOTA BENE: Non vengono forniti gli elenchi delle principali *set functions* e *sequence functions*, data la loro analogia pressocché totale con la lista appena vista (cambiano infatti solo i nomi delle funzioni, che si possono ottenere semplicemente sostituendo nelle funzioni viste alla sigla *col* rispettivamente *set* (per le set functions) e *seq* (per le sequence functions)).

SQL/X type	Implementation name	Description
char(n)	char *	Stringhe di caratteri a lunghezza fissa
varchar(n) (o string)	DB_VARCHAR char *	Stringhe di caratteri a lunghezza variabile
nchar(n)	DB_NCHAR	Stringhe di caratteri a lunghezza fissa contenenti caratteri nazionali
nchar varying(n)	DB_VARNCHAR	Stringhe di caratteri di lunghezza variabile contenenti caratteri nazionali
bit(n)	DB_BIT char *	Stringhe di bit a lunghezza fissa
bit varying(n)	DB_VARBIT char *	Stringhe di bit di lunghezza variabile
numeric (o decimal)	DB_NUMERIC	Numeri interi a precisione specificabile
integer	DB_INT32 int	Numeri interi
smallint	short	Numeri interi a precisione ridotta
monetary	DB_MONETARY	Tipo apposito per memorizzare valori monetari
float	float	Numeri reali a precisione singola
double	double	Numeri reali a doppia precisione
date	DB_DATE	Tipo apposito per memorizzare le date
time	DB_TIME	Tipo apposito per memorizzare gli orari
timestamp	DB_TIMESTAMP	Tipo apposito per memorizzare una combinazione di data e orario
set	DB_COLLECTION *	Set di valori dello stesso tipo con duplicati
multiset	DB_COLLECTION *	Set di valori appartenenti ad uno o piú tipi di dati
sequence (o list)	DB_COLLECTION *	Set di valori dello stesso tipo senza duplicati

Tabella 3.1: Il sistema dei tipi base in UNISQL/X

SQL/X base type	DB_TYPE
char(n)	DB_TYPE_CHAR
varchar(n) (o string)	DB_TYPE_VARCHAR (o DB_TYPE_STRING)
nchar(n)	DB_TYPE_NCHAR
nchar varying(n)	DB_TYPE_VARNCHAR
bit(n)	DB_TYPE_BIT
bit varying(n)	DB_TYPE_VARBIT
numeric (o decimal)	DB_TYPE_NUMERIC
integer	DB_TYPE_INTEGER
smallint	DB_TYPE_SMALLINT (o DB_TYPE_SHORT)
monetary	DB_TYPE_MONETARY
float	DB_TYPE_FLOAT
double	DB_TYPE_DOUBLE
date	DB_TYPE_DATE
time	DB_TYPE_TIME
timestamp	DB_TYPE_TIMESTAMP
set	DB_TYPE_SET
multiset	DB_TYPE_MULTISET
sequence (o list)	DB_TYPE_SEQUENCE (o DB_TYPE_LIST)

Tabella 3.2: Descrizione del tipo DB_TYPE

SQL/X base type	DB_TYPE
char(n)	DB_TYPE_C_CHAR
varchar(n) (o string)	DB_TYPE_C_CHAR
nchar(n)	DB_TYPE_C_NCHAR
nchar varying(n)	DB_TYPE_C_NCHAR
bit(n)	DB_TYPE_C_BIT
bit varying(n)	DB_TYPE_C_BIT
numeric (o decimal)	DB_TYPE_C_NUMERIC
integer	DB_TYPE_C_INT
smallint	DB_TYPE_C_SHORT
monetary	DB_TYPE_C_MONETARY
float	DB_TYPE_C_FLOAT
double	DB_TYPE_C_DOUBLE
date	DB_TYPE_C_DATE
time	DB_TYPE_C_TIME
timestamp	DB_TYPE_C_TIMESTAMP
set	DB_TYPE_C_SET
multiset	DB_TYPE_C_SET
sequence (o list)	DB_TYPE_C_SET

Tabella 3.3: Descrizione del tipo DB_TYPE_C

SQL/X base type	DB_TYPE
DB_TYPE_BIT	DB_TYPE_C_BIT
DB_TYPE_CHAR	DB_TYPE_C_CHAR
DB_TYPE_DATE	DB_TYPE_C_DATE
DB_TYPE_DOUBLE	DB_TYPE_C_DOUBLE
DB_TYPE_ERROR	DB_TYPE_C_ERROR
DB_TYPE_FLOAT	DB_TYPE_C_FLOAT
DB_TYPE_INTEGER	DB_TYPE_C_INT
DB_TYPE_LIST	DB_TYPE_C_SET
DB_TYPE_MONETARY	DB_TYPE_C_MONETARY
DB_TYPE_MULTISET	DB_TYPE_C_SET
DB_TYPE_NCHAR	DB_TYPE_C_NCHAR
DB_TYPE_NULL	-
DB_TYPE_NUMERIC	DB_TYPE_C_NUMERIC
DB_TYPE_OBJECT	DB_TYPE_C_OBJECT
DB_TYPE_SEQUENCE	DB_TYPE_C_SET
DB_TYPE_SET	DB_TYPE_C_SET
DB_TYPE_SHORT	DB_TYPE_C_SHORT
DB_TYPE_SMALLINT	DB_TYPE_C_SHORT
DB_TYPE_STRING	DB_TYPE_C_CHAR
DB_TYPE_TIME	DB_TYPE_C_TIME
DB_TYPE_TIMESTAMP	DB_TYPE_C_TIMESTAMP
DB_TYPE_VARBIT	DB_TYPE_C_BIT
DB_TYPE_VARCHAR	DB_TYPE_C_CHAR
DB_TYPE_VARNCHAR	DB_TYPE_C_NCHAR

Tabella 3.4: Corrispondenza tra "DB_TYPE" e "DB_TYPE_C"

C type	Description
DB_C_BIT	E' la struttura in linguaggio C equivalente a DB_BIT e a DB_VARBIT (char ∗).
DB_C_CHAR	E' la struttura in linguaggio C equivalente a DB_CHAR (char ∗).
DB_C_COLLECTION	E' la struttura in linguaggio C equivalente a DB_COLLECTION.
DB_C_DATE	E' la struttura in linguaggio C equivalente a DB_DATE.
DB_C_DOUBLE	E' il tipo double del linguaggio C.
DB_C_FLOAT	E' il tipo float del linguaggio C.
DB_C_IDENTIFIER	E' la struttura in linguaggio C equivalente a DB_IDENTIFIER.
DB_C_INT	E' il tipo integer del linguaggio C.
DB_C_LONG	E' il tipo long del linguaggio C.
DB_C_MONETARY	E' la struttura del linguaggio C che memorizza i valori monetari.
DB_C_NCHAR	E' la struttura del linguaggio C che memorizza le stringhe contenenti caratteri nazionali.
DB_C_NUMERIC	E' la struttura in linguaggio C equivalente a DB_NUMERIC (char ∗).
DB_C_OBJECT	E' la struttura in linguaggio C equivalente a DB_OBJECT.
DB_C_SHORT	E' il tipo short del linguaggio C.
DB_C_TIME	E' la struttura in linguaggio C equivalente a DB_TIME.
DB_C_TIMESTAMP	E' la struttura in linguaggio C equivalente a DB_TIMESTAMP.
DB_C_VARCHAR	E' la struttura in linguaggio C equivalente a DB_VARCHAR (char ∗).
DB_C_VARNCHAR	E' la struttura in linguaggio C equivalente a DB_VARNCHAR.
DB_TYPE_C	E' il tipo usato per elencare e descrivere i tipi dei valori usati per settare o accogliere il contenuto di DB_VALUE.

Tabella 3.5: Descrizione dei tipi C definiti in UNISQL/X

SQL/X type	C type
integer	int
smallint	short
float	float
double	double
string	char *
varchar	varchar
bit	bit
bit varjing	bit varjing
<any>	DB_VALUE
<object>	DB_OBJECT
time	DB_TIME
timestamp	DB_TIMESTAMP
date	DB_DATE
monetary	DB_MONETARY
set	DB_COLLECTION *
multiset	DB_COLLECTION *
sequence	DB_COLLECTION *

Tabella 3.6: Corrispondenza tra tipi UNISQL/X e tipi delle *Host variable*

Function	C type definition
db_get_bit	DB_C_BIT
db_get_char	DB_C_CHAR
db_get_collection	DB_COLLECTION
db_get_date	DB_DATE
db_get_double	DB_C_DOUBLE
db_get_error	DB_ERROR
db_get_float	DB_C_FLOAT
db_get_int	DB_INT32
db_get_monetary	DB_MONETARY
db_get_nchar	DB_C_NCHAR
db_get_numeric	DB_C_NUMERIC
db_get_object	DB_OBJECT
db_get_short	DB_C_SHORT
db_get_string	DB_C_CHAR
db_get_string_size	DB_INT32
db_get_time	DB_TIME
db_get_timestamp	DB_TIMESTAMP

Tabella 3.7: Corrispondenze tra istruzioni "db_get" e tipi C

Function	**SQL/X type**
db_make_bit	bit(n)
db_make_char	char(n)
db_make_date	date
db_make_double	double
db_make_error	DB_ERROR
db_make_float	float
db_make_int	integer
db_make_monetary	monetary
db_make_multiset	multiset
db_make_nchar	nchar(n)
db_make_null	-
db_make_numeric	numeric
db_make_object	DB_OBJECT
db_make_sequence	sequence (o list)
db_make_set	set
db_make_short	smallint
db_make_string	string (o varchar(n))
db_make_time	time
db_make_timestamp	timestamp
db_make_varbit	bit varying(n)
db_make_varchar	varchar(n) (o string)
db_make_varnchar	nchar varying(n)

Tabella 3.8: Corrispondenze tra istruzioni "db_make" e tipi UNISQL/X

Capitolo 4

IMPLEMENTAZIONE DI SCHEMI DI DATI ODMG93 IN UNISQL/X

Il lavoro di questa tesi é stato rivolto alla realizzazione di un modulo software che, avvalendosi dei risultati ottenuti dagli altri moduli componenti di OCDL-Designer, produce in output il file `nomefile.c` (dove `nomefile` é il nome del file dato in input al traduttore *ODL-Trasl*).
Tale file contiene il codice C standard che, sfruttando l' interfaccia API (*Application Programming Interface*) fornita da UNISQL/X, consente, se opportunamente compilato ed eseguito, di creare fisicamente il database descritto nel file `nomefile.sc`.
La figura 4.1 mostra la struttura del programma OCDL-Designer, cosí come si presenta dopo le modifiche apportate con il lavoro di questa tesi.

Scopo di questo capitolo é perció quello di descrivere il modulo "*creazione database fisico*"; tale descrizione si articola in due parti:

1. Nel paragrafo 4.1 viene riportata una panoramica generale di quanto realizzato, in modo tale da evidenziare in modo chiaro e sintetico i concetti cardine del lavoro svolto.

2. Nel paragrafo 4.2 e seguenti ci si addentra invece in modo piú specifico nella descrizione del codice, allo scopo di facilitare il piú possibile la comprensione delle motivazioni che hanno portato a certe scelte implementative piuttosto che ad altre.
   In questa fase si é cercato di descrivere le varie funzioni in modo sufficientemente preciso; infatti penso che l' utilitá primaria di una descrizione di codice sia quella di guidare il lettore a comprenderne la

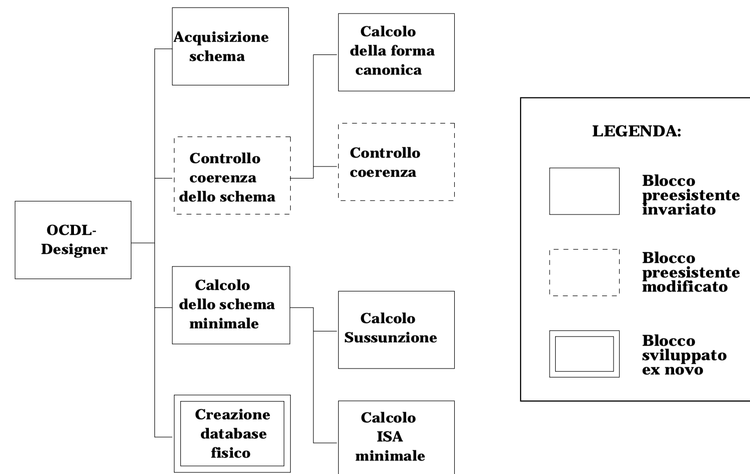

Figura 4.1: Struttura del programma OCDL-Designer

struttura, e perció una descrizione troppo superficiale risulta quasi inutile per chi legge, dal momento che non contiene un livello di dettaglio sufficiente da poter essere usata (anche solo in prima battuta) per accedere al codice ed eventualmente manipolarlo o completarlo. Questa considerazione assume a mio avviso ancora piú valore se si pensa che la mia tesi é parte di un progetto piú complesso (e perció il modulo da me realizzato probabilmente in futuro sará soggetto a modifiche e/o integrazioni).

4.1 MODULO "CREAZIONE DATABASE FISICO" - STRUTTURA GENERALE

La creazione dello schema fisico del database in UNISQL/X consiste nelle seguenti azioni:

- Vengono create tutte le classi dello schema.

- Tutti i tipi valore vengono assimilati a classi e creati come tali, dal momento che UNISQL/X non supporta il concetto di `type`.

- Tutte le classi virtuali vengono tradotte come classi primitive. Ció é reso necessario dal fatto che UNISQL definisce il concetto di *Virtual class*, ma non consente l' ereditarietá tra classi primitive e virtuali. Naturalmente la scelta fatta comporta la necessitá di creare anche le strutture che realizzano la C.S. di appartenenza ad ogni classe virtuale. In particolare, per ogni classe virtuale presente nello schema, viene associata ad ogni classe primitiva una coppia trigger-metodo, la quale verifica, per ogni oggetto inserito, se tale oggetto é dotato o meno della struttura necessaria per poter appartenere alla classe virtuale a cui la coppia suddetta si riferisce.
- I tipi base definiti dall' utente (memorizzati all' interno della lista `ListaB`) vengono assimilati ai corrispondenti tipi base predefiniti.
- Vengono aggiunti ad ogni classe tutti i relativi attributi, sia che essi mappino in tipi primitivi sia che mappino in classi o tipi valore.
- Nel caso sia necessario imporre delle restrizioni sul dominio di un attributo viene utilizzato il costrutto di **trigger** messo a disposizione da UNISQL/X. In tabella 4.1 sono riportate le scelte fatte per mappare in UNISQL/X i tipi predefiniti di OLCD.

OLCD Type	UNISQL/X Type	Trigger?
integer	integer	no
real	float	no
string	string	no
boolean	integer	yes
range	integer	yes
vinteger	integer	yes
vreal	float	yes
vstring	string	yes
vboolean	integer	yes
{ }	set	no
< >	list	no

Tabella 4.1: Mapping dei tipi tra OLCD e UNISQL/X

- Vengono create tutte le signature dei metodi definiti sulle classi dello schema (che sono memorizzate in `ListaO`), tralasciando di considerare i relativi body, di cui peraltro non v'é traccia nelle strutture dati di OCDL-Designer. Fa eccezione il caso in cui sia necessario imporre delle restrizioni sul dominio degli argomenti o del tipo di ritorno dei

metodi, nel qual caso viene creata anche la porzione di body necessaria allo scopo.

- Per ogni classe vengono create fisicamente tutte le relazioni di ereditarietá a cui la classe stessa partecipa col ruolo di figlia, sia che si tratti di relazioni esplicitamente definite sia che invece si tratti di relazioni implicite calcolate tramite l' algoritmo di sussunzione.
- Per ogni rule definita sullo schema viene creata una coppia trigger-metodo, la quale, per ogni oggetto inserito nella classe a cui la rule si riferisce, controlla che esso soddisfi le condizioni espresse nella regola. In particolare si possono verificare tre casi:
  - L' oggetto non soddisfa una delle condizioni espresse nell' antecedente della rule; in questo caso la regola non viene considerata di interesse per l' oggetto, che non viene rimosso (indipendentemente dal fatto che verifichino o meno tutte le condizioni espresse nel conseguente).
  - L' oggetto soddisfa tutte le condizioni dell' antecedente ma non tutte quelle del conseguente; in questo caso la rule si considera violata e l' oggetto viene rimosso dalla classe in cui era stato inserito.
  - L' oggetto soddisfa tutte le condizioni sia dell' antecedente sia del conseguente; in tal caso la rule é soddisfatta e l' oggetto non viene rimosso.

  NOTA BENE: La distinzione prevista in OLCD tra regole di tipo classe virtuale e regole di tipo valore scompare, a causa del fatto che, come detto in precedenza, UNISQL non consente di definire il concetto di *type*, e perció i tipi valore vengono implementati come classi primitive.

- I `Newtype` che derivano dall' operazione di controllo di coerenza dello schema e che OCDL-Designer inserisce all' interno di `ListaN` vengono ignorati.
- Il modulo "`creazione_database_fisico`" realizza anche un controllo su tutti i nomi di classi, attributi e metodi dello schema per evitare che possano verificarsi utilizzi impropri di parole riservate di UNISQL/X[1]. Se ció si verifica:

[1] Tale controllo é quantomai opportuno visto l' elevato numero di parole riservate di UNISQL/X (circa 370).

  - Viene riportato su schermo un messaggio di errore che evidenzia le keyword usate impropriamente.
  - Non viene scritto nulla nel file `nomefile.c`.

- Nel caso lo schema sia incoerente non viene creato il file *nomefile.c* e viene riportato su schermo un messaggio d' errore che indica che per poter creare fisicamente il database bisogna renderne coerente lo schema.

- Nel caso vi siano rule riferite alla *TOP Class*, viene riportato in output un messaggio di "Warning", il quale avverte che tutte le rule che si vuole implementare fisicamente devono essere riferite ad una classe definita dall' utente, quindi offre una duplice possibilitá:

  - Abortire la scrittura del file `nomefile.c`, per consentire all' utente di correggere lo schema e rilanciare OCDL-Designer.
  - Proseguire ignorando, in fase di creazione del database, le rule riferite alla *TOP Class*.

- Ogni rule contenente attributi non definiti all' interno delle opportune classi dello schema non viene accettata in fase di creazione del database.
  Anche in questo caso viene riportato in output un messaggio di "Warning", il quale avverte del problema e offre la stessa duplice alternativa vista al punto precedente.

- Ogni classe primitiva o tipo valore dello schema eredita tutte le rule relative alle sue eventuali superclassi.
  E' questo un concetto che finora nell' ambito di ODB-Tools non era mai stato precisato, in quanto ci si limitava a dire che una regola era riferita ad una data classe, ma che la necessitá di interfacciarsi con un OODBMS ha portato subito in primo piano.
  A livello implementativo, il fatto che le regole siano realizzate attraverso delle coppie trigger-metodo (le quali godono del principio di ereditarietá) fa sí che non sia necessario prevedere la stesura di codice supplementare al fine di garantire il rispetto della suddetta norma.

- Siccome UNISQL/X richiede la specifica del nome assoluto del (o dei) file in cui sono contenuti i body dei vari metodi dello schema correntemente analizzato, si é cercato di fare in modo che ció fosse il meno vincolante possibile per l' utente; per questo motivo si é stabilito che il nome della directory in cui mettere i file suddetti

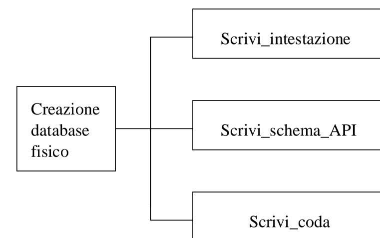

Figura 4.2: Struttura del modulo per la creazione del database fisico in UNISQL/X

debba essere preventivamente memorizzato in una variabile di sistema chiamata **UNISQL_METHOD**. Al momento dell' esecuzione di OCDL-Designer viene quindi controllata l' esistenza o meno di tale variabile, ed in caso di assenza viene prima abortita la scrittura del file *nomefile.c*, quindi viene riportato un messaggio che avverte di settare UNISQL_METHOD e rilanciare in seguito OCDL-Designer.
In questo modo é possibile per l' utente controllare ed eventualmente cambiare con estrema facilitá il direttorio in cui memorizzare i body dei metodi.

4.2 DESCRIZIONE DELLE FUNZIONI

La creazione del database in UNISQL/X come giá detto in precedenza avviene attraverso la scansione delle strutture dati utilizzate da OCDL-Designer (ListaN, ListaB e ListaO) e la successiva creazione del file `nomefile.c`.
In figura 4.2 é riportato il diagramma PHOS relativo al modulo che stiamo considerando[2].

4.2.1 DESCRIZIONE DELLA FUNZIONE "SCRIVI_INTESTAZIONE"

Questa funzione scrive il codice C che corrisponde all' intestazione del file `nomefile.c`; tale intestazione contiene sia le istruzioni API che consentono di aprire il database in cui si vuole inserire lo schema, sia la definizione delle strutture dati e delle variabili usate all' interno del file.

[2]Per maggiori informazioni circa i diagrammi PHOS si veda l' appendice B.1 a pag. 171.

Da notare é il fatto che a questo livello non é possibile creare fisicamente il database, poiché tale operazione in UNISQL/X é fattibile solo da prompt utilizzando il comando `createdb`.
Per questo motivo é necessario, dopo aver compilato il file `nomefile.c`, passare come parametro all' eseguibile ottenuto il nome del database che deve perció essere stato preventivamente creato tramite il suddetto comando.

4.2.2 DESCRIZIONE DELLA FUNZIONE "SCRIVI_SCHEMA_API"

E' la funzione che di fatto si occupa di scrivere il codice C che crea il database in UNISQL/X. La figura 4.3 a pag. 72 riporta la sua descrizione attraverso il modello PHOS[3].

Entriamo ora un pó nel dettaglio dell' analisi della struttura di questa funzione che rappresenta indubbiamente il cuore della parte di software realizzato con il lavoro di questa tesi:

1. Dopo avere inizializzato le varie variabili e allocato spazio in memoria per tutti i puntatori utilizzati, la funzione scandisce una prima volta **ListaN**[4], e per ogni record della lista:

   (a) Se si tratta di un' operazione, di un Newtype[5] oppure di un antecedente o conseguente di rule lo ignora, passando direttamente all' elemento successivo di `ListaN`.

   (b) Se invece si tratta di una classe, la funzione verifica che essa non sia equivalente ad una classe giá esistente, quindi in caso di risposta negativa la crea[6], altrimenti passa direttamente al successivo record di `listaN`.

[3]Per maggiori dettagli circa il modello PHOS si veda l' appendice B.1 a pag. 171.
[4]In UNISQL é necessario creare preliminarmente tutte le classi, dal momento che, quando aggiungo loro i rispettivi attributi, se questi mappano in altre classi, é necessario che queste ultime esistano giá.
[5]Ricordiamo che col termine *Newtype* intendiamo dei tipi o classi non appartenenti allo schema iniziale, che derivano dall' operazione di controllo di coerenza dello schema, e che OCDL-Designer inserisce all' interno di `ListaN`. Per approfondimenti si veda [Gar95].
[6]Tale verifica é resa necessaria, per tutte le classi incontrate eccetto la prima, a causa di quanto verrá specificato al punto (c) (sottoparagrafo iii.)

