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

Progetto e realizzazione di un pre-processore ODL/C++

Relatore
Chiar.mo Prof. Sonia Bergamaschi

Tesi di Laurea di
Ivano Bianchi

Anno Accademico 1996 - 97

Parole chiave:
Pre-Processore
Basi di Dati ad Oggetti
ODMG-93
C++
Regole di Integrità

Ai miei genitori

RINGRAZIAMENTI

Desidero ringraziare il mio relatore, **Prof.ssa Sonia Bergamaschi**, per la sua grande disponibilità.
Ringrazio inoltre la **Dott.ssa Alessandra Garuti** per l'aiuto nei problemi tecnici.

Indice

Capitolo 1

Introduzione

Scopo della Tesi L'idea di base è quella di dotare il progettista di basi di dati ad oggetti di un ambiente software che gli consenta di creare uno schema di basi di dati ad oggetti e lo aiuti nel controllo di correttezza e di consistenza di tale schema.

L'ambiente software dovrà permette di utilizzare sia le potenzialità espressive del C++ (che fonde alle note caratteristiche del C, la possibilità di una programmazione orientata agli oggetti) che le potenzialità del linguaggio di alto livello ODL (Linguaggio di Descrizione di schemi di basi di dati nell'ambito dello Standard ODMG-93[1] per basi di dati ad oggetti). Inoltre, per esprimere in maniera dichiarativa vincoli di integrità su schemi di basi di dati, il progettista potrà utilizzare una versione estesa di ODL (ODL_{Rule}) che permette di definire regole "if then" per ognuna delle classi descritte.

L'ausilio al controllo di correttezza e di consistenza di uno schema di basi di dati verrà realizzato utilizzando il componente OCDL_Designer di ODB-Tools[2] basato su tecniche di intelligenza artificiale e sviluppato presso il Dipartimento di Scienze dell'Ingegneria.

Elemento fondamentale per lo sviluppo di un ambiente software con queste finalità è dotarsi si un pre-processore ODL_{Rule}/C++ in grado di acquisire un sorgente includente istruzioni nei due linguaggi. Nella presente tesi è stato quindi progettato e realizzato un pre-processore ODL_{Rule}/C++ ispirato dall'approccio Standard ODMG-93. Tale standard definisce appunto i principi di collegamento di ODL/OML diretto verso il C++ e definisce un linguaggio standard ODBMS per ODL/OML.

[1] Si veda [Cat94]
[2] Si veda [Gar95]

La presente tesi tratta soltanto il progetto di schemi di basi di dati, quindi non viene approfondita la parte relativa a OML (Linguaggio di Manipolazione degli Oggetti), se non per attingere a dichiarazioni di particolari classi di oggetti fondamentali ivi definite e per l'utilizzo di funzioni e metodi per la gestione delle basi di dati nell'ambito delle regole di integrità.

Nel Capitolo 2 viene riportato un ampio stralcio dello standard ODMG-93 relativo al legame tra C++ e ODL/OML, al quale la presente tesi si ispira. Nel Capitolo 3 si offre una visione riassuntiva dell'ambiente ODB-Tools preesistente a cui il pre-processore ODL_{Rule}/C++ si va ad aggiungere. Nel Capitolo 4 viene presentato il progetto del pre-processore ODL_{Rule}/C++ a grandi linee. Nel Capitolo 5 si entra nei particolari della realizzazione del pre-processore ODL_{Rule}/C++ analizzando ogni componente e ponendo particolare attenzione alla trattazione delle regole di integrità.

Capitolo 2

Lo Standard ODMG-93

2.1 Introduzione

Questo capitolo definisce il legame tra C++ e ODL/OML espresso nello standard[1].

ODL è l'acronimo di Object Definition Language (Linguaggio di Definizione degli Oggetti). Il legame del C++ con ODL è espresso come una libreria di classi ed una estensione della grammatica C++ per la definizione standard delle classi. La libreria di classi fornisce classi e funzioni per implementare i concetti definiti dal modello ad oggetti di ODMG. L'estensione consiste in una sola parola chiave aggiuntiva ed in una sintassi ampliata che aggiunge un supporto dichiarativo per le relazioni nelle dichiarazioni di classi in C++. OML è invece l'acronimo di Object Manipulation Language (Linguaggio di Manipolazione degli Oggetti). Questo è il linguaggio utilizzato per recuperare oggetti dalla base di dati e modificarli. La sintassi e la semantica C++ OML sono quelle C++ standard nel contesto della libreria standard di classi.