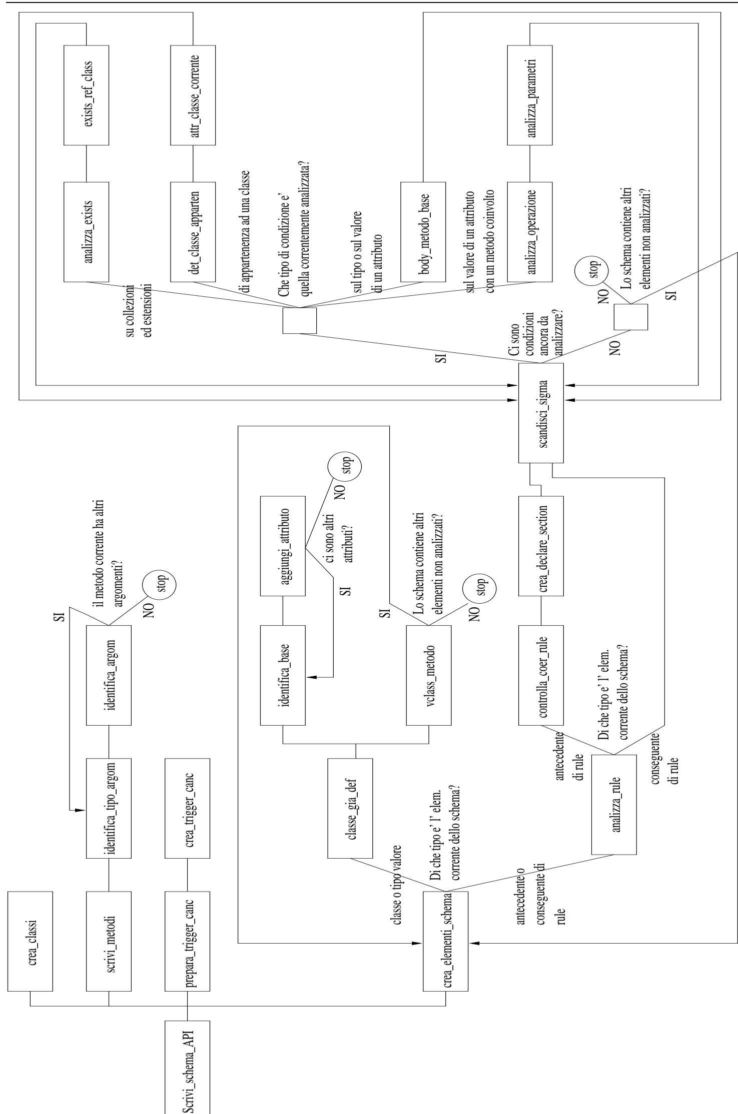

Figura 4.3: Diagramma PHOS della funzione Scrivi_schema_API

NOTA BENE: Al punto precedente ho parlato di classi, ma lo stesso discorso lo si puó ritenere valido anche per i tipi valore; infatti UNISQL non supporta il concetto di `type`, e quindi, per tradurre quanto in OLCD é definito attraverso tale clausola, devo far ricorso al concetto di classe, ragion per cui d' ora in poi, salvo avviso contrario, ogni cosa diremo riguardo le classi sará da intendersi come applicabile anche ai tipi valore.

(c) La creazione della classe correntemente analizzata é affidata alla funzione "**crea-classi**", la quale:

i. Per prima cosa esegue un controllo preventivo volto ad eliminare la possibilitá di assegnare alla classe stessa un nome coincidente con una keyword di UNISQL, cosa che porterebbe in fase di creazione del database a comportamenti imprevedibili del sistema[7].
In particolare, nel caso in cui venga rilevato un conflitto, il programma riporta subito in output un messaggio di errore per avvisare che si sta tentando di assegnare ad una classe un nome non consentito; parallelamente viene poi settato opportunamente un flag che servirá ad impedire la creazione del file `nomefile.c`[8].

In particolare, se supponiamo di avere definito uno schema in cui sia presente una classe di nome *Level* e una di nome *Section* e di averlo memorizzato (in sintassi OLCD) nel file **Schema1.sc**, al momento dell' esecuzione di OCDL-Designer verrá abortita la scrittura del file **Schema1.c** e verrá riportato in output il seguente messaggio d' errore:

```
Writing output file (Schema1.c):  ABORT
 (Schema contains following UNISQL Keywords:
   LEVEL : (class name)
```

[7]Tale controllo assume grande rilevanza, dal momento che in UNISQL l' elenco delle parole riservate é molto lungo (piú di 370 unitá) e coinvolge termini di uso estremamente comune (es. `name, level, section, count`).

[8]L' uso del flag é giustificato dalla necessitá di non interrompere la scansione di *listaN* nel momento in cui rilevo l' utilizzo improprio di una keyword di UNISQL, bensí di proseguirla ed ultimarla, in modo da consentire la visualizzazione di un messaggio d' errore completo, all' interno del quale vengano riportati tutti i conflitti di nome rilevati (e non solo il primo). Per far ció é necessario quindi ricorrere all' uso del suddetto flag, il quale consente di spostare al termine dell' esame di *listaN* l' interruzione del processo di scrittura sul file `nomefile.c` e la cancellazione di quanto giá scritto sul file medesimo.


```
   SECTION : (class name)
  If you want write file Schema1.c, please resolve
  name conflicts and rerun OCDL-Designer)
```

ii. Se non sono stati rilevati conflitti di nome viene creata la classe.

iii. Dopo aver creato la classe si esamina la lista dei `gs` ad essa associata (facilmente accessibile grazie al puntatore `gs` dei record appartenenti a `ListaN`), allo scopo di verificare se esistono classi equivalenti a quella appena definita[9].
In caso di risposta affermativa, tutte le classi equivalenti tra loro vengono associate ad un' unica rappresentazione fisica, a cui viene dato il nome indicato nel campo `name` del record corrente di `ListaN`. In caso contrario invece si passa semplicemente al record successivo della lista.

2. Prima di aggiungere alle varie classi i rispettivi attributi, nonché analizzare le rule ad esse relative, vengono create le signature di tutti i metodi definiti sullo schema appena creato[10].
Tale operazione é realizzata attraverso la funzione "**scrivi-metodi**", la quale scandisce la lista **ListaO**, e per ogni record in essa contenuto:

(a) Determina la classe a cui il metodo si riferisce (tale informazione é giá contenuta nella lista dei sigma che parte dal record di *listaO* relativo al metodo correntemente analizzato).

(b) Controlla che il metodo non abbia un nome coincidente con una keyword di UNISQL. In caso ció accada viene da un lato riportato in output un messaggio di errore, dall' altro settato opportunamente un flag in modo del tutto simile a quanto visto per le classi con nome improprio.

[9]Vengono considerate solo equivalenze con classi o tipi valore, non con *Newtype* oppure con antecedenti o conseguenti di rule.

[10]Questa operazione é necessaria dal momento che, quando in seguito si procederá all' analisi della struttura delle varie classi e rules componenti lo schema, potrá capitare di imbattersi in una regola contenente una chiamata a metodo. E' chiaro che in questo caso UNISQL, per poter riconoscere tale metodo, richiede che esso sia giá stato definito e associato alla classe opportuna (anche se non é necessario che sia giá stato scritto il suo body, dato che quest' ultimo viene esaminato solo a run-time, nel momento in cui la rule viene effettivamente eseguita).

Ad esempio, se suppongo che nello schema precedentemente considerato, accanto alle due classi (*Level* e *Section*) esista anche un metodo *Count* relativo ad una classe di nome *Person*, il messaggio riportato in output da OCDL-Designer sará:

```
Writing output file (Schema1.c):  ABORT
 (Schema contains following UNISQL Keywords:
    LEVEL : (class name)
    SECTION : (class name)
    COUNT : (method name in class PERSON )
 If you want write file Schema1.c, please resolve
 name conflicts and rerun OCDL-Designer)
```

(c) Se viene riscontrato un conflitto di nome il programma abortisce l' analisi del metodo corrente; in caso contrario scrive sul file `nomefile.c` il codice necessario per creare fisicamente il metodo, quindi prosegue nell' analisi eseguendo i passi sotto riportati.

(d) Scandisce la lista referenziata dal puntatore `sigma` del record corrente di `lista0`, la quale conterrá informazioni circa gli eventuali argomenti del metodo appena creato[11]; per ognuno di essi viene invocata la funzione "**identifica-tipo-argomento**", la quale determina il tipo dell' argomento corrente:

- Se si tratta di un tipo valore o di una classe aggiunge direttamente l' argomento al metodo[12].
- Se si tratta di un tipo primitivo, chiama la funzione "**identifica-argomento**", la quale individua di quale tra i tipi primitivi ammessi si tratta e a seconda del risultato aggiunge in modo opportuno l' argomento corrente al metodo.

(e) Viene quindi inserita in *nomefile.c* una parte di codice che ha lo scopo di creare, una volta che tale file sará stato compilato ed eseguito, un ulteriore file (chiamato <**nomefile**>**_met.ec**) che sará deputato a contenere i body di tutti i metodi definiti sullo schema correntemente esaminato.

[11] Si noti come il tipo di ritorno del metodo venga trattato a tutti gli effetti come un argomento, piú precisamente come il primo argomento memorizzato nella lista puntata dal campo `sigma` del record corrente di `Lista0`.

[12] Tale aggiunta viene fatta semplicemente arricchendo il codice del file *nomefile.c* dell' istruzione API db_add_argument, la quale consente appunto, una volta creato un metodo relativo ad una classe, di definire i suoi parametri

In particolare per ogni metodo verrá creata automaticamente in ogni caso l' intestazione del body e, solo se necessario, il codice embedded ESQL/X che effettua un controllo sui parametri (compreso il tipo di ritorno) passati alla funzione che implementa il body del metodo stesso.
Cioé se ad esempio ho uno schema siffatto (scritto in OLCD):

```
prim Filo = ^ [ tipo-filo :  range 1 2,
    sezione :  range 0.0 1000.0,
    corrente-impiego :  range 0.0 +inf ];

operation Filo = real f dividi
    (in param1 :  range 0 1000, in param2 :  real );
```

In questo caso per il metodo `dividi` viene creata, in seguito all' esecuzione del file *nomefile.c*, non solo l' intestazione ma anche quella porzione di body che controlla (a run time) che il valore passato come `param1` sia compreso tra 0 e 1000.

NOTA BENE: Il file <**nomefile**>**_met.ec** di cui si é parlato sopra, é scritto in EMBEDDED ESQL/X; pertanto dovrá essere preliminarmente precompilato (attraverso il precompilatore **esqlx**) e solo successivamente compilato (usando ad esempio il programma **gcc**).

La sopranominata intestazione del body di un metodo é da intendersi comprensiva di:

- Indicazione del nome del metodo e dei suoi parametri nella sintassi opportuna.
- Indicazione degli estremi della DECLARE SECTION[13].
- Messaggi (inseriti come commento) che indicano a chi si occuperá in un secondo momento materialmente della scrittura del body del metodo corrente la posizione corretta in cui inserire sia la dichiarazione delle variabili usate sia il codice vero e proprio.

[13] Per Declare Section si intende quella parte del body di un metodo contenente la dichiarazione di tutte le variabili usate per interagire con il database (chiamate all' interno della manualistica di UNISQL/X *Host-variables*).

Ad esempio l' intestazione del sopracitato *dividi* avrá la seguente forma:

```
void Filo_dividi(DB_OBJECT *o,
        DB_VALUE ret_val, DB_VALUE param1,
        DB_VALUE param2)
{
EXEC SQLX BEGIN DECLARE SECTION;

/* Insert here local variable declarations */

EXEC SQLX END DECLARE SECTION;

/* Place here the method body */

}
```

In particolare si noti che:

- Il nome del body é in realtá composto dal nome del metodo piú il nome della classe a cui quest' ultimo appartiene: ció é dovuto alla necessitá di mantenere distinti tra loro i body dei vari metodi definiti all' interno dello schema, dal momento che é stata fatta la scelta di raggrupparli tutti in un unico file, ragion per cui sarebbe impossibile, senza lo stratagemma suddetto, distinguere tra loro funzioni omonime relative a classi distinte (caso possibile in OLCD).
- Nel caso in cui l' esecuzione del file *nomefile.c* porti alla scrittura del codice di controllo di uno o piú parametri del metodo, tale codice viene aggiunto subito dopo la chiusura della `Declare Section`, a meno che non si tratti del *Return Value*, il cui controllo viene invece sempre inserito subito prima della parentesi graffa che chiude il body.
- Per quanto riguarda invece la spiegazione relativa alla sintassi usata per descrivere i parametri si veda il paragrafo 3.2.3 a pag. 39.

3. Prima di scandire una seconda volta `ListaN` allo scopo di aggiugere alle varie classi i relativi attributi e di analizzare le varie rules definite sullo schema correntemente analizzato, viene chiamata la funzione

"`prepara_trigger_cancellazione`", la quale, assieme alle sottofunzioni "`crea_trigger_cancellazione`" e "`trova_padre`", realizza una parte del sistema che si occupa di eliminare dal database gli oggetti che non rispettano le regole di integritá.
Tale sistema é esemplificabile attraverso i seguenti punti:

- Ogni rule viene tradotta in UNISQL attraverso un trigger, il quale ha come compito quello di chiamare, dopo ogni inserimento fatto all' interno della classe a cui il trigger stesso é riferito, un metodo, il quale non fa altro che controllare se l' oggetto inserito soddisfa le condizioni espresse nella rule, provvedendo poi a eliminarlo in caso di risposta negativa.
- Tale eliminazione é sempre accompagnata dalla presentazione all' utente di un messaggio, nel quale viene specificato l' OID dell' istanza cancellata e i nomi della classe a cui tale istanza apparteneva nonché della rule violata. Se ad esempio supponiamo che l' oggetto appartenga alla classe *Person*, abbia OID pari a 420980 e violi la rule *Rule1*, il messaggio assumerá la forma:

```
WARNING: object 420980 in class Person rejected:
    (rule Rule1 violated)
```

- Ogni metodo cosí realizzato non é altro che la traduzione in Embedded ESQL/X di una regola definita all' interno dello schema. I dettagli circa i criteri e le metodologie con cui i vari costrutti che posso trovare all' interno di una rule vengono tradotti verranno riportati in seguito, al punto (h).

Un sistema di questo tipo, pur avendo il vantaggio di essere semplice ed abbastanza immediato, presenta un difetto facilmente evidenziabile; se ogni metodo corrispondente ad una rule elimina l' oggetto relativamente al quale é stato invocato, nel caso tale oggetto non soddisfi i vincoli imposti dalla rule stessa, eventuali altri metodi che cercassero di accedere al medesimo oggetto non vi riuscirebbero, provocando il verificarsi di un errore di sistema.
D' altro canto ognuno dei metodi suddetti é legato come sappiamo ad un trigger, il quale per sua natura viene sempre eseguito in corrispondenza al verificarsi dell' evento scatenante (nel nostro caso l' inserimento nel database dell' oggetto da esaminare).
Per questo motivo, se mi trovo nella condizione di avere piú di una rule

(e di conseguenza piú di un trigger) relativamente ad una stessa classe, puó capitare, al momento del primo inserimento, che l' istanza inserita non soddisfi una delle regole di integritá e quindi venga cancellata dal relativo metodo. Se in seguito tuttavia il sistema si trova a dovere ancora eseguire alcuni dei trigger definiti su quella classe, si troverá impossibilitato a farlo e genererá un errore di sistema.
Tale problema é stato risolto evitando di cancellare direttamente l' oggetto, nel caso violi una rule, ma limitandosi a settare opportunamente un flag in modo che sia chiaro che l' istanza in questione andrá eliminata. In seguito, solo dopo avere terminato il controllo di tutte le varie regole relative alla classe a cui l' oggetto appartiene, viene eseguito un metodo il quale non fa altro che controllare lo stato del flag suddetto e quindi eliminare l' istanza, se necessario.
Affinché tuttavia tale metodo sia automaticamente eseguito al momento giusto é necessario legarlo ad un trigger, il quale dovrá essere peró eseguito sempre dopo tutti gli altri trigger relativi alla classe corrente (i quali servono per verificare il rispetto o meno delle rule), dal momento che solo allora il valore del sopracitato flag sará veramente indicativo della necessitá o meno di eliminare l' oggetto inserito. Ció é realizzabile grazie al fatto che UNISQL consente di mettere in sequenza temporale l' esecuzione di piú trigger relativi alla stessa classe, semplicemente associando loro, al momento della creazione, una prioritá. In questo modo, assegnando ai trigger di controllo del flag prioritá inferiore agli altri , é possibile far sí che vengano eseguiti per ultimi.
A questo punto peró si presenta un altro problema, legato all' ereditarietá dei trigger (analoga a quella degli attributi e metodi); infatti non é possibile prevedere una coppia trigger-metodo di controllo del flag per ogni classe, poiché cosí facendo ogni classe erediterebbe anche le coppie relative alle sue superclassi, con il risultato che potrebbe accadere che il sistema controlli il flag e cerchi di cancellare l' oggetto corrente piú di una volta, provocando il verificarsi di un errore.
Tale problema si risolve creando una coppia trigger-metodo adibita a controllo flag relativamente a tutte e sole quelle classi che:

(a) Non hanno superclassi

(b) Se non hanno sottoclassi devono avere associata ad esse stesse almeno una rule.

(c) Se hanno una o piú sottoclassi é sufficiente che ad esse stesse e/o a una delle sottoclassi sia riferita almeno una rule.

Solo il rispetto delle regole sopra enunciate infatti consente, visto il principio di ereditarietá dei trigger, di avere associata ad ogni classe a cui é riferita almeno una regola una sola coppia trigger-metodo adibita al controllo del flag opportuno ed all' eventuale eliminazione dell' oggetto corrente[14].

Scendendo ora un pó piú nel dettaglio di quale parte del meccanismo appena descritto viene realizzato dalla funzione "`prepara_trigger_cancellazione`", diciamo che essa:

(a) Scorre *ListaN* e determina tutte le classi a cui é riferita almeno una rule e che non hanno nessuna superclasse[15].

(b) Per ognuna di esse chiama la funzione "`crea_trigger_cancellazione`", la quale da un lato crea la coppia trigger-metodo che elimina dal database, se necessario, l' oggetto correntemente analizzato (nel caso esso non soddisfi almeno una regola di integritá), dall' altro si preoccupa di scrivere nel file **<nomefile>_met.ec** il body del metodo suddetto.

4. A questo punto la funzione "`scrivi_schema_API`" scandisce un' ulteriore volta la lista `ListaN`, per poter sia aggiungere alle classi create in precedenza i rispettivi attributi, sia realizzare le coppie trigger-metodo che traducono fisicamente le rules definite sullo schema corrente.
In particolare per ogni record incontrato la funzione esamina il campo `type` e:

   (a) Se si tratta di un *Newtype* o di un' operazione, passa semplicemente al record successivo[16].

[14] Il caso di una classe a cui non sia associata alcuna rule non é interessante dal momento che in quel caso non potrá mai capitare che il controllo del flag imponga la cancellazione dell' oggetto corrente, poiché non ci sono regole che possano settare opportunamente, se violate, il flag stesso. Proprio in virtú di questo il meccanismo appena descritto puó permettersi di non tenere sotto controllo le classi prive di rules, lasciando che queste ultime, in alcuni casi (a seconda della struttura dello schema), ereditino da superclassi la coppia trigger-metodo di cui si é discusso finora, visto che comunque tale coppia non potrá mai sortire l' effetto di cancellare l' oggetto corrente.

[15] Sono, in base a quanto detto poc'anzi, tutte e sole le classi in relazione alle quali é necessario creare la coppia trigger-metodo di livello 1 che controlla il flag.
In particolare la ricerca delle classi prive di superclassi viene effettuata attraverso la funzione "`trova_padre`", la quale sfrutta per raggiungere lo scopo la descrizione della gerarchia di classi dello schema contenuta nella lista puntata dal campo *gs* di ogni elemento di *ListaN* corrispondente a una classe.

[16] Passo al record successivo anche nel caso io incontri un' operazione, dal momento che i metodi li ho giá creati scandendo la lista `Lista0`.

(b) Se invece si tratta di una classe (primitiva o virtuale) o di un tipo valore, viene chiamata la funzione `classe_gia_def`, la quale, attraverso l' analisi della lista dei *gs*, determina se tale classe (tipo valore) é stata effettivamente creata oppure é stata assimilata ad una altra classe, essendo ad essa equivalente, nel qual caso si passa semplicemente all' elemento successivo di *ListaN*, senza analizzare la lista dei *sigma* relativa alla classe corrente.

(c) Nel caso la classe sia stata effettivamente creata, si procede ad esaminare la lista puntata dal campo `sigma` del record corrente di `ListaN`. Piú precisamente:

- Se trova l' indicazione di una o piú superclassi della classe corrente la ignora[17].
- Se incontra invece un attributo si puó comportare in tre modi diversi:
  - Se l' attributo mappa in una classe o in un tipo valore viene chiamata la funzione "**aggiungi_attributo**", la quale non fa altro che creare l' attributo in questione.
  - Se l' attributo mappa invece in un tipo base predefinito (es. integer, real, char ecc.) viene invocata la funzione "**identifica_base**", la quale identifica di quale tipo base si tratta e quindi chiama "**aggiungi_attributo**".
  - Se l' attributo mappa infine in un tipo base definito dall' utente, viene scandita la lista **ListaB**, per determinare a quale tipo base predefinito devo ricondurmi; quindi, una volta fatto ció, mi comporto come al punto precedente.

  NOTA BENE: Anche in questo caso é previsto un controllo volto ad evitare che si possa assegnare ad un attributo un termine che é anche Keyword di UNISQL[18]. Se ció avviene ci si comporta in modo esattamente analogo a quanto visto nei casi di conflitto generato da nomi di classi o di metodi definiti all' interno dello schema.

(d) Successivamente, sempre nel caso in cui il record corrente di *ListaN* rappresenti una classe, viene scandita la lista puntata dal campo `gs`, allo scopo di determinare tutte le superclassi della

[17] Questo perché la determinazione delle relazioni di ereditarietá della classe corrente verrá fatta in seguito tramite la lista puntata dal campo `gs` del record corrente di *ListaN*. Il motivo di ció é dovuto al fatto che, cosí facendo, tengo conto anche dei risultati dell' algoritmo di sussunzione e di conseguenza delle eventuali relazioni di ereditarietá derivate.

[18] Il controllo é svolto dalla funzione "**aggiungi_attributo**".

classe corrente, sia quelle esplicitamente indicate nello schema, sia quelle desunte tramite l' algoritmo di sussunzione. In particolare per ogni record della lista cosí esaminata vengono effettuati i seguenti controlli:

- Se si tratta di un `Newtype`, di una rule o di una classe (tipo valore) equivalente alla classe correntemente esaminata si passa al record successivo senza compiere alcuna operazione[19].
- Se si tratta invece di una classe o di un tipo valore non equivalente alla classe indicata dal record corrente di `ListaN`, significa che siamo di fronte ad una relazione di ereditarietá che puó essere sia definita sia derivata tramite l' algoritmo di sussunzione. In tal caso la funzione "`scrivi_schema_API`" si preoccupa di creare fisicamente il legame trovato tra le due classi (tipi valore)[20].

(e) Se poi il record corrente di *ListaN* é relativo ad una classe virtuale che non é stata assimilata a nessuna altra classe, oltre a quanto appena visto viene chiamata anche la funzione "`vclass_metodo`", la quale si preoccupa di implementare la condizione sufficiente (C.S.) di appartenenza alla classe virtuale in questione. Tale C.S. viene realizzata associando ad ogni classe primitiva dello schema una coppia trigger-metodo, la quale verifica, per ogni oggetto inserito, se tale oggetto é dotato o meno della struttura necessaria per poter appartenere alla classe virtuale correntemente analizzata. La verifica suddetta viene fatta:

- Sfruttando l' istruzione di *insert*, seguita dall' istruzione di *rollback*, in modo tale da non lasciare traccia del tentato inserimento.
- A questo punto, analizzando la variabile di sistema **SQLCODE**, é possibile sapere se il precedente inserimento é andato a buon fine e perció se la C.S. di appartenenza é verificata o meno.

NOTA BENE: Si potrebbe legittimamente pensare che, essendo soddisfatta la C.S. di appartenenza alla classe virtuale corren-

[19] A noi interessa determinare tutte e sole le classi padre della classe correntemente esaminata, e perció non siamo interessati a relazioni di ereditarietá derivate con `Newtype` o rule, né tantomeno a relazioni di equivalenza, dal momento che queste sono state giá tenute in considerazione al momento della creazione delle varie classi componenti lo schema.

[20] Siccome UNISQL/X richiede, per poter legare tramite relazione di ereditarietá due classi, che esse siano giá state definite, ecco spiegato il motivo per cui é necessario creare preliminarmente tutte le classi dello schema, prima di procedere alla loro descrizione.

te, il sopracitato rollback possa essere omesso, provvedendo cosí a copiare immediatamente l' oggetto all' interno della classe virtuale stessa. Il motivo per cui ció non viene fatto é da ricercarsi nella considerazione che, nel caso lo schema contenga una gerarchia di classi virtuali, devo popolare solo le classi piú specializzate tra tutte quelle la cui C.S. di appartenenza é soddisfatta dall' oggetto corrente. Per poter far ció si é perció scelto di tentare l' inserimento in tutte le classi virtuali dello schema, memorizzando in una struttura temporanea i nomi di tutte quelle classi che consentono tale inserimento (per le quali perció la C.S. di appartenenza é verificata). In seguito verranno di fatto popolate con l' oggetto corrente solo le classi piú specializzate tra quelle contenute nella sopracitata struttura.

La fase di popolazione delle classi virtuali opportune, successiva alla fase di verifica delle varie C.S. di appartenenza, viene gestita associando a ciascuna classe primitiva dello schema un' ulteriore coppia trigger-metodo, la quale:

- Analizza la struttura dati temporanea contenente l' elenco di tutte le classi virtuali la cui C.S. di appartenenza é soddisfatta dall' oggetto corrente.
- Determina quali sono le classi virtuali piú specializzate tra queste.
- Inserisce solo in queste classi l' oggetto corrente (tramite una semplice istruzione di **insert**).

Da notare é infine il fatto che, affinché tutto funzioni, é necessario che, all' interno di ogni classe primitiva, il trigger deputato alla fase di popolazione delle classi virtuali opportune abbia prioritá inferiore rispetto a tutti i trigger relativi alla verifica delle C.S. di appartenenza alle varie classi virtuali, in modo tale da essere eseguito solo dopo aver determinato il nome di tutte le classi virtuali che possono contenere l' oggetto corrente.

(f) Se invece l' elemento corrente di *ListaN* contiene informazioni circa il conseguente di una rule, allora si passa semplicemente al record successivo. Questo comportamento é dovuta alla necessitá di gestire la possibilitá che OLCD dá di definire separatamente l' antecedente e il conseguente della stessa rule. Per questo motivo si é scelto di realizzare un software che vada a caccia degli antecedenti all' interno di *ListaN*, trascurando in prima battuta i conseguenti, salvo poi ricercare ed analizzare questi ultimi subito dopo aver terminato l' analisi del relativo antecedente.

(g) Se invece sono in presenza di un antecedente di rule, viene chiamata la funzione "`analizza_rule`", la quale:

- Riceve come parametro sia il puntatore alla lista dei *sigma* relativa alla porzione di rule correntemente analizzata sia l' informazione (sotto forma di variabile booleana) se si tratta di un antecedente o di un conseguente.
  Tale informazione é fondamentale dal momento che consente di differenziare il comportamento della funzione, laddove necessario, a seconda di quale parte di una regola sto esaminando.
- In particolare se sono in presenza di un antecedente:
  i. Si determina la classe a cui la regola si riferisce. Per far ció basta semplicemente analizzare il primo record della lista dei sigma relativa all' elemento corrente di *ListaN*, che contiene appunto tale informazione.

  NOTA BENE: Non viene riconosciuta la possibilitá di realizzare rules che siano riferite alla **TOP Class**; in altre parole é sempre necessario che una regola sia associata ad una classe o a un tipo valore definito all' interno dello schema correntemente analizzato.
  Nel caso ció non accada, viene riportato in output, al momento dell' esecuzione di OCDL_Designer, un messaggio di "Warning", il quale avvisa della presenza di una regola riferita alla Top Class, lasciando poi la possibilitá all' utente di decidere se interrompere la scrittura del file *nomefile.c* e modificare lo schema del database, oppure proseguire ignorando tale rule (a livello di implementazione fisica in UNISQL).
  In particolare, se supponiamo che il nome della regola non legale sia `Rule1`, il messaggio sará:

```
Writing output file (Schema1.c):  INTERRUPTED
  (WARNING: Rule Rule1 not referred to a class:
    Press y to continue (Rule1 will be ignored).
    Press n to abort.)
```

ii. Prima di proseguire viene invocata la funzione "`controlla_coerenza_rule`", la quale analizza tutte le condizioni contenute nella rule corrente, e per ognuna di esse verifica che gli attributi ivi nominati siano effettivamente esistenti nello schema, all' interno delle opportune classi.
Tale controllo é reso necessario dal fatto che UNISQL richiede, al contrario di OLCD, che tutti gli attributi usati in una rule siano stati preventivamente creati all' interno dell' opportuna classe.
In altri termini, se supponiamo dato il seguente semplice schema:

```
prim Manager = ^ [ name :  string,
     salary :  integer ];
prim Branch = ^ [ name :  string,
     managed-by :  Manager ];

antev Rule1a = Branch & ^ [ name :
     vstring "Research" ];
consv Rule1c = Branch & ^ [ managed-by :
     ^ [ level :  range 10 15 ] ];
```

Poiché l' attributo *level* non fa parte della classe *Manager*, al momento dell' esecuzione di OCDL-Designer viene generato un messaggio di errore, il quale segnala l' errore dando all' utente la possibilitá di scegliere se:
– Abortire la scrittura del file *nomefile.c*, in modo tale da poter correggere la rule e rilanciare in seguito OCDL-Designer.
– Proseguire ignorando, in fase di creazione del database, la rule *Rule1*.

Facendo riferimento a questo caso specifico, il messaggio sará il seguente:

```
Writing output file (Schema1.c):  INTERRUPTED
  (WARNING: Rule Rule1 contains following
   unknown attributes:
     LEVEL : (in class MANAGER)
    Press y to continue (Rule1 will be ignored).
```



```
    Press n to abort.)
```

iii. A questo punto, se non sono state rilevate anomalie, viene creata la coppia trigger-metodo che realizza fisicamente la regola. In particolare il trigger ha il compito di provocare, in seguito ad ogni inserimento all' interno della classe a cui é riferito, l' esecuzione del metodo, il quale contiene il codice necessario per controllare che l' oggetto inserito soddisfi tutti i vincoli espressi nella rule.

iv. Si passa quindi alla scrittura del codice che consente, una volta compilato ed eseguito il file *nomefile.c*, di realizzare il body del metodo.
Tale body (come tutti quelli di cui si é parlato finora) é scritto in `Embedded ESQL/X` e viene messo nel file **<nomefile>_met.ec**. In particolare si comincia scrivendo la signature, la quale ha la caratteristica di essere priva di parametri.

v. A seguire viene chiamata una prima volta la funzione "`crea_declare_section`", passandole come parametro il puntatore alla lista dei *sigma* relativa all' antecedente della rule corrente. La funzione semplicemente analizza la struttura di tale lista, provvedendo a creare l' opportuna porzione di Declare Section[21].

vi. Successivamente, dopo aver rintracciato all' interno di *ListaN* il record relativo al conseguente della regola corrente, viene chiamata una seconda volta "`crea_declare_section`", passandole questa volta come parametro il puntatore alla lista dei *sigma* che a partire da tale record si snoda, cosí da poter ultimare la scrittura della Declare Section.

vii. A questo punto inizia la fase di scrittura del codice relativo al body corrente. In particolare per prima cosa viene settato a FALSE il flag che indica se l' oggetto che ha invocato il metodo di cui sto realizzando il body rispetta o meno la rule che il metodo stesso implementa[22].

[21] Ricordo che per Declare Section si intende quella parte del body di un metodo all' interno della quale sono definite tutte quelle variabili d' ambiente che saranno in seguito usate nel body stesso per interagire con il database

[22] Il codice del body é strutturato infatti come vedremo in modo da modificare il valore del flag solo nel caso l' oggetto inserito sia conforme alla rule corrente.

viii. Quindi viene chiamata la funzione "`scandisci_sigma`", in modo tale da scrivere nel file **<nomefile>_met.ec** il codice che realizza fisicamente i vincoli espressi all' interno dell' antecedente correntemente analizzato[23].

ix. Da ultimo la funzione "`analizza_rule`" scandisce *ListaN*, rintraccia il puntatore alla lista dei *sigma* che descrive il conseguente della regola che sto esaminando, quindi chiama ricorsivamente se stessa, passando come parametro tale puntatore, assieme all' informazione che si tratta di un conseguente.

- Se invece sono in presenza di un conseguente, la funzione "`analizza_rule`":
  i. Per prima cosa chiama "`scandisci_sigma`", in modo tale da scrivere sempre nel file **<nomefile>_met.ec** il codice relativo ai vincoli espressi nel conseguente.
  ii. A seguire chiude il body del metodo, avendo cura inoltre di inserire un controllo finale che consente di visualizzare il messaggio d' errore (giá visto in precedenza) previsto nel caso in cui l' oggetto corrente non soddisfi la regola di integritá che il body appena scritto realizza.

NOTA: La filosofia adottata per realizzare in UNISQL le rules é quella di accorpare antecedente e conseguente di ognuna di esse in un unico body, implementando le varie condizioni sotto forma di statement "**if**" e cicli "**for**" concatenati tra loro. Solo se tutte le condizioni espresse all' interno della rule risulteranno verificate sará possibile raggiungere il cuore del body, il quale non fa altro che settare a TRUE il flag che indica se l' oggetto che ha invocato il metodo rispetta o no la regola che il metodo stesso realizza.

(h) Analizziamo ora la funzione "`scandisci_sigma`", la quale come giá detto in precedenza realizza il cuore del body del metodo che implementa la rule correntemente analizzata.
In particolare valgono le seguenti considerazioni:

- Ogni rule in OLCD puó essere vista come un insieme di condizioni di vario tipo in "and" tra loro.
- In ogni rule puó essere individuata una parte antecedente e una conseguente.

[23] Per la struttura della funzione si veda pag. 87.

- Se l' oggetto a cui applico la rule viola una o piú condizioni della parte antecedente, la rule non viene considerata in relazione al tale oggetto.
- In caso contrario si procede all' esame delle condizioni contenute nel conseguente. Se anche una sola di queste é violata, la rule provoca il rifiuto dell' istanza corrente.
- La tipologia delle condizioni che posso trovare all' interno di una rule é la seguente[24]:
  i. Condizioni di appartenenza ad una classe.
  ii. Condizioni sul tipo di un attributo.
  iii. Condizioni sul valore di un attributo.
  iv. Condizioni su collezioni ed estensioni.

Vediamo a questo punto come "`scandisci_sigma`" traduce le varie condizioni sopraelencate:

Condizioni di appartenenza ad una classe: Per prima cosa si determina il nome della classe a cui la condizione di appartenenza si riferisce. Per far ció chiamo la funzione "`determina_classe_appartenenza`", la quale, analizzando la struttura della rule corrente, determina il percorso che, partendo dalla classe a cui la regola si riferisce, consente di risalire al nome cercato.
Successivamente viene chiamata la funzione "`attr_classe_corrente`", la quale mette a disposizione l' elenco degli attributi della classe a cui la condizione si riferisce.
Sfruttando poi tutte le informazioni acquisite, viene aggiunto al file **<nomefile>_met.ec** il codice che verificherá la condizione che sto realizzando.
Tale codice é composto da:
  i. Una istruzione di commit delle transazioni giá eseguite, necessaria per evitare che queste possano essere indebitamente cancellate dal successivo rollback.
  ii. Una istruzione di insert dell' oggetto, relativamente al quale viene verificata la rule corrente, nella classe a cui la condizione di appartenenza si riferisce.
  iii. Un rollback della transazione appena eseguita, dal momento che di essa non deve in ogni caso restare traccia.

[24] Per una descrizione piú completa ed esauriente dei costrutti sopra elencati si veda il paragrafo 2.3.5 a pag. 15.

iv. Da ultimo viene analizzato il valore della variabile di sistema **SQLCODE**, la quale conterá un valore indicativo del fatto che il sopracitato insert sia o meno andato a buon fine (e di conseguenza che la condizione di appartenenza sia verificata o meno).

Condizioni sul tipo di un attributo: Si tratta delle condizioni di appartenenza di un attributo ad un range di interi o float.

La gestione di questo caso viene fatta chiamando la funzione "`body_metodo_base`", la quale traduce la condizione sotto forma di una "**select**", fatta in modo da verificare che l' oggetto che deve soddisfare la rule corrente abbia l' attributo indicato dalla condizione stessa compreso nel range specificato.

Questo sistema consente di trattare con estrema facilitá il caso in cui la condizione sia riferita non ad un attributo della classe corrente, bensí ad un attributo di un' altra classe (legata da un percorso a quella corrente). In questo caso infatti basta semplicemente inserire nella clausola "**where**" della "**select**" non direttamente il nome dell' attributo ma la path expression che a quest' ultimo consente di pervenire.

Condizioni sul valore di un attributo: Vengono gestite sfruttando la funzione "`body_metodo_base`", in modo del tutto analogo a quanto visto sopra per le condizioni sul tipo di un attributo, eccezion fatta per il caso in cui il valore di confronto sia il return value di un' operazione.

In quest' ultimo caso infatti viene invocata la funzione "`analizza_operazione`", la quale:

- Determina l' OID dell' oggetto a cui si riferisce l' operazione (sfruttando l' istruzione API *db_get*)[25].
- Chiama la funzione "`analizza_parametri`", la quale ha il compito di determinare la lista dei parametri da passare all' operazione.
  In particolare:
  - Se il parametro é una costante, esso viene aggiunto semplicemente alla lista.
  - Se si tratta invece del valore di un attributo, esso viene rintracciato sempre sfruttando l' istruzione API *db_get* e memorizzato in un' opportuna *Host_variable*, che viene poi anche aggiunta alla lista.
- Si scrive quindi sul file **<nomefile>_met.ec** il codice che realizza la chiamata all' operazione opportuna (viene usata l' istruzione "**call**")[26], avendo cura di passare come insieme dei parametri la lista generata in precedenza.
  Viene inoltre prestata attenzione a porre l' eventuale return value all' interno di una *Host_variable*.
- A questo punto il comportamento é diverso a seconda che il valore di ritorno sia un valore singolo oppure un set (sequenza) di valori.
  - Nel primo caso semplicemente viene aggiunta al codice del body corrente una "**select**", la quale verifica se l' attributo indicato nella rule é uguale al return value ricavato dalla sopracitata chiamata[27].
  - Nel secondo caso invece la gestione si presenta un pó piú complessa, dal momento che UNISQL/X non consente di confrontare tra loro, all' interno della clausola "**where**" di una "**select**", due set (o sequenze) di valori, nel caso uno di essi (o entrambi) sia contenuto in una *Host_variable*.
    Per questo motivo é necessario memorizzare (sempre tramite l' istruzione *db_get*) anche il valore dell' attributo di tipo set in una variabile-ambiente, e quindi confrontare quest' ultima con quella ricavata come return value dall' operazione appena eseguita. Tale confronto consiste nello scorrere elemento per elemento i due set (sequenze), e quindi verificare se ogni elemento trovato in uno esiste anche nell' altro. E' chiaro che il codice che implementa tutto ció dovrá essere diverso a seconda che io abbia a che fare con due set o con due sequenze, dal momento che in quest' ultimo caso devo tener conto anche del numero di volte in cui ogni valore ricorre all' interno della sequenza.

[25] Tale passo é necessario dal momento che puó capitare che venga invocata un' operazione che é riferita ad una classe diversa da quella a cui la rule é legata.

[26] Si usa l' istruzione call dal momento che, come giá detto in precedenza, le operazioni definite sullo schema corrente, vengono tradotte fisicamente tramite metodi, e di conseguenza una chiamata ad una operazione corrisponde, dal punto di vista fisico, ad una chiamata a metodo)

[27] Anche in questo caso, se capita che l' attributo non appartenga alla classe a cui la rule si riferisce, basta utilizzare al posto del suo nome la path expression che rappresenta il percorso che a quell' attributo conduce.

Condizioni su collezioni ed estensioni: Si tratta dei tipici costrutti di **exists** e **forall** che si possono presentare all' interno di una rule.

In particolare per quanto riguarda il costrutto **exists**, esso viene gestito dalla funzione "`analizza_exists`", la quale:

- Determina la classe a cui l' exists si riferisce (attraverso la funzione "`exists_ref_class`").
- Determina il set di oggetti coinvolti nell' exists (usando la funzione *db_get*), memorizzandolo all' interno di una *Host_variable*. Tale set non é altro che il valore di un opportuno attributo che mappa nella classe di riferimento dell' exists.
- Viene definito a questo punto un cursore, il quale seleziona, all' interno della classe di riferimento dell'exists, tutti gli oggetti che hanno OID memorizzato nella variabile-ambiente sopracitata.
- Per ognuno degli oggetti selezionati poi vengono controllate una ad una le condizioni che si trovano innestate nell'exists, e solo nel caso in cui siano verificate tutte viene incrementato un opportuno contatore.
- Al termine dell' analisi di tutti gli oggetti selezionati dal cursore, solo se il contatore ha valore >0 significa che la condizione di exists é verificata.

Per quanto riguarda invece il il costrutto **forall**, esso viene gestito in modo del tutto analogo a quanto visto per l' exists, eccezion fatta per il fatto che il contatore utilizzato non viene comparato a zero ma ad un altro contatore, contenente il numero di istanze appartenenti alla classe di riferimento del forall[28].

In questo caso solo se i due contatori sono uguali la condizione é verificata.

4.2.3 DESCRIZIONE FUNZIONE "SCRIVI_CODA"

Scrive il codice C che corrisponde alla coda del file *nomefile.c*, la quale contiene le istruzioni sia per chiudere il database sia per fare il commit delle transazioni eseguite tramite le varie istruzioni API.

[28] Il valore di tale contatore é semplicemente ottenuto tramite una select che conta il numero di istanze presenti nella classe a cui il forall é riferito.

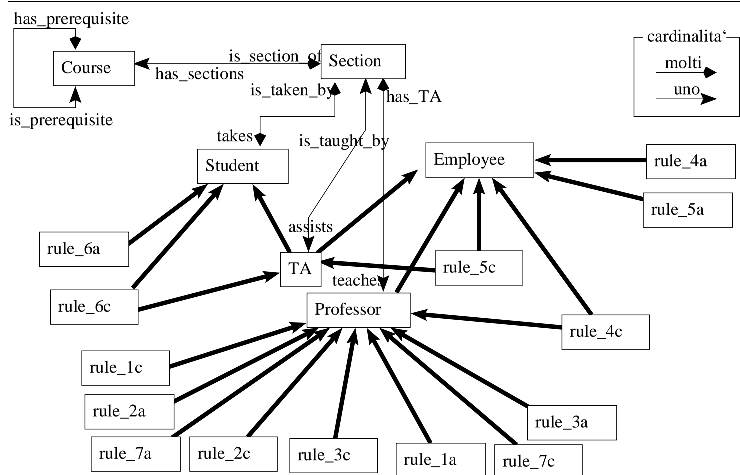

Figura 4.4: Schema dell'esempio con indicazione esplicita delle rules

4.3 IMPLEMENTAZIONE DI RULE: UN ESEMPIO

Alla luce di quanto affermato in precedenza, se consideriamo lo schema di figura 4.4, esso avrá in ODL esteso la seguente descrizione:

```
interface Course        ( extent courses keys name, number )
{       attribute string name_course;
        attribute unsigned short number;
        relationship list<Sections> has_sections
                inverse Sections::is_section_of
                {order_by Sections::number};
        relationship set<Course> has_prerequisites
                inverse Course::is_prerequisite_for;
        relationship set<Course> is_prerequisite_for
                inverse Course::has_prerequisites;
        boolean offer (in unsigned short semester)
                raises (already_offered);
        boolean drop  (in unsigned short semester)
                raises (not_offered);
        };
```

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



```
        void drop_course (in unsigned short Course)
                raises (not_registered_for_that_course);
        void assign_major (in unsigned short Department);
        short transfer(in unsigned short old_section,
                       in unsigned short new_section)
                raises (section_full, not_registered_in_section);
        };
//
// rules
//
rule rule_1 forall X in Professor:  X.rank = "Research"
   then                             X.annual_salary <= 40000 ;

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

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

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

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

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

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

Se memorizziamo in un file che supponiamo di chiamare **Schema1.odl** tale schema e lo diamo in ingresso al traduttore di ODB-Tools, otteniamo in output tra le altre cose il file **Schema1.sc**, il quale conterrá la descrizione dello schema stesso conforme alla sintassi OLCD. Tale file in particolare avrá la seguente struttura:

```
type Address =    [  college : string , room_number : string ] ;
prim Student =  ^ [ name_stud : string ,
                    student_id : integer ,
```

```
                    dorm_address : Address ,
                    takes : { Sections }  ] ;
prim TA =       Employee &
                Student &
                ^ [  assists : Sections  ] ;
prim Professor =  Employee &
                  ^ [ rank : string ,
                      teaches : { Sections }  ] ;
prim Employee = ^ [ name_emp : string ,
                    id   : integer ,
                    annual_salary : integer ] ;
prim Sections =  ^ [ number : string  ,
                    is_taught_by : Professor  ,
                    has_TA : TA  ,
                    is_section_of : Course  ,
                    is_taken_by : { Student }  ] ;
prim Course =  ^ [ name_course   : string ,
                   number : integer ,
                   has_sections : { Sections }  ,
                   has_prerequisites : { Course }  ,
                   is_prerequisite_for : { Course }  ] ;
antev rule_7a = Professor & ^ [ rank : vstring "Full" ] ;
consv rule_7c = Professor &
               ^ [ teaches :
                    !{ ^ [ is_section_of : ^ [ number : vstring "1" ] ,
                           is_section_of : Course ] } ] ;
antev rule_6a = Student & ^ [ student_id : range -inf 2000 ] ;
consv rule_6c = Student & TA ;
antev rule_5a = Employee & ^ [ annual_salary : range -inf 30000 ] ;
consv rule_5c = Employee & TA ;
antev rule_4a = Employee & ^ [ annual_salary : range 30000 +inf ] ;
consv rule_4c = Employee & Professor ;
antev rule_3a = Professor & ^ [ rank : vstring "Full" ] ;
consv rule_3c = Professor & ^ [ annual_salary : range 60000 +inf function
                 calcola ( in rank : string ) ] ;
antev rule_2a = Professor & ^ [ rank : vstring "Associate" ] ;
consv rule_2c = Professor & ^ [ annual_salary : range 40000 60000 function
                 calcola ( in rank : string ) ] ;
antev rule_1a = Professor & ^ [ rank : vstring "Research" ] ;
consv rule_1c = Professor & ^ [ annual_salary : range -inf 40000 function
                 calcola ( in rank : string ) ] ;
operation Professor = range 60000 +inf f calcola ( in param1 : string ) .
```

Andando ora ad eseguire su tale schema OCDL-Designer, ottengo in out-

put vari files, tra cui **Schema1.c**. Compilando ed eseguendo quest' ultimo viene generato automaticamente il file **Schema1_met.ec**, il quale contiene i body di tutti i metodi che implementano le varie rules definite sullo schema e che consentono perció di popolare il database esclusivamente con oggetti conformi alle regole stesse. Tale file sará cosí strutturato:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "dbi.h"

int flag=0;

void Professor_calcola(DB_OBJECT *o, DB_VALUE ret_val, DB_VALUE param1)
{
   EXEC SQLX BEGIN DECLARE SECTION;

   /* insert here local variable declarations */

   int ret_val_var;
   EXEC SQLX END DECLARE SECTION;

   /* Place here the method body */

   ret_val_var=DB_GET_INT(&ret_val);
   if ((ret_val_var>2147483647) || (ret_val_var<60000))
     {
       printf("Error: parameter 0 out of range\n");
       exit(-1);
     }

}


void Student_met_del_obj(DB_OBJECT *o, DB_VALUE ret_val)
{
   EXEC SQLX BEGIN DECLARE SECTION;

   DB_OBJECT *obj = o;

   EXEC SQLX END DECLARE SECTION;

   if (flag>0)
```

```
    {
      EXEC SQLX DELETE FROM ALL Student x WHERE x.identity=:obj;

      EXEC SQLX COMMIT WORK;
    }
  flag=0;

}


void Employee_met_del_obj(DB_OBJECT *o, DB_VALUE ret_val)
{
  EXEC SQLX BEGIN DECLARE SECTION;

  DB_OBJECT *obj = o;

  EXEC SQLX END DECLARE SECTION;

  if (flag>0)
    {
      EXEC SQLX DELETE FROM ALL Employee x WHERE x.identity=:obj;

      EXEC SQLX COMMIT WORK;
    }
  flag=0;

}


void Professor_rule_7_met_ins(DB_OBJECT *o, DB_VALUE ret_val)
{
  EXEC SQLX BEGIN DECLARE SECTION;

  DB_OBJECT *obj = o;
  DB_OBJECT *buf1,*buf_call;
  DB_VALUE buf1_coll,buf_call_coll;
  int i=0,n;
  int counter,error_var;
  int flag_buf;
  int rule_7c_exists1=0;
  DB_OBJECT *rule_7c_buf1;
  DB_VALUE rule_7c_val1;
  DB_COLLECTION *rule_7c_exists1_coll;
```



```
DB_VALUE rule_7c_exists1_val;

EXEC SQLX END DECLARE SECTION;

flag_buf=flag;

flag++;

EXEC SQLX SELECT count(*)
INTO :counter
FROM Professor x
WHERE x.identity=:obj and x.rank='Full';

if (counter==0)
  { flag--; }
else
{
buf1=obj;

error_var=db_get(buf1,"teaches",&rule_7c_exists1_val);
if (error_var>=0)
{

rule_7c_exists1_coll=DB_GET_SET(&rule_7c_exists1_val);

EXEC SQLX DECLARE rule_7c_curs1 CURSOR
FOR SELECT x.identity
    FROM Sections x
    WHERE x.is_section_of.number='1';

EXEC SQLX OPEN rule_7c_curs1;

for(;;)
{
EXEC SQLX FETCH rule_7c_curs1 INTO :rule_7c_buf1;

if (SQLCODE==100)
   break;

db_make_object(&rule_7c_val1,rule_7c_buf1);

if (db_col_ismember(rule_7c_exists1_coll,&rule_7c_val1)!=0)
{
```

```
error_var=db_get(rule_7c_buf1,"is_section_of",&buf1_coll);

if (error_var>=0)
{

buf1=DB_GET_OBJECT(&buf1_coll);

EXEC SQLX COMMIT WORK;

EXEC SQLX INSERT INTO Course(has_prerequisites,has_sections,
                        is_prerequisite_for,name_course,number)
  SELECT has_prerequisites,has_sections,is_prerequisite_for,
         name_course,number
  FROM Course x
  WHERE x.identity=:buf1;

if (SQLCODE!=0)
  {
    continue;
  }
else
  {
    EXEC SQLX ROLLBACK WORK;
  }

rule_7c_exists1++;
}
}

EXEC SQLX CLOSE rule_7c_curs1;

if (rule_7c_exists1>0)
{
flag--;
}
}
}
}
if (flag>flag_buf)
  {
    if (flag==1)
       printf("WARNING: object %d in class Professor
              rejected\n     (rule rule_7 violated)\n",obj);
```



```
       else
          printf("     (rule rule_7 violated)\n");

    }

}


void Student_rule_6_met_ins(DB_OBJECT *o, DB_VALUE ret_val)
{
   EXEC SQLX BEGIN DECLARE SECTION;

   DB_OBJECT *obj = o;
   DB_OBJECT *buf1,*buf_call;
   DB_VALUE buf1_coll,buf_call_coll;
   int i=0,n;
   int counter,error_var;
   int flag_buf;

   EXEC SQLX END DECLARE SECTION;

   flag_buf=flag;

   flag++;

   EXEC SQLX SELECT count(*)
   INTO :counter
   FROM Student x
   WHERE x.identity=:obj
   AND x.student_id BETWEEN -2147483647 and 2000;

   if (counter==0)
     { flag--; }
   else
   {
   buf1=obj;

   EXEC SQLX COMMIT WORK;

   EXEC SQLX INSERT INTO TA(dorm_address,name_stud,student_id,takes)
     SELECT dorm_address,name_stud,student_id,takes
     FROM Student x
     WHERE x.identity=:buf1;
```

```
   if (SQLCODE==0) {
   EXEC SQLX ROLLBACK WORK;

   flag--;
   }
   }
   if (flag>flag_buf)
     {
       if (flag==1)
          printf("WARNING: object %d in class Student
                 rejected\n     (rule rule_6 violated)\n",obj);
       else
          printf("     (rule rule_6 violated)\n");

     }

}


void Employee_rule_5_met_ins(DB_OBJECT *o, DB_VALUE ret_val)
{
   EXEC SQLX BEGIN DECLARE SECTION;

   DB_OBJECT *obj = o;
   DB_OBJECT *buf1,*buf_call;
   DB_VALUE buf1_coll,buf_call_coll;
   int i=0,n;
   int counter,error_var;
   int flag_buf;

   EXEC SQLX END DECLARE SECTION;

   flag_buf=flag;

   flag++;

   EXEC SQLX SELECT count(*)
   INTO :counter
   FROM Employee x
   WHERE x.identity=:obj
   AND x.annual_salary BETWEEN -2147483647 and 30000;
```



```
   if (counter==0)
     { flag--; }
   else
   {
   buf1=obj;

   EXEC SQLX COMMIT WORK;

   EXEC SQLX INSERT INTO TA(annual_salary,id,name_emp)
     SELECT annual_salary,id,name_emp
     FROM Employee x
     WHERE x.identity=:buf1;

   if (SQLCODE==0) {
   EXEC SQLX ROLLBACK WORK;

   flag--;
   }
   }
   if (flag>flag_buf)
     {
       if (flag==1)
          printf("WARNING: object %d in class Employee
                 rejected\n     (rule rule_5 violated)\n",obj);
       else
          printf("     (rule rule_5 violated)\n");

     }

}


void Employee_rule_4_met_ins(DB_OBJECT *o, DB_VALUE ret_val)
{
   EXEC SQLX BEGIN DECLARE SECTION;

   DB_OBJECT *obj = o;
   DB_OBJECT *buf1,*buf_call;
   DB_VALUE buf1_coll,buf_call_coll;
   int i=0,n;
   int counter,error_var;
   int flag_buf;
```

```
   EXEC SQLX END DECLARE SECTION;

   flag_buf=flag;

   flag++;

   EXEC SQLX SELECT count(*)
   INTO :counter
   FROM Employee x
   WHERE x.identity=:obj
   AND x.annual_salary BETWEEN 30000 and 2147483647;

   if (counter==0)
     { flag--; }
   else
   {
   buf1=obj;

   EXEC SQLX COMMIT WORK;

   EXEC SQLX INSERT INTO Professor(annual_salary,id,name_emp)
     SELECT annual_salary,id,name_emp
     FROM Employee x
     WHERE x.identity=:buf1;

   if (SQLCODE==0) {
   EXEC SQLX ROLLBACK WORK;

   flag--;
   }
   }
   if (flag>flag_buf)
     {
       if (flag==1)
          printf("WARNING: object %d in class Employee
                 rejected\n     (rule rule_4 violated)\n",obj);
       else
          printf("     (rule rule_4 violated)\n");

     }

}
```



```
void Professor_rule_3_met_ins(DB_OBJECT *o, DB_VALUE ret_val)
{
   EXEC SQLX BEGIN DECLARE SECTION;

   DB_OBJECT *obj = o;
   DB_OBJECT *buf1,*buf_call;
   DB_VALUE buf1_coll,buf_call_coll;
   int i=0,n;
   int counter,error_var;
   int flag_buf;
   int calcola_in_Professor_ret_val;
   char calcola_in_Professor_var1[256];

   EXEC SQLX END DECLARE SECTION;

   flag_buf=flag;

   flag++;

   EXEC SQLX SELECT count(*)
   INTO :counter
   FROM Professor x
   WHERE x.identity=:obj and x.rank='Full';

   if (counter==0)
     { flag--; }
   else
   {
   buf_call=obj;

   error_var=db_get(buf_call,"rank",&calcola_in_Professor_var1);

   if (error_var>=0)
   {

   EXEC SQLX CALL calcola(:calcola_in_Professor_var1)
   ON :buf_call INTO :calcola_in_Professor_ret_val;

   EXEC SQLX SELECT COUNT(*)
   INTO :counter
   FROM Professor x
   WHERE x.annual_salary=:calcola_in_Professor_ret_val
```

```
   AND x.identity=:buf_call;

   if (counter>0)
   {
   flag--;
   }
   }
   }
   if (flag>flag_buf)
     {
       if (flag==1)
          printf("WARNING: object %d in class Professor
                 rejected\n     (rule rule_3 violated)\n",obj);
       else
          printf("     (rule rule_3 violated)\n");

     }

}


void Professor_rule_2_met_ins(DB_OBJECT *o, DB_VALUE ret_val)
{
   EXEC SQLX BEGIN DECLARE SECTION;

   DB_OBJECT *obj = o;
   DB_OBJECT *buf1,*buf_call;
   DB_VALUE buf1_coll,buf_call_coll;
   int i=0,n;
   int counter,error_var;
   int flag_buf;
   int calcola_in_Professor_ret_val;
   char calcola_in_Professor_var1[256];

   EXEC SQLX END DECLARE SECTION;

   flag_buf=flag;

   flag++;

   EXEC SQLX SELECT count(*)
   INTO :counter
   FROM Professor x
```



```
   WHERE x.identity=:obj and x.rank='Associate';

   if (counter==0)
     { flag--; }
   else
   {
   buf_call=obj;

   error_var=db_get(buf_call,"rank",&calcola_in_Professor_var1);

   if (error_var>=0)
   {

   EXEC SQLX CALL calcola(:calcola_in_Professor_var1)
   ON :buf_call INTO :calcola_in_Professor_ret_val;

   EXEC SQLX SELECT COUNT(*)
   INTO :counter
   FROM Professor x
   WHERE x.annual_salary=:calcola_in_Professor_ret_val
   AND x.identity=:buf_call;

   if (counter>0)
   {
   flag--;
   }
   }
   }
   if (flag>flag_buf)
     {
       if (flag==1)
          printf("WARNING: object %d in class Professor
                 rejected\n     (rule rule_2 violated)\n",obj);
       else
          printf("     (rule rule_2 violated)\n");

     }

}


void Professor_rule_1_met_ins(DB_OBJECT *o, DB_VALUE ret_val)
{
```

```
EXEC SQLX BEGIN DECLARE SECTION;

DB_OBJECT *obj = o;
DB_OBJECT *buf1,*buf_call;
DB_VALUE buf1_coll,buf_call_coll;
int i=0,n;
int counter,error_var;
int flag_buf;
int calcola_in_Professor_ret_val;
char calcola_in_Professor_var1[256];

EXEC SQLX END DECLARE SECTION;

flag_buf=flag;

flag++;

EXEC SQLX SELECT count(*)
INTO :counter
FROM Professor x
WHERE x.identity=:obj and x.rank='Research';

if (counter==0)
  { flag--; }
else
{
buf_call=obj;

error_var=db_get(buf_call,"rank",&calcola_in_Professor_var1);

if (error_var>=0)
{

EXEC SQLX CALL calcola(:calcola_in_Professor_var1)
ON :buf_call INTO :calcola_in_Professor_ret_val;

EXEC SQLX SELECT COUNT(*)
INTO :counter
FROM Professor x
WHERE x.annual_salary=:calcola_in_Professor_ret_val
AND x.identity=:buf_call;

if (counter>0)
```



```
   {
   flag--;
   }
   }
   }
   if (flag>flag_buf)
     {
       if (flag==1)
          printf("WARNING: object %d in class Professor
                 rejected\n     (rule rule_1 violated)\n",obj);
       else
          printf("     (rule rule_1 violated)\n");

     }

}
```

Appendice A

CONFRONTO TRA LE RELEASE 1.1 E 2.0 DI ODMG93

Obiettivo di questo capitolo é quello di confrontare le release 1.1 e 2.0 di ODMG93, allo scopo di evidenziare in cosa la versione 2.0 si differenzia dalla precedente[1].
L' analisi verrá condotta seguendo l' ordine in cui i componenti sono analizzati all' interno dello standard, cioé:

- Object Model
- ODL (*Object Definition Language*)
- OQL (*Object Query Language*)

A.1 OBJECT MODEL

Per quanto riguarda l' Object Model, si possono notare le seguenti differenze, raggruppate per argomento:

A.1.1 TIPI E CLASSI: INTERFACCIA E IMPLEMENTAZIONE

Si possono evidenziare due differenze fondamentali:

[1] Per maggiori informazioni circa le due versioni dello standard si veda rispettivamente [ODMG1.1] per la release 1.1 e [ODMG2.0] per la release 2.0.

1. **Release 1.1** Un tipo é caratterizzato da un' *interfaccia* ed una o piú *implementazioni*. L' interfaccia in particolare definisce in modo astratto le proprietá del tipo e le operazioni che possono essere fatte su di esso, un' implementazione invece definisce le strutture dati (in termini di quali istanze del tipo sono fisicamente rappresentate) e i metodi che operano su di esse.
   Dal punto di vista sintattico la definizione di un tipo é affidata al costrutto *interface*.

   Release 2.0 Un tipo é dotato di una *specifica* e una o piú *implementazioni*. In particolare quest' ultimo concetto é lo stesso visto al punto precedente, mentre quello di specifica é del tutto equivalente al precedente concetto di interfaccia; il cambiamento di denominazione é giustificato dal fatto che in questa nuova versione dello standard per interfaccia si intende solo una parte del concetto di specifica.
   Infatti la specifica di un tipo si divide in:

   literal: definisce solo lo stato del tipo, vale a dire l' insieme delle proprietá ad esso relative.

   interface: definisce solo il comportamento del tipo, cioé l' insieme delle operazioni eseguibili su di esso.

   class: definisce sia lo stato, sia il comportamento di un tipo.

   Proprio in virtú di questa distinzione, la definizione sintattica di un tipo puó essere fatta ancora attraverso il costrutto *interface*, nel caso io voglia descrivere solo il suo comportamento, oppure si puó usare il costrutto *class*, nel caso in cui invece si voglia definire, oltre al comportamento, anche il suo stato.
   Chiaramente, da quanto detto risulta chiaro che una *class* é un tipo direttamente istanziabile (cioé le cui istanze possono essere create dal programmatore), un' *interface* invece no.

2. **Release 1.1** E' prevista la possibilitá di definire una sola relazione di ereditarietá tra tipi-oggetto (indicata sintatticamente tramite " : "), che comporta l' acquisizione, da parte del sub-tipo, di tutte le proprietá e operazioni relative a (uno o piú) super-tipi.

   Release 2.0 Si distinguono due tipi di relazione di ereditarietá:

   ereditarietá di comportamento: indicata sintatticamente tramite i due punti (" : "), puó riguardare solo tipi-oggetto e puó essere singola o multipla.
   Implica l' ereditá del solo comportamento, cioé delle sole

operazioni fattibili sul tipo considerato.
Proprio per questo:

- Una *interface* o una *class* possono ereditare il comportamento di una o piú *interface*.
- Una *class* o una *interface* non possono ereditare il comportamento da una *class*.

ereditarietá di stato: indicata sintatticamente tramite la parola chiave *EXTENDS*, puó essere solo singola e si applica anche essa solo a tipi-oggetto.
Si differenzia dalla precedente perché si tratta di una relazione tra **classi** per mezzo di cui la classe subordinata eredita tutte le proprietá e le operazioni della sua superclasse.

Il motivo della mancanza, all' interno della vecchia versione dello standard, di questa distinzione tra i due tipi di ereditarietá é facilmente intuibile se si considera che tale distinzione é indotta dalla separazione tra i concetti di classe e interfaccia.

A.1.2 OGGETTI

1. Sappiamo che tutti gli oggetti di un database si possono considerare come istanze di un tipo *Object*. ODMG 2.0 amplia l' insieme delle operazioni eseguibili sulle istanze di questo tipo in seguito alla loro creazione da parte dell' utente:

```
interface Object {
      enum         Lock-Type{read, write, upgrade};
      exception    LockNotGranted;
      void         lock(in Lock-type mode) raises(LockNotGranted);
      boolean      try-lock(in Lock-Type mode);
      boolean      same-as(in Object anObject);
      Object       copy();
      void         delete();
};
```

   Di queste solo quelle con i nomi in corsivo erano presenti nella release 1.1.
   In particolare l' operazione di *lock* esegue il lock di un oggetto, *try-lock* fa la stessa cosa ritornando un valore booleano che indica se l' operazione é andata o meno a buon fine.

   Copy esegue invece la copia di un oggetto, e il risultato é un oggetto con stesso stato ma identitá diversa da quello di cui ho fatto la copia.

   Inoltre la versione 2.0 dello standard definisce un nuovo tipo (*ObjectFactory*), usato per creare nuove istanze del tipo *object*.

```
interface ObjectFactory {
      Object      new();
};
```

2. Se poi considero tutta la gerarchia dei tipi oggetto, ogni istanza di uno dei sottotipi del tipo "Object" puó essere creata tramite l' operazione **new** definita in un' interfaccia *factory* (a sua volta sottotipo dell' interfaccia *ObjectFactory*) relativa al tipo dell' istanza da creare.

3. Per quanto riguarda la gerarchia dei sottotipi del tipo object non vi sono sostanziali differenze nel passaggio dalla vecchia alla nuova release, eccezion fatta per l' introduzione in quest' ultima dei tipi strutturati

   - Date
   - Interval
   - Time
   - Timestamp

   che prima erano definiti solo all' interno della gerarchia dei tipi letterali.

4. Relativamente al tempo di vita degli oggetti con cui popolo il mio database, ODMG 2.0 sottolinea (cosa che la vecchia release non faceva) come esso sia indipendente dal tipo; in altre parole un tipo puó avere alcune istanze persistenti, altre volatili.
   Questa proprietá consente di manipolare oggetti, persistenti e non, usando le medesime operazioni.

A.1.3 LITERALS

I letterali come sappiamo sono oggetti privi di identificatore e costituiscono nel loro insieme un tipo chiamato appunto tipo "literal".

Release 1.1: Il tipo literal viene organizzato nella seguente gerarchia di sottotipi:

- ◇ *literal-type*
  - ◇ *Atomic Literal*
    - ◇ Integer
    - ◇ Float
    - ◇ Character
    - ◇ Boolean
  - ◇ *Structured Literal*
    - ◇ *Immutable-Collection Literal*
      - ◇ Set<>
      - ◇ Bag<>
      - ◇ List<>
      - ◇ Array<>
      - ◇ Enumeration
    - ◇ *Immutable-Structure Literal*
      - ◇ Date
      - ◇ Time
      - ◇ Timestamp
      - ◇ Interval
      - ◇ Structure<>

Release 2.0: La gerarchia di sottotipi si presenta cosí strutturata:

- ◇ *literal-type*
  - ◇ *Atomic Literal*
    - ◇ long
    - ◇ short
    - ◇ unsigned long
    - ◇ unsigned short
    - ◇ float
    - ◇ double
    - ◇ boolean
    - ◇ octet
    - ◇ char
    - ◇ string
    - ◇ enum<>
  - ◇ *Collection Literal*
    - ◇ Set<>
    - ◇ Bag<>
    - ◇ List<>
    - ◇ Array<>
    - ◇ Dictionary<>

  - ◇ *Structured Literal*
    - ◇ Date
    - ◇ Time
    - ◇ Timestamp
    - ◇ Interval
    - ◇ Structure<> (consente all' utente di definire una struttura con le caratteristiche desiderate).

Come si puó facilmente notare, la nuova versione dello standard:

- amplia notevolmente il parco tipi catalogati come atomici.
- considera il tipo enumerazione come atomico e non piú come un "collection-literal".
- definisce il tipo *dictionary*<> tra i "collection-literals".
  A tal proposito ricordiamo che col termine *dictionary* si intende una sequenza non ordinata di coppie chiave-valore priva di chiavi duplicate.

A.1.4 METADATI

I metadati sono informazioni descrittive circa gli oggetti che popolano un database e ne definiscono lo schema.
Sono usati dall' ODBMS per definire la struttura del database e guidare a run-time gli accessi ad esso, e sono memorizzati in un *ODL schema Repository*, che risulta accessibile usando le stesse operazioni che si usano per manipolare gli oggetti e i tipi definiti dall' utente.
La Release 2.0 si preoccupa di definire nel dettaglio la struttura di questo deposito di schemi, piú di quanto non faccia la versione precedente dello standard. Nostro intento sará ora quello di esaminare come ODMG gestisce questi metadati.

L' ODL SCHEMA REPOSITORY

Come giá detto si tratta del "deposito" all' interno del quale sono memorizzati i metadati, e di conseguenza la struttura del database.
Internamente é costituito da un insieme di *interface*, descritte in ODL, che possono essere di tre tipi principali:

1. **MetaObject**
2. **Specifier**

3. **Operand**

Prima di scendere nel dettaglio della classificazione appena fatta, vogliamo tuttavia enucleare un paio di concetti preliminari:

- Tutti i *MetaObject* che verranno in seguito definiti sono raggruppati nel modulo

```
module ODLMetaObjects {
        //contiene tutte le interfacce che definiscono i meta-oggetti
};
```

- Viene aggiunta, sempre all' interno dello *schema repository*, una interfaccia, chiamata **Scope**, la quale definisce una gerarchia di nomi per i vari *meta-Object* all' interno del deposito:

```
interface Scope {
    exception    DuplicateName{};
    void         bind(in string name, in MetaObject value)
                     raises(DuplicateName);
    MetaObject resolve(in string name);
    MetaObject un-bind(in string name);
    };
```

Come si puó facilmente notare, vengono fornite tre operazioni:

bind : aggiunge alla gerarchia di nomi un meta-oggetto con un nome specificato.

resolve : trova all' interno del deposito il meta-oggetto con il path specificato come parametro.

un-bind : elimina, relativamente al meta-oggetto il cui nome é passato come parametro, il legame tra l' oggetto e il nome.

META OBJECT

La prima delle tre interfacce principali che posso avere all' interno dello *schema repository* é quella chiamata **MetaObject**, dotata della seguente sintassi:


```
interface MetaObject {
    attribute    string   name;
    attribute    string   comment;
    relationship DefiningScope  definedln
                     inverse DefiningScope::defines;
    };
```

Tutti i Meta-oggetti che definiró in seguito hanno un nome e un commento; inoltre sono in relazione con altri meta-oggetti, grazie al legame con l' interfaccia **DefiningScope**, a sua volta cosí definita:

```
enum  PrimitiveKind {pk-boolean,pk-char,pk-short,pk-ushort,pk-long,
                     pk-ulong,pk-float,pk-double,pk-octet,pk-string,
                     pk-void,pk-any};
enum  CollectionKind {ck-list,ck-array,ck-bag,ck-set,ck-dictionary};

interface DefiningScope : Scope {
    relationship    list<MetaObject>defines
                        inverse MetaObject::definedln;
    exception       InvalidType{string reason;};
    exception       InvalidExpression{string reason;};
    exception       CannotRemove{string reason;};
    PrimitiveType   create-primitive-type(in PrimitiveKind kind);
    Collection      create-collection-type(in CollectionKind kind
                        in Operand maxSize, in Type subType);
    Operand         create-operand(in string expression)
                        raises(InvalidExpression);
    Member          create-member(in string memberName,
                        in Type memberType);
    UnionCase       create-case(in string caseName,in Type caseType,
                        in list<Operand> caseLabels)
                        raises(DuplicateName, InvalidType);
    Constant        add-constant(in string name,in Operand value)
                        raises(DuplicateName);
    TypeDefinition  add-typedef(in string name,in Type alias)
                        raises(DuplicateName);
    Enumeration     add-enumeration(in string name,
                        in list<string> elementNames)
                        raises(DuplicateName, InvalidType);
    Structure       add-structure(in string name,
                        in list<Member> fields)
```

```
                raises(DuplicateName, InvalidType);
Union       add-union(in string name,in Type switchType,
                in list<UnionCase> cases)
                raises(DuplicateName, InvalidType);
Exception   add-exception(in string name,in Structure result)
                raises(DuplicateName);
void        remove-constant(in Constant object)
                raises(CannotRemove);
void        remove-typedef(in TypeDefinition object)
                raises(CannotRemove);
void        remove-enumeration(in Enumeration object)
                raises(CannotRemove);
void        remove-structure(in Structure object)
                raises(CannotRemove);
void        remove-union(in Union object)
                raises(CannotRemove);
void        remove-exception(in Exception object)
                raises(CannotRemove);
};
```

Tale interfaccia definisce istanze che contengono, attraverso la relazione *defines*, altri meta-oggetti; inoltre definisce operazioni per creare, aggiungere o eliminare meta-oggetti da una sua istanza.

Esistono vari tipi di meta-oggetto:

1. **MODULI :**I moduli e il deposito di schemi stesso (che é un modulo specializzato), sono *DefiningScopes* che definiscono operazioni per creare o rimuovere al loro interno moduli e interfacce:

```
interface  Module : MetaObject,DefiningScope {
    Module      add-module(in string name)
                    raises(DuplicateName);
    Interface   add-interface(in string name,
                    in list<Interface> inherits)
                    raises(DuplicateName);
    void        remove-module(in Module object)
                    raises(CannotRemove);
    void        remove-interface(in Interface object)
                    raises(CannotRemove);
};
```



```
interface Repository : Module {};
```

2. **OPERAZIONI :**Si tratta di meta-oggetti che mantengono una lista di parametri (attraverso la relazione *signature*), definiscono un tipo di ritorno (attraverso la relazione *result*), e sono in grado di trattare eccezioni (attraverso la relazione *exception*):

```
interface ScopedMetaObject : MetaObject,Scope {};

interface  Operation : ScopedMetaObject {
    relationship   list<Parameter>   signature
                       inverse Parameter::operation;
    relationship   Type              result
                       inverse Type::operations;
    relationship   list<Exception>   exceptions
                       inverse Exception::operations;
};
```

L' interfaccia **ScopedMetaObject** viene introdotta per racchiudere e consolidare le operazioni definite in *Scope* e *MetaObject*.

3. **ECCEZIONI :**Come visto al punto precedente, le operazioni possono trattare eccezioni, attraverso un' opportuna relazione che ha nome *operations* o *exceptions*, a seconda del verso in cui tale relazione é percorsa. Le eccezioni fanno inoltre riferimento (tramite la relazione *result*) a una *Structure* che verrá esaminata in seguito e che ha il compito sia di definire i risultati dell' eccezione sia di tener traccia di tutte le operazioni che ad essa si riferiscono.

```
interface  Exception : MetaObject {
    relationship   Structure        result
                       inverse Structure::exceptionResult;
    relationship   set<Operation> operations
                       inverse Operation::exceptions;
};
```

4. **COSTANTI :**Viene definita, per ogni costante, una relazione **hasValue** che associa staticamente un valore a un nome dello "schema repository" (che sará appunto il nome della costante). Tale valore é definito tramite la sottoclasse *Operand* e puó essere, come vedremo in seguito, un letterale, un riferimento ad un' altra costante o il valore di una espressione costante.

```
interface  Constant : MetaObject {
      relationship    Operand                hasValue
                          inverse Operand::ValueOf;
      relationship    Type                   type
                          inverse Type::constants;
      relationship    set<ConstOperand>  referencedBy
                          inverse ConstOperand::references;
      relationship    Enumeration            enumeration
                          inverse Enumeration::elements;
      any             value();
      };
```

Da questa sintassi risulta chiaro che per ogni costante:

- Viene definito, oltre al valore, il tipo, attraverso la relazione *type*.
- Viene mantenuta traccia, attraverso *referencedBy*, delle altre costanti dello stesso tipo.
- L' operazione *value* consente in ogni istante di conoscere il valore della costante.

5. **PROPRIETA' :**Si tratta di una superclasse delle interfacce *Attribute* e *Relationship*, che a loro volta definiscono lo stato dei vari oggetti presenti nel database.

```
interface  Property : MetaObject {
      relationship    Type    type
                          inverse Type::properties;
      };

interface  Attribute : Property {
      attribute      boolean    isReadOnly;
      };

enum Cardinality {c1-1,c1-N,cN-1,cN-M};

interface  Relationship : Property {
      exception      integrityError{};
      relationship   Relationship    traversal
                         inverse Relationship::traversal;
      Cardinality    getCardinality();
      };
```


NOTE:
- Ad ogni proprietá é associato un tipo (attraverso la relazione *type*).
- Un attributo puó essere definito come read-only o meno.
- Per ogni relazione vengono definite:
  (a) un' operazione *getCardinality()* che ne restituisce la cardinalitá.
  (b) una relazione ricorsiva con la stessa interfaccia *Relationship*, necessaria per rappresentare ognuna delle due direzioni di attraversamento.

6. **TIPI :**L' interfaccia **Type** é usata per rappresentare informazioni circa i tipi di dato.
Al suo interno viene definito un insieme di relazioni con tutte le altre interfacce dello "schema repository" che ad esso si riferiscono. Tali relazioni da un lato consentono una semplice amministrazione dei tipi interni al deposito, dall' altro costituiscono un valido aiuto per poter assicurare l' integritá referenziale del deposito stesso.

```
interface  Type : MetaObject {
      relationship    set<Collection>        collections
                          inverse Collection::subtype;
      relationship    set<Specifier>         specifiers
                          inverse Specifier::type;
      relationship    set<Union>             unions
                          inverse Union::switchType;
      relationship    set<Operation>         operations
                          inverse Operation::result;
      relationship    set<Property>          properties
                          inverse Property::type;
      relationship    set<Constant>          constants
                          inverse Constant::type;
      relationship    set<TypeDefinition>  TypeDefs
                          inverse TypeDefinition::alias;
      };
```

Viene inoltre definita una sottoclasse di *Type*, chiamata **PrimitiveKind**, la quale dá informazioni circa i tipi primitivi, classificandoli tramite l' attributo *kind*:

```
interface  PrimitiveType : Type {
      attribute      PrimitiveKind    kind;
      };
```

Infine la sottoclasse **TypeDefinition** aggiunge, per ogni tipo contenuto nel deposito, l' informazione circa gli alias con cui esso é indicato:

```
interface  TypeDefinition : Type {
      relationship    Type    alias
                          inverse Type::TypeDefs;
      };
```

7. **INTERFACCE :**Si tratta di una sottoclasse di *Type* e di *DefinedScope* e rappresenta il tipo piú importante all' interno del deposito.
   Una "interface" come sappiamo definisce il comportamento di un insieme di oggetti del database; per tal motivo l' interfaccia **interface** contiene operazioni per creare e rimuovere attributi, relazioni e operazioni che si trovano al suo interno, in aggiunta alle operazioni ereditate da *DefinedScope*.

```
interface  Interface : Type,DefiningScope {
      struct ParameterSpec {
         string      paramName;
         Direction paramMode;
         Type        paramType;};
      relationship  set<Inheritance>        inherits
                        inverse Inheritance::derivesFrom;
      relationship  set<Inheritance>        derives
                        inverse Inheritance::inheritsTo;
      exception     BadParameter{string reason;};
      exception     BadRelation{string reason;};
      Attribute     add-attribute(in string attrName,
                        in Type attrType)
                        raises(DuplicateName);
      Relationship add-relationship(in string relName,
                        in Type relType,
                        in Relationship relTraversal)
                        raises(DuplicateName,BadRelationship);
      Operation     add-operation(in string opName,
                        in Type opResult,
```



```
                        in list<ParameterSpec> opParams,
                        in list<Exception> opRaises)
                        raises(DuplicateName,BadParameter);
      void          remove-attribute(in Attribute object)
                        raises(CannotRemove);
      void          remove-relationship(in Relationship object)
                        raises(CannotRemove);
      void          remove-operation(in Operation object)
                        raises(CannotRemove);
      };
```

Inoltre i meta-oggetti di tipo *interface* sono collegati all'interfaccia **Inheritance** (a formare un grafo ad ereditarietá multipla) tramite le due relazioni *inherits* e *derives*.

```
interface  Inheritance {
      relationship    Interface        derivesFrom
                          inverse Interface::inherits;
      relationship    Interface        inheritsTo
                          inverse Interface::derives;
      };
```

Questa interfaccia puó contenere vari meta-oggetti, eccetto Moduli e Interfacce.

8. **CLASSI :**Sono considerate sottotipi dei meta-oggetti visti al punto precedente, a cui aggiungono le proprietá necessarie per definire lo stato degli oggetti memorizzati nel database.

```
interface  Class : Interface {
      attribute    list<string>    extents;
      attribute    list<string>    keys;
      relationship Class           extender
                       inverse Class::extensions;
      relationship set<Class>      extensions
                       inverse Class::extender;
      };
```

NOTE: Dalla sintassi sopra riportata si evince che:

- E' possibile definire sulle istanze di una qualunque classe del database una o piú chiavi (grazie all' attributo *keys*).

- L' attributo *extents* consente di riferirsi all' insieme delle istanze della classe relativa.
- Le classi sono unite tra loro a formare una gerarchia di ereditarietá singola tramite le relazioni *extender* ed *extension*. In questo modo una classe puó ereditare stato e comportamento da una sua superclasse.

9. **COLLEZIONI :**Si tratta di tipi che raggruppano un numero variabile di elementi di uno stesso sottotipo.
L' interfaccia **Collection** fornisce informazioni circa le collezioni presenti nel database.

```
interface Collection : Type {
    attribute       CollectionKind   kind;
    relationship    Operand          maxSize
                        inverse Operand::sizeOf;
    relationship    Type             subtype
                        inverse Type::collections;
    boolean         isOrdered();
    unsigned long   bound();
    };
```

NOTE:
- La relazione *maxSize* permette di specificare la dimensione massima della collezione.
- L' operazione *isOrdered* restituisce un valore booleano che indica se la collezione é ordinata o meno.

10. **TIPI COSTRUITI :**Si tratta di tipi i cui elementi a loro volta fanno riferimento ad altri tipi; in particolare distinguiamo:

- Tipo Enumerazione
- Tipo Struttura
- Tipo Union

Viene definita un' interfaccia **ScopedType**, che ha il solo compito di consolidare i meccanismi che stanno alla base dei tipi costruiti; quindi vengono introdotte tre sue sottoclassi, **Enumeration**, **Structure** e **Union**, che forniscono informazioni circa gli omonimi costrutti presenti nel database:

```
interface ScopedType : Scope,Type {};
```



```
interface Enumeration : ScopedType {
    relationship   list<Constant>    elements
                       inverse Constant::enumeration;
    };

interface Structure : ScopedType {
    relationship   list<Member>      fields
                       inverse Member::structure-type;
    relationship   Exception         exceptionResult
                       inverse Exception::result;
    };

interface Union : ScopedType {
    relationship   list<UnionCase>   cases
                       inverse UnionCase::union-type;
    relationship   Type              switchType
                       inverse Type::unions;
    };
```

NOTE:
- Un' istanza di *Enumeration* contiene una lista di *Constant* (attraverso la relazione *elements*).
- Un' istanza di *Structure* contiene una lista di *Member* (attraverso la relazione *fields*).
- Un' istanza di *Union* contiene una lista di *UnionCase* (attraverso la relazione *cases*).

SPECIFIER

Sono usati per assegnare un nome a un tipo in determinati contesti. E' prevista l' esistenza di un' interfaccia **Specifier**, che ha lo scopo di consolidare questo concetto e di fungere da supertipo per **Member**, **UnionCase** e **Parameter**, che sono referenziati rispettivamente dai precedentemente visti *Structure*, *Union* e *Operation*.

```
interface Specifier {
    attribute      string   name;
    relationship   Type     type
                       inverse Type::specifiers;
    };
```

```
interface  Member : Specifier {
      relationship   Structure        structure-type
                          inverse Structure::fields;
      };

interface  UnionCase : Specifier {
      relationship   Union            union-type
                          inverse Union::cases;
      relationship   list<Operand>caseLabels
                          inverse Operand::caseIn;
      };

enum Direction {mode-in,mode-out,mode-inout};

interface  Parameter : Specifier {
      attribute      Direction        parameterMode;
      relationship   Operation        operation
                          inverse Operation::signature;
      };
```

OPERAND

Gli operandi rappresentano il tipo base di tutti i valori costanti contenuti nello "schema repository". Viene fornita una interfaccia **Operand**, che contiene da un lato relazioni con *Constant*, *Collection*, *UnionCases* e *Expression*, dall' altro un' operazione *value* che ritorna il valore dell' operando.

```
interface  Operand {
      relationship   Expression       operandIn
                          inverse Expression::hasOperands;
      relationship   Constant         ValueOf
                          inverse Constant::hasValue;
      relationship   Collection       sizeOf
                          inverse Collection::maxSize;
      relationship   UnionCase        caseIn
                          inverse UnionCase::caseLabels;
      any            value();
      };
```

Vengono poi definite due ulteriori interfacce:


```
interface  Literal : Operand {
      attribute   any   literalValue;
      };

interface  ConstOperand : Operand {
      relationship   Constant        references
                          inverse Constant::referencedBy;
      };
```

NOTE:
- *Literal* indica che ogni literal é caratterizzato da un valore, *literalValue*
- *ConstOperand* indica come il valore dell' operando sia determinato non direttamente ma attraverso il riferimento ad un' altra costante(tramite la relazione *references*).

Concludiamo analizzando le espressioni, vale a dire insiemi di uno o piú operandi con un operatore associato.
Viene fornita l' interfaccia **Expression**, che indica come ogni espressione sia dotata di un operatore (definito come stringa) e una lista di operandi.

```
interface  Expression : Operand {
      attribute      string          operator;
      relationship   list<Operand>hasOperands
                          inverse Operand::operandIn;
      };
```

NOTA BENE : Si puó notare come la trattazione dell' argomento dei Meta-dati all' interno del manuale ODMG 2.0 sia viziato da una contraddizione di fondo; infatti vengono usati costrutti *interface*, nonostante al loro interno non vengano definiti solo metodi, ma anche attributi e relazioni.

STRUTTURA DELLO SCHEMA REPOSITORY

In figura A.1 é riportato il grafo che mostra la gerarchia delle varie metaclassi che definiscono la struttura dello schema di un database.

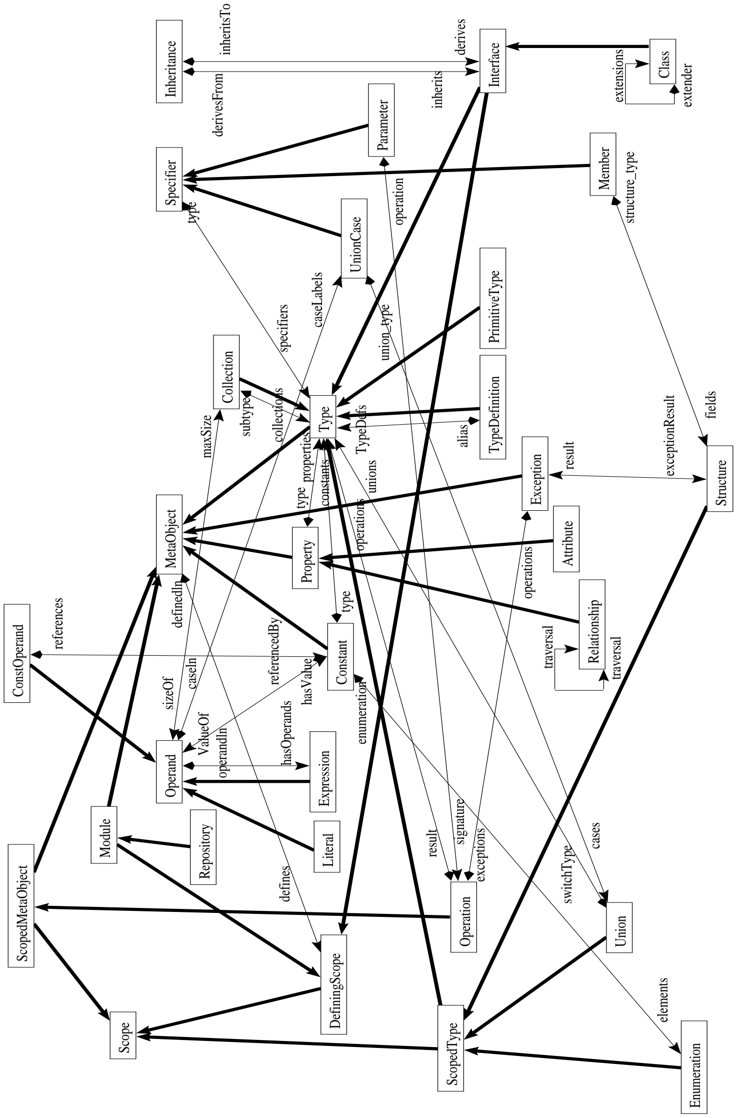

Figura A.1: Struttura del Database Schema Repository

ESEMPIO: CREAZIONE DI UNO SCHEMA

Per esemplificare quanto appena visto vogliamo ora applicare le varie operazioni definite sui metadati, allo scopo di creare lo schema di un database di un' universitá, supponendo che la sua definizione in ODL sia la seguente:

```
interface Course
    ( extent courses
    keys (name, number))
    {
    attribute     string            name;
    attribute     unsigned short    number;
    relationship  list<Section>     has-sections
                      inverse Section::is-section-of;
    };

interface Section
    ( extent sections
    keys (number))
    {
    attribute     string            number;
    attribute     Address           sec-address;
    relationship  Course            is-section-of
                      inverse Course::has-sections;
    relationship  set<Student>      is-taked
                     inverse Student takes;
    relationship  set<Employee>     is-assisted-from
                     inverse Employee assists;
    };

interface STheory : Section()
    {
    attribute     short             level;
    relationship  set<Professor>    is-assisted-from
                     inverse Professor assists;
    };

interface STraining : Section()
    {
    attribute     string          features;
    relationship  set<TA>         is-assisted-from
                     inverse TA assists;
    };
```

```
interface Employee
      ( extent employees
      keys (name, id))
      {
      attribute     string            name;
      attribute     short             id;
      attribute     unsigned short    annual-salary;
      attribute     string            domicile-city;
      attribute     string            residence-city;
      relationship  set<Section>      assists
                         inverse Section is-assisted-from;
      };

interface Professor: Employee
      ( extent professors)
      {
      attribute     string             rank;
      relationship  set<STheory>       assists
                         inverse STheory is-assisted-from;
      };

interface TA: Employee, Student ()
      {
      relationship  set<STraining>      assists
                         inverse STraining is-assisted-from;
      attribute     struct TA-Address { string city;
                                        string street;
                                        string tel-number; }
                                        address;
      };

interface Assistant: Employee, Student ()
      {
      attribute     Address      address;
      };

interface Student
      ( extent students
      keys (name, student-id))
      {
      attribute     string           name;
```


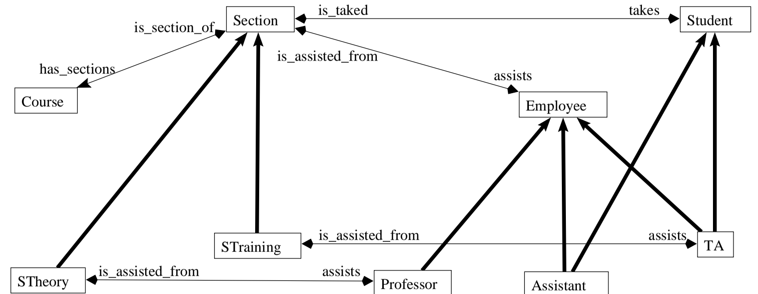

Figura A.2: Schema del database di un' universitá

```
      attribute     long             student-id;
      attribute     struct Address { string city;
                                     string street;
                                     } dorm-address;
      relationship  set<Section>      takes
                         inverse Section is-taked;
      };
```

La rappresentazione di tale schema é contenuta in figura A.2.

Sarebbe opportuno utilizzare uno dei bindings che ODMG93 offre, dal momento che l' ODL puro é un linguaggio astratto. Tuttavia la nuova versione dello standard non contiene ancora (pur promettendo che tal lacuna verrá in futuro colmata) la traduzione in C++ dei metodi che consentono la creazione o la modifica di uno schema.
Per tale motivo scriveremo la sequenza delle operazioni necessarie per creare lo schema suddetto in un linguaggio che non é quello previsto dal C++-binding, bensí uno pseudo-C in cui i metodi vengono riportati utilizzando la sintassi ODL appena vista.

```
typedef enum PrimitiveKind {pk-boolean,pk-char,pk-short, pk-ushort,
                            pk-long,pk-ulong,pk-float,pk-double,
                            pk-octet,pk-string,pk-void,pk-any};
typedef enum CollectionKind {ck-list,ck-array,ck-bag,ck-set, ck-dictionary};

begin
{
```

```
Module deposito;  /*serve per poter applicare i metodi
                       che creano le classi del database*/
Interface c1,c2,c3,c4,c5,c6,c7,c8;
Primitive-type ptipo-string,ptipo-short,ptipo-ushort,
                  ptipo-long;
Member m1,m2,m3;
Structure TA-ind,ind;
Attribute a11,a12,a21,a22,a31,a41,a51,a52,a53,a54,a55,a61
            a71,a81,a91,a92,a93;
Relationship r1,r21,r22,r23,r3,r4,r5,r6,r7,r9;
Literal o1;
Collection set-section,set-straining,set-stheory,list-section,
            set-student,set-professor,set-ta,set-employee;
string nome,lungh;
PrimitiveKind pk;
CollectionKind ck;

/* creo le varie interfacce */

/* creo l' interfaccia Course*/
strcpy(nome,"Course");
c1=deposito.add-interface(nome);

/* creo l' interfaccia Section*/
strcpy(nome,"Section");
c2=deposito.add-interface(nome,list("STraining","STheory "));

/* creo l' interfaccia STheory*/
strcpy(nome,"STheory");
c3=deposito.add-interface(nome);

/* creo l' interfaccia STraining*/
strcpy(nome,"STraining");
c4=deposito.add-interface(nome);

/* creo l' interfaccia Employee*/
strcpy(nome,"Employee");
c5=deposito.add-interface(nome,
                         list("Assistant","TA","Professor"));

/* creo l' interfaccia Professor*/
```



```
strcpy(nome,"Professor");
c6=deposito.add-interface(nome);

/* creo l' interfaccia TA*/
strcpy(nome,"TA");
c7=deposito.add-interface(nome);

/* creo l' interfaccia Assistant*/
strcpy(nome,"Assistant");
c8=deposito.add-interface(nome);

/* creo l' interfaccia Student*/
strcpy(nome,"Student");
c8=deposito.add-interface(nome,list("Assistant","TA");

/* creo i tipi primitivi*/

/* Tipo stringa*/
pk=pk-string;
ptipo-string=deposito.create-primitive-type(pk);

/* Tipo unsigned-short*/
pk=pk-ushort;
ptipo-ushort=deposito.create-primitive-type(pk);

/* Tipo short*/
pk=pk-short;
ptipo-short=deposito.create-primitive-type(pk);

/* Tipo long*/
pk=pk-long;
ptipo-long=deposito.create-primitive-type(pk);

/* creo le strutture "TA-Address" e "Address" */

/* creo il member (dell' indirizzo) "city" */
strcpy(nome,"city");
m1=deposito.create-member(nome,ptipo-string);

/* creo il member (dell' indirizzo) "street" */
strcpy(nome,"street");
```

```
m2=deposito.create-member(nome,ptipo-string);

/* creo il member (dell' indirizzo) "tel-number" */
strcpy(nome,"tel-number");
m3=deposito.create-member(nome,ptipo-string);

/* creo la struttura "TA-Address" */
strcpy(nome,"TA-Address");
TA-ind=deposito.add-structure(nome,list(m1,m2,m3));

/* creo la struttura "Address" */
strcpy(nome,"Address");
ind=deposito.add-structure(nome,list(m1,m2));

/* Creazione degli attributi */

/* associo a Count i suoi attributi*/
strcpy(nome,"name");
a11=c1.add-attribute(nome,ptipo-string);
strcpy(nome,"number");
a12=c1.add-attribute(nome,ptipo-short);

/* associo a Section i suoi attributi*/
strcpy(nome,"number");
a21=c2.add-attribute(nome,ptipo-string);
strcpy(nome,"sec-address");
a22=c2.add-attribute(nome,ind);

/* associo a STheory i suoi attributi*/
strcpy(nome,"level");
a31=c3.add-attribute(nome,ptipo-short);

/* associo a STraining i suoi attributi*/
strcpy(nome,"features");
a41=c4.add-attribute(nome,ptipo-string);

/* associo a Employee i suoi attributi*/
strcpy(nome,"name");
a51=c5.add-attribute(nome,ptipo-string);
strcpy(nome,"id");
a52=c5.add-attribute(nome,ptipo-short);
```



```
strcpy(nome,"annual-salary");
a53=c5.add-attribute(nome,ptipo-ushort);
strcpy(nome,"domicile-city");
a54=c5.add-attribute(nome,ptipo-string);
strcpy(nome,"residence-city");
a55=c5.add-attribute(nome,ptipo-string);

/* associo a Professor i suoi attributi*/
strcpy(nome,"rank");
a61=c6.add-attribute(nome,ptipo-string);

/* associo a TA i suoi attributi*/
strcpy(nome,"address");
a71=c7.add-attribute(nome,TA-ind);

/* associo a Assistant i suoi attributi*/
strcpy(nome,"address");
a81=c8.add-attribute(nome,ind);

/* associo a Student i suoi attributi*/
strcpy(nome,"name");
a91=c9.add-attribute(nome,ptipo-string);
strcpy(nome,"student-id");
a92=c9.add-attribute(nome,ptipo-long);
strcpy(nome,"dorm-address");
a93=c9.add-attribute(nome,ind);

/* creo i tipi "Collection" set e list per creare i quali devo
   prima creare un' istanza della classe Operand che contenga la dimensione
   massima della collezione*/

strcpy(lungh,"10000");
o1=deposito.create-operand(lungh);

/* creo il tipo set<Section>*/
ck=ck-set,
set-section=deposito.create-collection-type(ck,o1,c2);

/* creo il tipo set<STraining>*/
set-straining=deposito.create-collection-type(ck,o1,c4);
```

```
/* creo il tipo set<STheory>*/
set-stheory=deposito.create-collection-type(ck,o1,c3);

/* creo il tipo set<Student>*/
set-student=deposito.create-collection-type(ck,o1,c9);

/* creo il tipo set<Employee>*/
set-employee=deposito.create-collection-type(ck,o1,c5);

/* creo il tipo set<Professor>*/
set-professor=deposito.create-collection-type(ck,o1,c6);

/* creo il tipo set< TA>*/
set-ta=deposito.create-collection-type(ck,o1,c7);

/* creo il tipo list<Section>*/
ck=ck-list,
list-section=deposito.create-collection-type(ck,o1,c1);

/* creazione delle relazioni */

/* associo a Course le sue relazioni*/
strcpy(nome,"has-section");
r1=c1.add-relationship(nome,list-section,r21);

/* associo a Section le sue relazioni*/
strcpy(nome,"is-section-of");
r21=c2.add-relationship(nome,c1,r1);
strcpy(nome,"is-taked");
r22=c2.add-relationship(nome,set-student,r9);
strcpy(nome,"is-assisted-from");
r23=c2.add-relationship(nome,set-employee,r5);

/* associo a STheory le sue relazioni*/
strcpy(nome,"is-assisted-from");
r3=c3.add-relationship(nome,set-professor,r6);

/* associo a STraining le sue relazioni*/
strcpy(nome,"is-assisted-from");
r4=c4.add-relationship(nome,set-ta,r7);
```



```
/* associo a Employee le sue relazioni*/
strcpy(nome,"assists");
r5=c5.add-relationship(nome,set-section,r23);

/* associo a Professor le sue relazioni*/
strcpy(nome,"assists");
r6=c6.add-relationship(nome,set-stheory,r3);

/* associo a TA le sue relazioni*/
strcpy(nome,"assists");
r7=c7.add-relationship(nome,set-straining,r4);

/* associo a Student le sue relazioni*/
strcpy(nome,"takes");
r9=c9.add-relationship(nome,set-section,r22);
```

NOTE :

- Per applicare metodi relativi all' interfaccia *DefiningScope* (come ad esempio *create-primitive-type*) non ho creato un oggetto di questa metaclasse, bensí ho fatto uso di *deposito*, il quale, appartenendo all' interfaccia *Module*, eredita anche tutti i metodi di *DefiningScope*.
  Tuttoció é reso necessario dal fatto che l' interfaccia *DefiningScope* non é direttamente istanziabile.
- Nell' usare il metodo *add-interface* della classe *Module* ho supposto di dovere specificare il secondo parametro indicato nella sua dichiarazione solo nel caso in cui l' interfaccia che creo erediti proprietá e operazioni da una o piú superclassi.

A.1.5 LOCK E CONTROLLO DELLA CONCORRENZA

Nella nuova release viene aggiunta anche la trattazione del problema del controllo della concorrenza, realizzato tramite un meccanismo di locking degli oggetti di un database.
Tale meccanismo ha lo scopo di mantenere il database in uno stato consistente per ogni transazione attiva, consentendo a ogni processo il lock su un oggetto, a patto che non sia conflittuale con altri lock giá esistenti sull' oggetto stesso.

TIPI DI LOCK

Distinguiamo tre tipi di lock:

read: consente accesso in lettura allo stesso oggetto a piú processi contemporaneamente.

write: indica accesso esclusivo ad un oggetto a scopo di modifica.

upgrade: serve per prevenire una forma di deadlock che capita quando due processi ottengono entrambi un lock in lettura su di un oggetto e in seguito tentano di ottenerne un altro in scrittura sull' oggetto stesso.
Infatti i lock di tipo upgrade sono compatibili con lock di tipo read, ma sono in conflitto con lock di tipo write o altri lock di tipo upgrade.
Il deadlock viene scongiurato assegnando un lock di tipo upgrade, al posto di un lock di tipo read, ad ogni processo che richiede l' accesso in lettura ad un oggetto che si intende modificare nel seguito. Questo evita potenziali conflitti nel caso in cui successivamente venga assegnato un permesso di scrittura sull' oggetto stesso.

LOCKING IMPLICITO E ESPLICITO

L' ODMG Object Model supporta due tipi di locking:

lock implicito: si tratta di un lock ottenuto durante la navigazione dello schema del mio database.
Ad esempio un lock in lettura é ottenuto ogniqualvolta si accede ad un dato oggetto, uno in scrittura quando invece quest' ultimo viene modificato.
Si noti che l' assegnazione di un lock di questo tipo é automatica e non richiede alcuna operazione specifica.

lock esplicito: si ottiene in seguito a specifica richiesta, fatta tramite le operazioni **lock** o **try-lock** definite dallo standard ODMG sulle istanze del tipo *Object*.
Da notare che, mentre lock di tipo *read* o *write* possono essere ottenuti sia in modo implicito che esplicito, un lock di tipo *upgrade* é sempre esplicito.

DURATA DEL LOCK

Tutti i lock per default vengono mantenuti fino a che la transazione relativa raggiunge il "commit" o viene abortita (concordemente alla politica adottata da SQL92).

A.1.6 TRANSAZIONI

Per quanto riguarda le transazioni, esse sono viste dall' ODBMS come istanze di un tipo *Transaction*, su cui é possibile definire un set di operazioni, cosa che giá la release 1.1 si preoccupava di fare.
La nuova versione dello standard da un lato arricchisce questo set, dall' altro definisce un nuovo tipo, accanto al tipo *Transaction*, che chiama *TransactionFactory* che viene usato per creare transazioni.
Entrando un pó piú nel dettaglio, le operazioni definite sui due tipi sono le seguenti:

```
interface TransactionFactory {
    Transaction    new();
    Transaction    current();
    };

interface Transaction {
    exception    TransactionInProgress{};
    exception    TransactionNotInProgress{};
    void         begin() raises(TransactionInProgress);
    void         commit() raises(TransactionNotInProgress);
    void         abort() raises(TransactionNotInProgress);
    void         checkpoint() raises(TransactionNotInProgress);
    void         join();
    void         leave();
    boolean      isOpen();
    };
```

Di queste quelle indicate in corsivo sono novitá introdotte dalla versione 2.0, mentre le altre esistevano giá anche nella versione 1.1.

A.1.7 OPERAZIONI SUL TIPO "DATABASE"

Ogni ODBMS puó gestire uno o piú database logici, ognuno dei quali puó essere memorizzato in uno o piú database fisici.
Ciascun database logico é visto poi dal sistema come un'istanza del tipo *Database*.
Analogamente alle transazioni, ODMG 2.0 arricchisce il set delle operazioni definite sul tipo "Database" e introduce un nuovo tipo (*DatabaseFactory*), usato per creare nuovi databases.
In particolare le operazioni definite sui due tipi sono le seguenti:

```
interface DatabaseFactory {
      Transaction   new();
      };

interface Database {
      void      open(<database-name>);
      void      close();
      void      bind(in any an-object, in string name);
      Object    unbind(in string name);
      Object    lookup(<object-name>);
      Module    schema();
      };
```

Di queste quelle indicate in corsivo sono novitá introdotte dalla versione 2.0, mentre le altre esistevano giá anche nella versione 1.1.

new(): crea una nuova istanza di tipo *Database*.

open(<database-name>): apre un database giá creato

bind(in any an-object, in string name): collega un nome a un database.

unbind(in string name): scollega un database dal suo nome.

lookup(<object name>): trova l' identificatore dell' oggetto "Database" avente come nome la stringa passata come parametro.

schema(): ritorna il meta-oggetto *Module* che contiene lo schema dell' ODBMS.

A.2 OBJECT DESCRIPTION LANGUAGE (ODL)

Per quanto riguarda invece l' ODL, si mettono in evidenza le seguenti differenze:

1. Una prima differenza riguarda la definizione dei tipi oggetto:

Release 1.1 Un tipo oggetto viene definito dal punto di vista sintattico tramite il costrutto *interface*, e il suo BNF é il seguente:

```
< interface-dcl >   ::= interface < identifier > [ < inheritance-spec >]
                        < type-property-list >
                        [: < persistence-dcl > ]
                        {[ < interface-body > ]};
< persistence-dcl > ::= persistent | transient
```

Release 2.0 Un tipo oggetto puó essere sintatticamente definito come *interface* o come *class* utilizzando gli omonimi costrutti. Il BNF in questo caso si presenta cosí strutturato:

```
< interface-dcl > ::= interface < identifier > [ < inheritance-spec >]
                      {[ < interface-body > ]};
        < class > ::= class-header {< interface-body >}
 < class-header > ::= class < identifier >
                      [ extends < scoped-name > ]
                      [ < inheritance-spec >]
                      [ < type-property-list >]
```

Si noti anche come la <type-property-list> sia prevista solo nella definizione di *class*, e non piú in quella di *interface*, come accadeva nella release 1.1.

2. Come si puó notare dall' analisi delle due porzioni di sintassi sopra riportate, v' é un' altra differenza che risulta subito evidente, che é l' eliminazione, all' interno della definizione di *interface*, della possibilitá di scegliere tra le opzioni *persistent* e *transient*, stabilendo cosí la persistenza o meno del tipo oggetto creato.

3. La Release 2.0 non prevede la possibilitá di evidenziare sintatticamente l' ordinamento di una relazione sulla base di un insieme di attributi. Ció era invece consentito nella Release 1.1 tramite la clausola opzionale order-by, da inserire alla fine della definizione di *relationship*, racchiusa tra parentesi graffe e seguita dalla lista di attributi di ordinamento, che doveva peró essere completamente contenuta nella "attribute list" del tipo target della relazione.

4. La Release 2.0 affianca all' ODL un altro linguaggio, l' OIF (*Object Interchange Format*), che puó essere usato per:

- Scambiare oggetti tra database.
- Fornire documentazione sul database definito con ODL.
- Guidare test su database.

Si tratta non di un linguaggio di programmazione, ma piuttosto di un linguaggio di specificazione di oggetti, che ha la interessante peculiaritá di non richiedere l' uso di parole chiave, eccezion fatta per i nomi con cui identifico tipi, attributi e relazioni all' interno del database definito con l' ausilio di ODL. Scopo del prossimo paragrafo sará perció quello di analizzare un pó piú nel dettaglio le caratteristiche di base di questo linguaggio.

N.B. OIF non é un linguaggio alternativo all' ODL, bensí di supporto; suo compito non é perció definire un database, ma consentire in maniera semplice tutta una serie di operazioni su di esso.

A.2.1 IL LINGUAGGIO OIF

Per caratterizzare lo stato di tutti gli oggetti contenuti in un database, l' OIF utilizza:

- Identificatori di oggetto
- Binding con tipi
- Valore degli attributi
- Links con altri oggetti

DEFINIZIONE DI UN OGGETTO

Di ogni oggetto OIF specifica un identificatore, il tipo a cui l' oggetto appartiene, il valore degli attributi e le relazioni con altri oggetti.

Esempio:

Supponiamo di considerare un database all' interno del quale sia definita la classe Person nel seguente modo:


```
interface Person {
   attribute string Name;
   attribute unsigned short Age;
};
```

Il frammento di codice

Jack **Person**{**Name** "Sally", **Age** 11} (A.1)

crea una istanza della classe Person, dotandola del nome convenzionale "Jack", che d' ora in poi sará usato per referenziare l' istanza appena definita (come identificatore) e pertanto non potrá in alcun modo essere duplicato in nessuna altra istanza del database.
Dal canto suo Person costituirá una parola chiave per il linguaggio OIF, dal momento che si tratta del nome di una classe del database, e quindi non potrá mai essere utilizzata come identificatore di una qualunque istanza. Infine, la stringa tra parentesi graffe inizializza, relativamente all' oggetto "Jack", gli attributi *Name* e *Age* rispettivamente con i valori "Sally" e 11.

NOTE:

- E' possibile anche non mettere nulla tra le parentesi graffe: in questo caso l' istanza viene creata senza che i suoi attributi vengano inizializzati.
- L' inizializzazione degli attributi puó anche essere fatta in modo "veloce"; infatti, se viene mantenuto l' ordine con cui essi sono specificati all' interno della definizione ODL della classe, é possibile omettere i nomi degli attributi e le virgole. La A.1 puó perció essere scritta nel seguente modo:

  Jack **Person**{"Sally" 11} (A.2)

- Spesso puó capitare che piú istanze debbano essere inizializzate con lo stesso set di valori dei loro attributi; in tal caso, OIF prevede la possibilitá di eseguire una inizializzazione per copia. La definizione

  Bill(Jack) **Person**{Jack} (A.3)

  crea un' istanza della classe Person, avente come identificatore "Bill" e come set di valori dei suoi attributi lo stesso set della istanza "Jack".

INIZIALIZZAZIONE DI ATTRIBUTI

OIF definisce anche i range di variabilitá degli attributi a seconda del tipo a cui essi appartengono:

TIPO BOOLEAN: Un attributo di tipo boolean puó assumere valore TRUE o FALSE.

TIPO CHARACTERS: Un attributo di tipo char é definito tramite uno o piú caratteri racchiusi tra apici (es. 'a','ab','abc').

TIPO SIGNED INTEGER: Gli interi si dividono in short- e long-integer, ed in entrambi i casi si tratta di una sequenza di numeri preceduta eventualmente dal segno meno. Per default il numero é considerato in base 10; é tuttavia possibile specificare una base differente, anteponendo alla sequenza di numeri "0" nel caso si richieda la base ottale, "0x" (oppure "0X") se invece si desidera utilizzare la base esadecimale.

TIPO UNSIGNED INTEGER: Valgono le considerazioni relative al punto precedente, eccezion fatta per il fatto che in questo caso non é prevista la possibilitá di usare il segno meno per indicare i numeri negativi.

TIPO FLOAT: Un attributo di tipo float si puó inizializzare tramite un float literal; esso é composto da (nell' ordine) un segno meno (opzionale), una parte intera, una virgola, una parte decimale, una "e" (oppure "E") e un esponente (che puó essere positivo o negativo). La parte intera, quella decimale e l' esponente sono da intendersi come sequenze di numeri.

TIPO STRING: Un attributo di tipo stringa é definito attraverso una sequenza di caratteri racchiusi tra doppi apici (es. "abcd", "pippo").

TIPO STRUCTURE: Supponiamo sia data la seguente definizione della classe *Person* tramite ODL:

```
struct PhoneNumber {
   Unsigned short CountryCode;
   Unsigned short AreaCode;
   Unsigned short PersonCode;
};
```



```
struct Address {
   string Street;
   string City;
   PhoneNumber Phone;
};

interface Person {
   attribute string Name;
   attribute Address PersonAddress;
};
```

L' inizializzazione in OIF é fatta nel modo seguente:

Jack **Person** {**Name** "Jack",
PersonAddress {**Street** "Willow Road",
City "Palo Alto",
Phone {**CountryCode** 1,
AreaCode 415,
PersonCode 1234}}}

TIPO ARRAY: Sia data la seguente definizione di classe:

```
interface Engineer {
   attribute unsigned short PersonID[3];
};
```

Supponiamo ora di aver definito un' istanza di questa classe, di averla dotata del identificatore "Jack", e di voler inizializzare alcuni campi dell' array "PersonID" relativo a tale istanza, ad esempio il primo ("PersonID[0]") e il terzo ("PersonID[2]"); la sintassi OIF che realizza ció é:

Jack **Engineer**{**PersonID**{[0] 450, [2] 270}} (A.4)

In questo modo assegno i valori 450 al campo "PersonID[0]" e 270 al campo "PersonID[2]".

NOTE: • Notiamo che, nel caso si debbano inizializzare un insieme di campi consecutivi di un array a partire dal primo, la

notazione puó essere semplificata nel modo seguente:

Jack **Engineer**{**PersonID**{450,180,270}} (A.5)

In questo modo inizializzo i primi tre campi dell' array "PersonID" ai valori 450,180 e 270 rispettivamente.

- Nelle A.4 e A.5 le virgole sono del tutto opzionali e possono essere sostituite con spazi bianchi.

TIPO COLLECTION: Sia data la seguente classe (definita in ODL):

```
interface Professor: Person {
  attribute set<string> Degrees;
};
```

Supponiamo di voler definire una istanza di questa classe avente la stringa "Feynman" come identificatore, e il cui set di stringhe relative all' attributo *Degrees* sia composto da "Masters" e "PhD"; la sintassi OIF per far ció é:

Feynman **Professor**{**Degrees** {"Masters","PhD"}} (A.6)

Un discorso a parte merita il caso in cui il tipo "collection" sia un array dinamico, ad esempio:

```
struct Point {
  float X;
  float Y;
};

interface Polygon {
  attribute array<Point> RefPoints;
};
```

In questo caso *Polygon* ha come attributo *RefPoints*, che é un array di strutture. Supponendo di avere definito un' istanza P1 di questa classe, se scrivo:

P1 **Polygon**{**RefPoints**{[5]{**X** 7.5,**Y** 12.0},
[11]{**X** 22.5,**Y** 23.0}}} (A.7)

in questo modo ho inizializzato i campi con indice 5 e 11 dell' array di strutture relativo all' istanza P1 creata. I campi non specificati in A.7 restano non inizializzati.

DEFINIZIONE DI RELAZIONI

Vogliamo ora occuparci della sintassi OIF che consente di definire relazioni tra oggetti. Sia data la classe:

```
interface Person {
  relationship Company Employer
      inverse Company::Employees;
};
```

Supponendo ora, date un' istanza "Jack" di *Person* e un' istanza "McGraham" di *Company*, di volerle mettere in relazione, la sintassi OIF che realizza ció é:

Jack **Person**{**Employer** McGraham} (A.8)

Se peró la relazione non é piú *one-to-one* bensí *one-to-many* oppure *many-to-many* é necessario specificare i nomi di tutti gli oggetti "linkati" tramite la relazione. Ció significa che se considero

```
interface Company {
  relationship set<Person> Employees
      inverse Person::Employer;
};
```

supponendo che la compagnia "McGraham" sia in relazione con le persone "Jack", "Bill" e "Joe", la relazione tra queste istanze verrá indicata

McGraham **Company**{**Employees** {Jack, Bill, Joe}} (A.9)

E' chiaro che gli oggetti coinvolti nella relazione devono essere appartenenti alle classi che la relazione stessa collega.

DATA MIGRATION

E' previsto anche un meccanismo di *"object forward declaration"*, che induce una ricerca da parte del sistema all' interno del database, per determinare l' esistenza o meno di oggetti il cui nome convenzionale (identificatore) é usato nella dichiarazione di altri oggetti.
Se la ricerca dá esito negativo viene generato a "run-time" un errore.
Ad esempio, supponiamo sia data la seguente definizione di classe in ODL

```
interface Node {
   relationship set<Node> Pred
       inverse Node::Succ;
   relationship set<Node> Succ
       inverse Node::Pred;
};
```

ipotizziamo inoltre di aver definito i seguenti oggetti (tramite sintassi OIF)

A **Node**{**Pred** {B}}
E **Node**
B **Node**{**Pred** {E}, **Succ** {C}}
C **Node**{**Pred** {A}, **Succ** {F}}
F **Node**

In questo caso le forward declarations sono quelle di "E" e "F".
Grazie ad esse il sistema esegue automaticamente una ricerca, all' interno del database, di due oggetti aventi lo stesso nome convenzionale, linkando poi, in caso la ricerca abbia successo, gli oggetti trovati a "B" e "C".

COMMAND LINE UTILITIES

OIF definisce anche i seguenti programmi di utilitá, richiamabili dalla linea di comando:

odbdump: serve per effettuare il "dump" di un database. La sintassi é

odbdump <database-name>

L' effetto é quello di creare una rappresentazione secondo specifiche OIF del database nominato come parametro.
Da notare che gli identificatori di oggetto sono creati in modo automatico da algoritmi di generazione di nomi.

odbload: serve per effettuare il "load" di un database. La sintassi é

odbload <database-name> <file 1> ... <file n>

L' effetto é quello di popolare il database specificato con gli oggetti contenuti nei file indicati.

A.3 OBJECT QUERY LANGUAGE

Se ora consideriamo l' OQL, i principali concetti introdotti dalla release 2.0 che non erano trattati nella release 1.1 sono:

A.3.1 PATH EXPRESSIONS

Per accedere ai dati contenuti in uno schema predefinito, ODMG 2.0 introduce la possibilitá di eseguire, accanto ad un accesso di tipo tradizionale, basato sul linguaggio di interrogazione OQL, un accesso di tipo "navigazionale". Viene in altri termini consentita la navigazione all' interno dello schema attraverso l' uso appunto delle Path expressions, le quali sfruttano la gerarchia di aggregazioni di cui lo schema é dotato.
Per far ció si utilizza il "." (o indifferentemente "->") nel modo che andiamo ora a mostrare con un esempio.
Se ho una persona P e voglio sapere il nome della cittá dove essa vive, utilizzo la path expression:

p.address.city.name

Questa ha lo stesso effetto di una query che parte dalla classe dove sono contenute tutte le persone, seleziona quella desiderata (la persona P), per poi andare a navigare la gerarchia di aggregazioni che parte da quella classe fino a raggiungere il nome della cittá dove la persona selezionata vive.

NOTE:

- E' facile comprendere che l' uso di Path expressions é alquanto pesante per il sistema, dal momento che la loro semplicitá e potenza espressiva nasconde la necessitá, da parte del sistema, di eseguire in modo automatico tutti i join impliciti (vale a dire indotti dalla gerarchia di aggregazione) che la Path expression usata sottointende.
- Le Path expressions sono spesso utilizzate all' interno di interrogazioni scritte in OQL per rendere queste piú semplici e leggibili.

A.3.2 VALORI NULLI

La nuova Release tratta anche in modo esplicito il problema dei valori nulli (nil); essi sono considerati a loro volta degli oggetti, accedendo ai quali si ottiene come risultato UNDEFINED, che é un valore che deve essere opportunamente gestito. A tal proposito ODMG definisce le seguenti regole:

- Le operazioni di "." e "->", applicate a UNDEFINED, danno come risulato UNDEFINED.
- Operazioni logiche (=,!=,<,>,<=,>=) in cui uno o entrambi gli operatori sono UNDEFINED danno False come risultato.
- vengono definite due operazioni unarie che rendono un valore booleano a seconda che il loro operando sia uguale o meno al valore nullo, che sono:

  is-undefined(op): restituisce True se op=UNDEFINED, False in caso contrario.

  is-defined(op): restituisce False se op=UNDEFINED, True in caso contrario.
- Ogni altra operazione che coinvolga un UNDEFINED restituisce a run-time un errore.

A.3.3 INVOCAZIONE DI METODI

Viene fornita la possibilitá di invocare metodi all' interno di query. La notazione per poter fare ció é la stessa usata per accedere a un attributo o navigare una relazione, nel caso il metodo non abbia parametri; se invece li ha, bisogna elencarli tra parentesi tonde.
Ad esempio, se ho una classe *Person* su cui ho definito due metodi, **oldest-child** (senza parametri), che ritorna un oggetto della classe *Person* e **lives-in(<city>)**, che ritorna un valore booleano indicante se la persona vive o meno nella cittá <city>, la query

```
select p.oldest-child.address.street
from Persons p
where p.lives-in("Paris")
```

seleziona la strada dell' indirizzo del bambino piú vecchio tra quelli che vivono a Parigi.

A.3.4 POLIMORFISMO

ODMG 2.0 tratta anche il problema del polimorfismo. Siano date due classi, *Person* e *Employee*, legate tra loro da una relazione di ereditarietá, e supponiamo di avere due metodi "X", con la stessa signature, inseriti in queste due classi. L' invocazione di "X" implica la necessitá di stabilire a run-time a

quale metodo faccio riferimento; tale compito é svolto automaticamente dal sistema, il quale sceglie basandosi di volta in volta sul fatto che la persona considerata appartenga a *Person* oppure a *Employee* (fenomeno del late binding).
E' peró possibile forzare il sistema a considerare uno dei metodi omonimi, specificando esplicitamente il nome della classe in cui tale metodo é contenuto: in riferimento all' esempio precedente, la query

```
select ((Employee)p).seniority
from Persons p
where X=true
```

forza l' applicazione del metodo "X" associato alla classe *Employee*; chiaramente gli oggetti della classe *Person* che non sono *Employee* non vengono considerati.

A.3.5 COMPATIBILITA' TRA TIPI

Viene introdotta la definizione di compatibilitá tra tipi nel modo seguente: Dato un tipo "t"

1. t é compatibile con se stesso
2. Se t é compatibile con un tipo t', allora
   - set(t) é compatibile con set(t')
   - bag(t) é compatibile con bag(t')
   - list(t) é compatibile con list(t')
   - array(t) é compatibile con array(t')
3. Se t é supertipo di t_1 e t_2, allora t_1 e t_2 sono compatibili. Ció in particolare implica che:
   - Literals e oggetti non sono compatibili.
   - Literals di tipo atomico (es. char, float) sono compatibili solo con se stessi.
   - Literals di tipo strutturato sono compatibili solo se hanno un supertipo comune.
   - Oggetti di tipo atomico sono compatibili solo se hanno un supertipo comune.

- Oggetti di tipo collezione sono compatibili solo se appartengono alla stessa collezione e se i tipi dei loro membri sono compatibili.

Da notare é poi anche il fatto che se t_1, t_2, …,t_n sono compatibili, allora esiste un unico t tale che:

1. $t > t_i, \ \forall i = 1, \ldots, n$
2. $\forall t' : \ t'!{=}t \ and \ t' > t_i, \ \forall i \rightarrow t' > t$

Questo tipo t é indicato con lub$(t_1, t_2, \ldots, t_n)$[2].

A.3.6 DEFINIZIONE DI QUERY

La Release 2.0 affronta in modo piú completo la questione delle espressioni che definiscono le query, rispetto a quanto non faccia la release 1.1. Vediamo perció nel dettaglio come viene trattato il problema nei due casi.

Release 1.1: Ci si limita a dire semplicemente che una query puó essere definita tramite la sintassi

define <q-name> as <q-expr>

dove <*q-name*> é il nome della query, mentre <*q-expr*> é l' espressione che definisce la query.

Release 2.0: Viene estesa la sintassi di definizione della query nel modo seguente:

define [query] <q-name>$(x_1, x_2, \ldots, x_n)$ as <q-expr>

dove $x_1, x_2, \ldots, x_n$ sono variabili che vengono utilizzate in <q-expr>, mentre [query] é una parola chiave opzionale. Se la definizione non é dotata di parametri é possibile omettere le parentesi tonde.

Esempi:

[2] *lub* é l' acronimo di *least upper bound*.


```
define age(x) as
    select p.age
    from Persons p
    where p.name=x

define smiths() as
    select p
    from Persons p
    where p.name="Smith"
```

Una query cosí definita é persistente e rimane attiva finché non é sovrascritta (da una nuova definizione) oppure cancellata tramite la sintassi

delete definition <q-name>

Affinché una definizione ne sovrascriva una preesistente, é necessario che abbia lo stesso nome (<q-name>) e un numero uguale di parametri; se anche una sola di queste due condizioni non é verificata la definizione introdotta é interpretata come nuova e non porta perció alla cancellazione di alcuna definizione giá esistente.

A.3.7 STRING EXPRESSIONS

ODMG 2.0 introduce anche operazioni su espressioni di tipo "string", non previste nella versione precedente dello standard. In particolare:

- Se s_1 e s_2 sono stringhe, $s_1||s_2$ e $s_1 + s_2$ sono pure esse espressioni di tipo stringa, il cui valore é dato dal concatenamento delle due stringhe date.
- Se c é un carattere e s é una stringa, allora c *in* s é un' espressione di tipo boolean il cui valore é true se il carattere c appartiene alla stringa, false in caso contrario.
- Se s é una stringa e i é un intero, allora Si é un' espressione di tipo carattere il cui valore é pari all' $i + 1$-esimo carattere della stringa s.
- Se s é una stringa e *low* e *up* sono interi, allora s[*low:up*] é una stringa il cui valore é la sottostringa della stringa data compresa tra il (low+1)-esimo e il (up+1)-esimo carattere.

- Se *s* e *pattern* sono due stringhe, allora *s like pattern* é un' espressione di tipo boolean il cui valore é true se *pattern* contiene *s*, false in caso contrario.
  Si noti in questo caso come sia possibile usare in *pattern* i caratteri speciali "?" oppure "_" al posto di un qualunque carattere, "*" oppure "%" al posto di una qualunque stringa (inclusa la stringa vuota).

  Esempio:

  'A nice string' like '%nice%str_ng'   restituisce valore true

  Nel caso sia necessario usare in *pattern* uno dei quattro caratteri speciali sopra elencati come semplice carattere lo si puó fare facendolo precedere da un backslash ('\').

A.3.8 COMPARAZIONE DI OGGETTI E LITERALS

La nuova versione dello standard introduce anche la sintassi che consente la comparazione di oggetti e literals, che é

$e_1 = e_2$ per verificare l' uguaglianza

$e_1 != e_2$ per verificare la disuguaglianza

Osservo che e_1 e e_2 sono espressioni che denotano oggetti (o literals) di tipi compatibili, e che entrambe le operazioni ritornano un valore boolean che é vero o falso a seconda che i due oggetti (literals) siano uguali (diversi) o meno.
Da notare c' é il fatto che due oggetti sono considerati uguali se sono lo stesso oggetto, vale a dire se hanno lo stesso identificatore.
Invece la determinazione dell' uguaglianza di due literals si basa sui seguenti criteri:

- Se si tratta di literals di tipo atomico, essi devono avere lo stesso valore.
- Se si tratta di literals di tipo "struct", essi devono avere la stessa struttura e gli stessi valori degli attributi.
- Se si tratta di literals di tipo "set", essi devono contenere lo stesso set di elementi.

- Se si tratta di literals di tipo "bag", essi devono non solo contenere lo stesso set di elementi, ma ogni elemento deve essere ripetuto lo stesso numero di volte.
- Se si tratta di literals di tipo "array", essi devono contenere lo stesso set di elementi nello stesso ordine.

A.3.9 QUANTIFICATORI ESISTENZIALI

Nella nuova Release viene introdotto, tra i quantificatori esistenziali (accanto a *exists*), il costruttore *unique*, dotato della stessa sintassi di *exists*, il quale peró ritorna valore true nel caso in cui esista un solo elemento che soddisfi le condizioni specificate.

A.3.10 OPERATORE ORDER-BY

ODMG 2.0 elimina, rispetto alla versione 1.1, l' operatore *sort-by* sostituendolo con l' operatore *order-by*, il quale, inserito all' interno di una query di selezione dopo la clausola **where**, ne ordina l' output sulla base dei criteri specificati dopo la parola chiave **order-by**.
La differenza fondamentale rispetto al sort-by risiede nel fatto che, mentre quest' ultimo aveva esistenza autonoma, vale a dire che da solo poteva tranquillamente costituire una query, l' order-by invece deve per forza essere inserito all' interno di una query di selezione (select).
Ció tuttavia non limita la potenza espressiva del linguaggio, dal momento che nulla mi vieta di scrivere query di tipo **select** che siano del tutto equivalenti ad un *sort-by*, vale a dire che si limitino ad ordinare l' output sulla base di opportuni criteri.
Si noti infine che l' ordinamento puó essere imposto crescente o decrescente tramite la specificazione rispettivamente delle parole chiave **asc** o **desc**, da mettere dopo l' elenco dei criteri sulla base dei quali ordinare l' output della query.

A.3.11 OPERATORE GROUP-BY

Viene inoltre modificato l' operatore **group-by**. Nella release 1.1 si trattava di un operatore autonomo, vale a dire capace da solo di costituire una query, mentre nella nuova versione viene utilizzato esclusivamente come clausola di una query di selezione.
Vale a tal proposito lo stesso discorso fatto al punto precedente circa l' inalterata potenza espressiva del linguaggio, nonostante la apparente

restrizione imposta.
Nella release 2.0 la clausola **group-by** viene messa dopo la clausola **where** e prima dell' eventuale **order-by**, ed ha la seguente sintassi:

select-query **group-by** *partition-attributes* [**having** *predicate*]

In questa espressione *select-query* é una query di selezione, *predicate* é un' espressione booleana, e *partition-attributes* é un' espressione strutturata avente la forma

$att_1 : e_1,\ att_2 : e_2, \ldots, att_n : e_n$

dove att_i sono i nomi degli attributi di partizione e e_i sono invece le espressioni da verificare per poter raggruppare un oggetto sotto il relativo attributo di partizione.
L' effetto del group-by é quello di raggruppare l' output della query sulla base dei *partition-attributes*, dando cosí origine ad un set di strutture, ognuna delle quali contiene elementi aventi tutti gli stessi valori per quanto riguarda gli attributi di partizione.

Ad esempio la query

```
select *
from Employees e
group-by low: salary<1000,
            medium: salary >=1000 and salary < 10000,
            high: Salary >= 10000
```

restituisce un set di tre elementi avente la struttura

```
set<struct(low:boolean, medium:boolean, high:boolean,
              partition: bag<struct(e Employee)>)>
```

In altri termini la terna di variabili booleane *low*, *medium* e *high* consente di distinguere tra loro le varie partizioni fatte, mentre *partition* rappresenta il bag all' interno del quale vengono messi tutti gli oggetti della classe Employee appartenenti a quella partizione.

E' poi possibile, attraverso la specificazione della clausola **having**, scartare alcune delle partizioni fatte, qualora i suoi elementi non soddisfino il *predicate* specificato (analogo a SQL).

A.3.12 DICHIARAZIONE DI VARIABILI NELLA CLAUSOLA "FROM"

La nuova versione dello standard consente variazioni nella sintassi usata per dichiarare le variabili nella clausola **from**, esattamente come in SQL.
In particolare, indicando con **x** una variabile e con **e** un' espressione di tipo collezione, le sintassi ammesse sono:

from x **in** e
from e **as** x
from e x
from e

In quest' ultimo caso l' omissione della variabile x comporta che al suo posto venga usato il nome dell' espressione stessa *(e)*, sia per costruire path expressions, sia per fare riferimento a attributi e proprietá della collezione.
Se poi non ci sono ambiguitá, é possibile non solo omettere nella clausola **from** la variabile **x**, ma addirittura usare direttamente i nomi delle proprietá e attributi, senza farli precedere dal nome della collezione a cui si riferiscono.

Cioé se ho:

```
select Persons.name
from Persons
where Persons.age=10
```

Questa query puó essere anche scritta

```
select name
from Persons
where age=10
```

Non ho ambiguitá se il nome della proprietá o attributo che uso da solo (senza farlo precedere né da un nome di una variabile né di una collezione) é unico tra tutte le classi coinvolte nella query.

A.3.13 ACCESSO AD UN ELEMENTO DI UN DICTIONARY ATTRAVERSO LA SUA CHIAVE

E' prevista la possibilitá di accedere ad un elemento appartenente ad un "dictionary" tramite la sua chiave. Infatti se e_1 é una espressione di tipo *dictionary(k,v)*, ed e_2 é un' espressione di tipo k, allora $e_1[e_2]$ é un' espressione

di tipo v; piú precisamente é il valore associato alla chiave e_2 nel dizionario e_1.

Esempio:

Diz["foobar"]

ritorna il valore associato alla chiave "foobar" nel dizionario Dic.

A.3.14 INCLUSIONE TRA SET O BAG

Vengono forniti nuovi operatori che consentono operazioni di inclusione tra sets o bag di tipi compatibili; in particolare, se e_1 e e_2 sono espressioni che indicano questi sets o bag

$e_1 < e_2$ è vero se e_1 é incluso in e_2 ma non ad esso uguale.

$e_1 <= e_2$ é vero se e_1 é incluso in e_2

A.3.15 RIMOZIONE DI DUPLICATI

E' possibile rimuovere i duplicati da bag, list e array; infatti se *e* é una espressione di tipo *col(t)*, dove *col* puó assumere i valori "set" o "bag", allora *distinct(e)* é un' espressione di tipo *set(t)* il cui valore é la stessa collezione *e*, cosí come si presenta dopo aver rimosso i duplicati.
Se invece *col* é uguale a "list" o a "array", allora *distinct(e)* é ancora un' espressione dello stesso tipo *col(t)*, ottenuta dall' espressione di partenza semplicemente prendendo, in caso di ripetizione, solo il primo valore di ogni elemento della lista.

Esempio:

distinct(list(1,4,2,3,2,4,1))

ritorna list(1,4,2,3).

A.3.16 ABBREVIAZIONI SINTATTICHE

La nuova versione dello standard poi introduce varianti sintattiche di alcuni costrutti, allo scopo di mantenere compatibilitá completa con il DML (*Data*

Manipulation Language) di SQL.
In particolare:

- Se mi trovo in una clausola **select** o **group-by** é consentita, per definire una struttura, anche la sintassi SQL-like

  select projection {, projection} ...
  select ... **group-by** projection {, projection}

  dove *projection* puó essere:

  1. expression **as** identifier
  2. identifier**:** expression
  3. expression

- Per quanto riguarda gli operatori di aggregazione (min, max, count, sum e avg), OQL adotta, accanto alla propria, anche la sintassi SQL. In altre parole, se indichiamo con il termine *aggregate* uno qualunque degli operatori sopra nominati

  select count(∗) from ... é equivalente a count(select ∗ from ...)

  select aggregate(query) from ... é equivalente a aggregate(select query from ...))

  select aggregate(distinct query) from ... é equivalente a aggregate(distinct(select query from ...))

- Se e_1 e e_2 sono espressioni, e_1 é una collezione di valori di un dato tipo t, e_2 é un' espressione di tipo t, e inoltre *relation* é un generico operatore relazionale (=, !=, <, <=, >, >=) allora:

  e_1 *relation* some e_2 é equivalente a exists x in e_2: e_1 *relation* x

  e_1 *relation* any e_2 é equivalente a exists x in e_2: e_1 *relation* x

  e_1 *relation* all e_2 é equivalente a for all x in e_2: e_1 *relation* x

  Esempio:

  10 < some (8,15,7,22) dá come risultato True.

- OQL accetta anche che le stringhe vengano racchiuse tra apici singoli e non doppi (come fa SQL). Ció introduce ambiguitá dal momento che in questo modo vengono indicati in OQL anche i singoli caratteri; tale ambiguitá viene risolta di volta in volta dal contesto.

A.4 JAVA BINDING

Lo standard descrive anche tutta una serie di bindings con linguaggi di programmazione, necessari per realizzare l' implementazione di quanto definito tramite ODL e OQL.
Tra questi la release 2.0 propone, accanto al C++- e Smalltalk-binding anche un Java-binding, di cui ci proponiamo di analizzare le caratteristiche.

A.4.1 JAVA OBJECT MODEL

Il linguaggio Java é comprensivo di un *Object Model* comparabile con quello presentato da ODMG93.
Scopo si questo paragrafo sará perció quello di analizzare le relazioni che esistono tra i due modelli.

CONCETTI NON SUPPORTATI : Il binding con Java non supporta ancora i seguenti concetti:

- Relationship
- Extents
- Keys
- Accesso al metaschema

OGGETTI : Un tipo oggetto in ODMG mappa in un corrispondente tipo oggetto Java.

LITERALS : Per quanto riguarda i letterali ODMG di tipo atomico, essi mappano nei corrispondenti tipi primitivi di Java.
Non é invece prevista l' esistenza di tipi strutturati.

STRUCTURE : La definizione di una *Structure* in ODMG mappa in una classe di Java.

IMPLEMENTAZIONE : Java supporta la definizione di un' interfaccia come di un' entitá indipendente dall' implementazione, e perció non istanziabile.

COLLECTION CLASSES : Per rappresentare oggetti collezione il binding fornisce le seguenti interfacce:


```
public interface Collection {...}
public interface Set extends Collection {...}
public interface Bag extends Collection {...}
public interface List extends Collection {...}
```

Si tratta delle stesse interfacce definite da ODMG93, anche se opportunamente adattate alla sintassi di Java; contengono perció la dichiarazione dei metodi che posso applicare a una qualunque collezione di oggetti.

ARRAY : Per definire gli array Java fornisce tre possibilitá:

1. utilizzare gli array di Java, i quali hanno lunghezza fissa e sono monodimensionali.
2. utilizzare la classe **Vector**, le cui istanze possono essere ridimensionate.
3. utilizzare una classe che implementi l' interfaccia **VArray** definita in ODMG; quest' ultima soluzione ha il vantaggio di creare istanze a cui posso applicare anche i metodi definiti nella classe *Collection*, di cui *Varray* é sottoclasse.

NOMI : I nomi ai vari oggetti vengono assegnati usando i metodi della classe *Database* definita nel Java OML.

ECCEZIONI : Il trattamento delle eccezioni viene fatto utilizzando il meccanismo apposito messo a disposizione da Java.
A livello di standard viene poi fornita tutta una serie di tipi di eccezione, per ognuno dei quali é allegata una descrizione delle sue caratteristiche ed effetti.

A.4.2 JAVA ODL

Si occupa della descrizione dello schema del database in termini di set di classi usando la sintassi Java.

DICHIARAZIONE DI TIPI E ATTRIBUTI

La dichiarazione di un attributo in Java é sintatticamente identica alla dichiarazione di una variabile.
La tabella A.1 descrive l' insieme delle corrispondenze tra i tipi definiti nell' ODMG Object Model e gli equivalenti tipi Java.

Object Model Type	Java Type	Literal?
Long	int (primitive), Integer (class)	yes
Short	short (primitive), Short (class)	yes
Unsigned long	long (primitive), Long (class)	yes
Unsigned short	int (primitive), Integer (class)	yes
Float	float (primitive), Float (class)	yes
Double	double (primitive), Double (class)	yes
Boolean	boolean (primitive), Boolean (class)	yes
Octet	byte (primitive), Integer (class)	yes
Char	char (primitive), Character (class)	yes
String	String	yes
Date	java.sql.Date	no
Time	java.sql.Time	no
TimeStamp	java.sql.TimeStamp	no
Set	interface Set	no
Bag	interface Bag	no
List	interface List	no
Time	array type [] or Vector	no
Iterator	Enumeration	no

Tabella A.1: Mapping dei tipi tra ODMG93 e Java

NOTE :
- I tipi primitivi possono essere rappresentati anche in modo del tutto equivalente dalle loro classi.
- La terza colonna indica se i vari tipi compresi in tabella sono di tipo literal o no.
- Il Java-binding non prevede alcun mappaggio per i tipi **Enum** e **Interval** descritti all' interno dell' ODMG-ODL.

DICHIARAZIONE DI RELAZIONI

Il concetto di relazione non é supportato dal Java-binding.

DICHIARAZIONE DI OPERAZIONI

Le dichiarazioni di operazioni sono sintatticamente identiche alle dichiarazioni di metodi in Java.

A.4.3 JAVA OML

Si occupa della manipolazione delle classi definite tramite Java-ODL.

PERSISTENZA DI OGGETTI E ATTRIBUTI

Le classi di uno schema di un database vengono divise in *persistente-capable* e non.
La persistenza o meno di un oggetto viene determinata al momento della sua creazione, sulla base delle seguenti regole:

- Se l' oggetto non appartiene a una classe *persistente-capable* non potrá mai essere persistente.
- Se l' oggetto appartiene a una classe *persistente-capable*, non é persistente ed é referenziato da un oggetto persistente, lo diventa anch' esso non appena la transazione raggiunge il "commit" (tale comportamento é detto *persistence by reachability*).

Per quanto riguarda invece gli attributi di una classe:

- Un attributo il cui tipo é espresso tramite una classe non *persistente-capable* viene trattato dal sistema come un attributo volatile.
- E' possibile, all' interno di una classe persistente, dichiarare un attributo come volatile usando la parola chiave **transient** messa a disposizione dal linguaggio Java. Ció implica che il valore dell' attributo in questione non venga memorizzato nel database.

  Ad esempio la sintassi

  ```
  public class Person {
      public String name;
      transient Something currentSomething;
      ...}
  ```

  definisce una classe *Person* con un attributo *currentSomething* che non é persistente.
  Quando un oggetto di questa classe é caricato in memoria dal database, il suo attributo dichiarato come volatile é posto da Java al valore di default relativo al tipo a cui appartiene.
  In caso di "abort" della transazione che coinvolge *Person*, *currentSomething* puó o essere lasciato invariato, oppure essere riportato al valore di default.

CANCELLAZIONE DI OGGETTI

Nel Java-binding, esattamente come in Java, non esistono costrutti espliciti per la cancellazione di oggetti.
Un oggetto puó essere eliminato automaticamente dal database se non é né nominato né referenziato da alcun oggetto persistente.

MODIFICA DI OGGETTI

Le modifiche attuate da una transazione su un oggetto vengono riportate sul database solo quando tale transazione raggiunge il "commit".

NOMI DI OGGETTI

L' assegnamento dei nomi ai vari oggetti di un database avviene in modo "piatto"; in altri termini ció sta a significare che i nomi dei vari oggetti devono essere unici all' interno del database.
Le operazioni per manipolare i nomi sono definite nella classe *Database* (vedi par.A.4.3 pag.166).

LOCKING DI OGGETTI

Il "locking" di oggetti é supportato attraverso metodi definiti sulla classe *Transaction* (vedi par.A.4.3 pag.163).

PROPRIETA'

Per accedere ad attributi e relazioni si fa uso della sintassi Java standard.

OPERAZIONI

Le operazioni in Java OML vengono definite come i metodi in linguaggio Java.

TRANSAZIONI

Per prima cosa il binding si preoccupa di adattare alla sintassi Java la classe **Transaction** definita all' interno dell' ODMG Object-Model:

```
public class Transaction {
    Transaction();
    public void join();
    public void leave();
```



```
    public static Transaction current();
    public void begin();
    public boolean isOpen();
    public void commit();
    public void abort();
    public void checkpoint();
    public void lock(Object obj,int mode);
    public static final int READ,UPGRADE,WRITE;
    }
```

Entrando poi nel dettaglio dell' analisi di questi metodi:

Transaction() : crea un oggetto della classe Transaction e lo associa al thread che ha eseguito il metodo.

join() : collega il thread che chiama questo metodo con la transazione grazie alla quale il metodo stesso viene eseguito, interrompendo ogni altra eventuale transazione in corso da parte del thread.

leave() : scollega il thread che ha chiamato il metodo dalla transazione tramite cui tale chiamata é stata eseguita, senza effettuare alcun collegamento sostitutivo.

current() : ritorna la transazione corrente relativa al thread che ha invocato questo metodo (null se il thread non sta eseguendo nessuna transazione).

begin() : fa partire (apre) la transazione tramite cui ho chiamato il metodo. Non é supportata la possibilitá di avere transazioni innestate.

isOpen() : ritorna true se la transazione é aperta, false in caso contrario.

commit() : esegue il commit della transazione.

abort() : abortisce e chiude la transazione.

checkpoint() : esegue il commit della transazione per poi riaprirla.

lock(Object obj,int mode) : esegue un lock di tipo *read*, *write* o *upgrade* su un oggetto.

NOTE :
- Prima di eseguire una qualunque operazione sul database, un thread deve creare un oggetto "Transazione" o associarsi a una transazione giá esistente.

- Un thread puó operare solo sulla sua transazione corrente; se tenta di eseguire operazioni (commit, checkpoint o altro) su una transazione prima di essersi connesso ad essa, viene generata l' eccezione *TransactionNonInProgress*.
- Le transazioni devono essere create e fatte partire esplicitamente; ció infatti non viene fatto automaticamente all' apertura del database, o in seguito al commit o abort di un' altra transazione.
- Una transazione viene implicitamente associata al thread che la ha creata.
- Il metodo *begin* fa partire una transazione. Non é possibile invocare questo metodo piú volte sulla stessa transazione, senza interporre tra due successive chiamate un *commit* o un *abort*. Se ció accade viene eseguita l' eccezione *TransactionNotInProgress* al momento di ogni chiamata successiva alla prima.
- Le operazioni eseguite su una transazione su cui non é ancora stato invocato il *begin* hanno risultato indefinito e possono provocare l' esecuzione dell' eccezione *TransactionNotInProgress*.
- Esistono tre modi in cui threads e transazioni possono essere associati:
  1. Un' applicazione puó avere un solo thread che fa operazioni sul database, il quale é associato ad un' unica transazione.
     E' il caso piú semplice ma anche il piú frequente. Altre applicazioni, contenute su macchine diverse o in spazi di indirizzamento separati, possono accedere allo stesso database tramite altre transazioni.
  2. Possono esistere thread multipli, ognuno dei quali é proprietario di una transazione diversa.
     E' il caso di un servizio utilizzato da piú clienti di una rete. In questo caso il DBMS tratta le transazioni come se si trovassero su spazi di indirizzamento separati.
  3. Piú thread condividono la stessa transazione.
     Questa soluzione é raccomandata solo per applicazioni sofisticate, dal momento che il programmatore deve costruire un controllo della concorrenza utilizzando il meccanismo di sincronismo di Java o altre tecniche.
- Quando una transazione raggiunge il *commit* rilascia tutti i lock che aveva definito; inoltre tutte le modifiche al database (creazione, cancellazione o modifica di oggetti) che la transazione ha fatto vengono memorizzate sul database stesso.

- L' operazione di *checkpoint* esegue come visto il "commit" di una transazione, per poi eseguirne il "restart". Chiaramente l' effetto é quello di rendere persistenti le modifiche eseguite dalla transazione successivamente al precedente *checkpoint*.
- L' operazione di *abort* provoca l' abbandono della transazione senza avere reso persistenti le modifiche che essa ha apportato al database.
- Una transazione ottiene in modo implicito un lock in lettura su un oggetto quando accede ad esso, uno in scrittura invece quando lo modifica.
- Il metodo *lock* modifica il lock che la transazione chiamante ha sull' oggetto in questione, in accordo con i parametri specificati nella dichiarazione del metodo stesso. Se il lock richiesto non puó essere ottenuto viene eseguita l' eccezione *LockNotGranted*.
- Gli oggetti della classe *Transaction* non possono essere resi persistenti, cioé non possono essere memorizzati nel database.

OPERAZIONI SUL DATABASE

Il binding per prima cosa adatta alla sintassi Java la definizione della classe **Database** data nell' ODMG Object-Model.

```
public class Database {
    //modalita' di accesso
    public static final int notOpen = 0;
    public static final int openReadOnly = 1;
    public static final int openReadWrite = 2;
    public static final int openExclusive = 3;

    public static Database open(String name, int accessMode)
                    throws ODMGException;
    public void close() throws ODMGException;
    public void bind(Object object, String name);
    public Object lookup(String name)
                    throws ObjectNameNotFoundException;
    public void unbind(String name)
                    throws ObjectNameNotFoundException;
    }
```

Entrando un pó piú nel dettaglio:

open(String name,int accessMode) : apre il database avente nome *name* in modalitá *accessMode*.
Un database deve essere aperto prima di poter accedere agli oggetti in esso contenuti.
Nel caso si tenti di aprire un database giá aperto viene eseguita l' eccezione *DatabaseOpen*, mentre se il database non esiste viene chiamata *DatabaseNotFound*.

close() : chiude il database correntemente usato.
Se successivamente all' invocazione di questo metodo si tenta di accedere al database, viene eseguita l' eccezione *DatabaseClosed*.

lookup(String name) : consente di accedere ad un oggetto della classe *Database* attraverso il suo nome.
Ritorna "null" se il nome specificato é collegato a *null*; genera invece l' eccezione *ObjectNameNotFound* se il nome non esiste.

bind(Object object,String name) : permette di collegare l' oggetto *object* al nome *name*.
Si noti come questa operazione, invocata su un oggetto volatile, abbia l' effetto di rendere tale oggetto persistente.

unbind(String name) : permette di scollegare un nome dal relativo oggetto; esegue l' eccezione *ObjectNameNotFound* se il nome non esiste.

Per concludere osserviamo che gli oggetti della classe *Database* sono volatili, esattamente come le istanze di *Transaction*, e che un database deve essere aperto prima dell' inizio di ogni transazione che lo modifichi e chiuso dopo la fine della transazione stessa.

A.4.4 JAVA OQL

Vediamo ora come il binding adatta l' OQL a Java.

METODI "COLLECTION QUERY"

L' interfaccia *Collection* vista in precedenza é dotata di un metodo avente la seguente signature:

```
Collection query(String predicate);
```


Tale metodo filtra la collezione che lo invoca usando il predicato passatogli come parametro, ritornando il risultato di tale operazione.
Il predicato viene dato sotto forma di una stringa scritta in modo conforme alla sintassi della clausola **where** di OQL.
Esiste una variabile predefinita **this**, usata all' interno del predicato, per denotare la collezione che si sta filtrando.

Ad esempio, supponendo di avere memorizzato nella variabile *Student* (di tipo *Collection*) un set di studenti, é possibile ricavare tra questi quelli che seguono corsi di matematica:

```
SetOfObject mathematicians;
mathematicians = Students.query                                    (A.10)
    ("exists s in this.takes: s.section-of.name = \"math\" ");
```

Sempre in *Collection*, il metodo

```
public Object selectElement(String predicate);
```

ha lo stesso comportamento di *query*, eccezion fatta per il fatto che puó essere utilizzato solo nel caso in cui il risultato del filtraggio sia un solo oggetto.
Invece il metodo

```
public Enumeration select(String predicate);
```

si differenzia dai due precedenti solo perché ritorna non una collezione ma una *Enumeration* sul risultato della query.

LA CLASSE *OQLQUERY*

La classe *OQLQuery* consente al programmatore di creare una query, passarle i parametri, eseguirla e darne i risultati.

```
class OQLQuery {
    public OQLQuery(){}
    public OQLQuery(String query){...}
    public create(String query){...}
    public bind(Object parameter){...}
    public Object execute() throws ODMGException {...}
    }
```

NOTE :

- I parametri della query devono essere oggetti. Ció significa che per indicare i tipi primitivi devo usare le classi relative (ad es. *Integer* al posto di *int*).
- Anche il risultato della query viene incapsulato in un oggetto, il che significa che se il valore di ritorno é di tipo intero, esso viene messo in un oggetto della classe *Integer*.
- Se la query ritorna una collezione, essa viene messa nella sottoclasse opportuna di *Collection* (ad es. *List* se si tratta di una lista).
- Un parametro, all' interno di una query, é indicato con $i, dove "i" é la posizione del parametro all' interno della lista passata alla query.
- Ogni variabile della lista di parametri deve essere opportunamente "settata" tramite il metodo *bind*. Se cosí non é, al momento della invocazione di *execute* viene eseguita l' eccezione *QueryParameterCountInvalid*. Se uno dei parametri non appartiene al tipo giusto, viene generata l' eccezione *QueryParameterTypeInvalid*.
- Dopo l' esecuzione di una query, la lista dei parametri viene "resettata".

Per chiarire quanto esposto, ci proponiamo ora di analizzare un esempio. Relativamente agli studenti che seguono corsi di matematica (ricavati tramite la A.10), vogliamo realizzare una query per ricavare gli assistenti (TA) il cui salario é maggiore di $50000 e che sono studenti di matematica. In uscita peró la query deve fornire i professori che insegnano a questi studenti.
Assumiamo che esista un set di assistenti chiamato TA.

```
//definizione delle variabili
Bag mathematicians;
Bag assistedProfs;
Double x;
OQLQuery query;

//rintraccio gli studenti che partecipano a corsi di matematica;
//mi servira' come parametro per la query vera e propria.
mathematicians = Students.query(
        "exists s in this.takes: s.section-of.name = \"math\" ");
```



```
//definisco la query di selezione per trovare i professori cercati
query = new OQLQuery(
        "select t.assists.taughtBy from t in TA where t.salary > $1 and t in $2 ");

//memorizzo in "x" il valore 50000 (la usero' come parametro della query)
x = new Double(50000.0);

//associo ai parametri della query di selezione i rispettivi valori
query.bind(x);
query.bind(mathematicians);

//eseguo la query
assistedProfs = (Bag) query.execute();
```

Appendice B

MODELLI DI INGEGNERIA DEL SOFTWARE

B.1 IL MODELLO PHOS

Brevemente, riassumiamo le principali regole PHOS per la rappresentazione dei programmi e delle procedure:

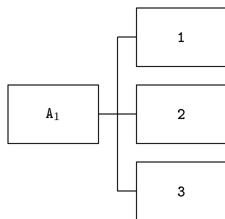

Figura B.1: Esempio di chiamate in sequenza

- La figura B.1 indica che la componente A_1 (ovvero la `procedura A1, indicata con questo tipo di carattere`, o il Modulo A_1) chiama in successione le componenti `1, 2, 3`.

- – nel primo caso della figura B.2 la componente A_2 chiama la componente `1` se si verifica la condizione, altrimenti chiama la componente `2`;

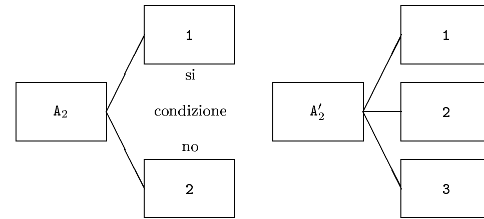

Figura B.2: Esempio di chiamate in alternativa

– nel secondo caso della figura B.2, invece, la componente A'_2 effettua chiamate in alternativa senza che la condizione venga indicata perché è implicita nel nome delle componenti chiamate.

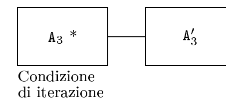

Figura B.3: Esempio di iterazione

- La figura B.3 indica che la componente A_3 itera su se stessa e ad ogni iterazione chiama la componente A'_3 (ad esempio, riceve in input una lista di elementi da passare singolarmente alla procedura chiamata e quindi dopo aver estratto il primo elemento, itera sul resto della lista finché questa non è vuota). A volte, per chiarezza, viene indicata anche la condizione di iterazione.

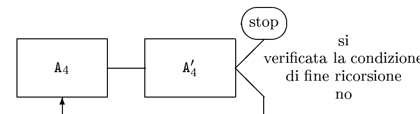

Figura B.4: Esempio di ricorsione

- La figura B.4 indica la ricorsione della componente A_4. Non é indicativo, qui, il fatto che la ricorsione sia tra due elementi: come vedremo, infatti,

in molti casi l' applicazione della procedura ricorsiva é attuata da varie componenti e la ricorsione viene controllata da una componente a parte.

Bibliografia

[1] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, 1987.

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

[3] P. Atzeni, editor. *LOGIDATA$^+$: Deductive Databases with Complex Objects*, volume 701 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg - Germany, 1993.

[4] P. Atzeni and D. S. Parker. Formal properties of net-based knowledge representation schemes. *Data and Knowledge Engineering*, 3:137–147, 1988.

[5] F. Baader and P. Hanschke. A scheme for integrating concrete domains into concept languages. In *12th International Joint Conference on Artificial Intelligence.*, Sydney, Australia, 1991.

[6] J. P. Ballerini, D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. A semantics driven query optimizer for OODBs. In A. Borgida, M. Lenzerini, D. Nardi, and B. Nebel, editors, *DL95 - Intern. Workshop on Description Logics*, volume 07.95 of *Dip. di Informatica e Sistemistica - Univ. di Roma "La Sapienza" - Rapp. Tecnico*, pages 59–62, Roma, June 1995.

[7] F. Bancilhon, C. Delobel, and P. Kanellakis (editors). *Building an Object-Oriented Database Systems: The story of O2*. Morgan Kaufmann, San Mateo, CA, 1992.

[8] G. Di Battista and M. Lenzerini. Deductive entity relationship modeling. *IEEE Trans. on Knowledge and Data Engineering*, 5(3):439–450, June 1993.

[9] H. W. Beck, S. K. Gala, and S. B. Navathe. Classification as a query processing technique in the CANDIDE data model. In *5th Int. Conf. on Data Engineering*, pages 572–581, Los Angeles, CA, 1989.

[10] D. Beneventano and S. Bergamaschi. Subsumption for complex object data models. In J. Biskup and R. Hull, editors, *4th Int. Conf. on Database Theory*, pages 357–375, Heidelberg, Germany, October 1992. Springer-Verlag.

[11] D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Using subsumption in semantic query optimization. In A. Napoli, editor, *IJCAI Workshop on Object-Based Representation Systems*, pages 19–31, August 1993.

[12] D. Beneventano, S. Bergamaschi, and C. Sartori. Taxonomic reasoning with cycles in LOGIDATA$^+$. In P. Atzeni, editor, *LOGIDATA$^+$: Deductive Databases with Complex Objects*, volume 701 of *Lecture Notes in Computer Science*, pages 105–128. Springer-Verlag, Heidelberg - Germany, 1993.

[13] D. Beneventano, S. Bergamaschi, and C. Sartori. Using subsumption for semantic query optimization in OODB. In *Int. Workshop on Description Logics*, volume D-94-10 of *DFKI - Document*, pages 97–100, Bonn, Germany, June 1994.

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

[15] S. Bergamaschi and B. Nebel. The complexity of multiple inheritance in complex object data models. In *Workshop on AI and Objects - IJCAI '91*, Sidney - Australia, August 1991.

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

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

[18] E. Bertino and L. Martino. *Object Oriented Database Systems*. Addison-Wesley, 1993.

[19] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick. CLASSIC: A structural data model for objects. In *SIGMOD*, pages 58–67, Portland, Oregon, 1989.

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

[21] F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In H. W. Schmidt, S. Ceri, and M. Missikoff, editors, *EDBT '88 - Advances in Database Technology*, pages 488–505, Heidelberg, Germany, March 1988. Springer-Verlag.

[22] D. Calvanese and M. Lenzerini. Making object-oriented schemas more expressive. In *PODS - Principles of Database Systems*. ACM Press, 1994.

[23] D. Calvanese, M. Lenzerini, and D. Nardi. A unified framework for class-based formalisms. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR '94 - Int. Conf on Principles of Knowledge Representation and Reasoning*, pages 109–120, Cambridge - MA, 1994. Morgan Kaufmann Publishers, Inc.

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

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

[26] 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.

[27] L. M. L. Delcambre and K. C. Davis. Automatic validation of object-oriented database structures. In *5th Int. Conf. on Data Engineering*, pages 2–9, Los Angeles, CA, 1989.

[28] F. M. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. In J. Allen, R. Fikes, and E. Sandewall, editors, *KR '91 - 2nd Int. Conf on Principles of Knowledge Representation and Reasoning*, pages 151–162, Cambridge - MA, April 1991. Morgan Kaufmann Publishers, Inc.

[29] F. M. Donini, A. Schaerf, and M. Buchheit. Decidable reasoning in terminological knowledge representation systems. In *13th International Joint Conference on Artificial Inteligence*, September 1993.

[30] Francesco M. Donini, Bernhard Hollunder, Maurizio Lenzerini, Alberto Marchetti Spaccamela, Daniele Nardi, and Werner Nutt. The complexity of existential quantification in concept languages. *Artificial Intelligence*, 53(2-3):309–327, February 1992.

[31] T. Finin and D. Silverman. Interactive classification as a knowledge acquisition tool. In L. Kershberg, editor, *Expert Database Systems*, pages 79–90. The Benjamin/Cummings Publishing Company, 1986.

[32] H. Gallaire and J. M. Nicholas. Logic and databases: An assessment. In S. Abiteboul and P. Kanellakis, editors, *ICDT 90*, pages 177–186, Heidelberg, Germany, December 1990. Springer-Verlag.

[33] Yuri Gurevich. The word problem for certain classes of semigroups. *Algebra and Logic*, 5:25–35, 1966.

[34] Philipp Hanschke. Specifying role interaction in concept languages. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. of the Third Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 318–329. Morgan Kaufmann Publishers, 1992.

[35] B. Hollunder, W. Nutt, and M. Schmidt-Schauss. Subsumption algorithms for concept description languages. In *9th ECAI*, pages 348–353, Stockholm, Sweden, 1990.

[36] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *SIGMOD '92*, pages 393–402. ACM, June 1992.

[37] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[38] C. Kung. A tableaux approach for consistency checking. In *IFIP WG 8.1 Conf. on Theoretical and Formal Aspects of Information Systems*, Sitges, Spain, April 1985.

[39] 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.

[40] Zohar Manna and Nachum Dershowitz. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, August 1979.

[41] R. Manthey. Satisfiability of integrity constraints: Reflections on a neglected problem. In *Int. Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1990.

[42] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in prolog. In *CADE 88 - 9th Conference on Automated Deduction*. Springer-Verlag - LNCS, 1988. ECRC Tech. Rep. KB-21, Nov. 87.

[43] B. Nebel. Terminological cycles: Semantics and computational properties. In J. F. Sowa, editor, *Principles of Semantic Networks*, chapter 11, pages 331–362. Morgan Kaufmann Publishers, Inc., San Mateo, Cal. USA, 1991.

[44] Emil L. Post. Recursive unsolvability of a problem of Thue. *Journal of Symbolic Logic*, 12:1–11, 1947.

[45] D. J. Rosenkrantz and H. B. Hunt. Processing conjunctive predicates and queries. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 64–72, Montreal, Canada, October 1980.

[46] M. Schmidt-Schauss and G. Smolka. Attributive concept descriptions with unions and complements. *Artificial Intelligence*, 48(1), 1991.

[47] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, Berlin, Germany, 1986.

[48] J. Sowa, editor. *Principles of Semantic Networks*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

[49] M. Stonebraker. *Object Relational DBMSs*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[50] M. F. van Bommel and G. E. Weddel. Reasoning about equations and functional dependencies on complex objects. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):455–469, June 1994.

[51] Grant E. Weddel. Reasoning about functional dependencies generalized for semantic data models. *ACM Trans. on Database Systems*, 17(1):32–64, March 1992.

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

[ODMG1.1] R. G. G. Cattell, editor. *The Object Database Standard: ODMG93 release 1.1*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[ODMG2.0] R. G. G. Cattell, editor. *The Object Database Standard: ODMG93 release 2.0*. Morgan Kaufmann Publishers, San Mateo, CA, 1998.

[Gar95] Alessandra Garuti. OCDL-Designer: un componente software per il controllo di consistenza di schemi di basi di dati ad oggetti con vincoli di integritá. Tesi di Laurea, Facoltà di Scienze matematiche fisiche e naturali, corso di laurea in scienze delle informazioni dell' Università di Bologna, Bologna, 1995.

[Cor97] Alberto Corni. Odb-dsqo un server www per la validazione di schemi di dati ad oggetti e l'ottimizzazione di interrogazioni conforme allo standard odmg-93. Tesi di Laurea, Facoltà di Scienze dell'Ingegneria, dell' Università di Modena, Modena, 1997.

[Ric98] Stefano Riccio. Elet-designer: uno strumento intelligente orientato agli oggetti per la progettazione di impianti elettrici industriali. Tesi di Laurea, Facoltà di Scienze dell'Ingegneria, dell' Università di Modena, Modena, 1998.

[Unisql] *UNISQL/X User's Manual (release 3.5)*. Austin, Texas 78759-7200, 1996.

[API] *UNISQL/X application programming interface reference manual (release 3.5)*. Austin, Texas 78759-7200, 1996.