ODL e OML specificano soltanto le caratteristiche logiche degli oggetti e delle operazioni utilizzate per manipolarli. Non si occupano della allocazione fisica degli oggetti. Non indirizzano cluster e non gestiscono locazioni di memoria associate con la rappresentazione fisica della allocazione degli oggetti; non accedono neppure a strutture come indici utilizzati per accelerare il recupero degli oggetti. Nel mondo ideale queste cose dovrebbero essere trasparenti al programmatore. Nel mondo reale invece non lo sono. Un insieme addizionale di costrutti chiamati *physical pragma* è definito per permettere al programmatore di inserire alcuni controlli diretti su queste cose, o almeno per renderlo

[1] Si veda [Cat94]

capace di procurarsi "aiuti" per il sottosistema di gestione delle allocazioni fornito come parte dell'esecuzione dell'ODBMS. Le physical pragma esistono sia all'interno di ODL che di OML. Esse sono aggiunte alle definizioni di tipi di oggetti in ODL, espresse come operazioni OML, o mostrate come argomenti opzionali per operazioni definite in OML. Siccome queste pragma non sono in nessun caso un linguaggio a sé stante, ma sono invece un insieme di costrutti aggiunti ad ODL/OML per indirizzare l'implementazione delle locazioni, esse vengono incluse come sottosezioni rilevanti di questo capitolo.

Il capitolo è organizzato nel modo seguente. La sezione 2.2 discute l'ODL. La sezione 2.3 discute l'OML. La sezione 2.4 discute l'OQL - il sottoinsieme distinto di OML che supporta il recupero associativo. Il recupero associativo è il recupero basato sui valori delle proprietà degli oggetti anziché sui loro ID o sui loro nomi. La sezione 2.5 fornisce un programma di esempio. La sezione 2.6 definisce un legame C++ ODL/OML che abbiamo stimato di offrire in futuro quando sarà comune utilizzarlo.

2.1.1 Principi di Progettazione del Linguaggio

I legami per ODL/OML definiti nel capitolo 2 di questa tesi, per la programmazione in uno specifico linguaggio, sono basati su di un principio di base: il programmatore *usa* un solo linguaggio, non due linguaggi separati con confini arbitrari tra loro. Questo principio possiede quattro corollari che sono evidenti nella progettazione del legame C++ definito nel corpo di questo capitolo:

1. C'è un solo sistema unificato di tipi tra il linguaggio di programmazione ed la base di dati; le istanze individuali di questi tipi comuni possono essere persistenti o transienti;

2. Il legame di programmazione tra uno specifico linguaggio (nel nostro caso il C++) e ODL/OML rispetta la sintassi e la semantica del linguaggio di programmazione di base in cui esso viene inserito;

3. Il legame è strutturato come un piccolo insieme di aggiunte al linguaggio di programmazione di base; questo non deve introdurre costrutti di uno specifico sottolinguaggio che duplicano funzionalità già presenti nel linguaggio di base;

4. Le espressioni in OML si compongono liberamente con le espressioni nel linguaggio di programmazione base e vice versa.

2.1.2 Legame col Linguaggio

L'approccio per il legame tra C++ ed il linguaggio dell'ODBMS descritto in questo standard è basato su di un puntatore "Ref-based".

In un approccio Ref-based, il legame C++ mappa il Modello ad Oggetti nel C++ introducendo un insieme di classi che possono avere istanze sia persistenti che transienti. Queste classi sono referenziate informalmente come "classi capaci di persistenza" all'interno del presente capitolo. Queste classi sono distinte dalle normali classi definite attraverso il linguaggio C++, che sono tutte transienti: infatti non sopravvivono all'esecuzione del processo nel quale sono create. Dove è necessario distinguere tra queste due categorie di classi, le prime sono chiamate "classi capaci di persistenza"; alle ultime ci si riferisce come "classi transienti". Per ogni classe T capace di persistenza, una classe ausiliaria Ref<T> viene definita. Le istanze delle classi capaci di persistenza sono allora referenziate usando riferimenti parametrizzati, e.g.,

```
(1) Ref<Professor> profP;
(2) Ref<Department> deptRef;
(3) profP->grant_tenure();
(4) deptRef=profP->dept;
```

L'istruzione (1) dichiara l'oggetto profP come un'istanza del tipo Ref<Professor>. L'struzione (2) dichiara deptRef come un'istanza del tipo Ref<Department>. L'istruzione (3) invoca l'operazione grant_tenure definita nella classe Professor, nella istanza della classe profP. L'operazione (4) assegna il valore dell'attributo dept del professore profP alla variabile deptRef.

Le istanze delle classi capaci di persistenza possono contenere membri di tipi C++, classi definite dall'utente o puntatori a dati transienti. Le applicazioni possono riferirsi a tali membri usando puntatori C++ (*) o riferimenti (&) soltanto durante l'esecuzione di una transazione. Quando è in atto una transazione, i puntatori interni ad oggetti collegati ad allocazioni permanenti saranno settati al valore 0 dall'operazione Transaction::commit.

In questo capitolo useremo i termini seguenti per descrivere le situazioni nell quali lo standard viene formalmente considerato indefinito o permette ad un implementatore di prendere decisioni su specifiche implementazioni nel rispetto dello standard stesso.

I termini sono:

Indefinito: il comportamento non viene specificato dallo standard. Le implementazioni hanno completa libertà (possono fare tutto o niente), ed il comportamento non ha necessità di essere documentato dall'implementatore o dal venditore.

Definito sull'Implementazione: il comportamento è specificato da ogni implementatore/venditore. All' implementatore/venditore è permesso prendere decisioni su specifiche implementazioni riguardanti il funzionamento. Comunque, il funzionamento deve essere ben definito e completamente documentato e pubblicato come parte dello standard relativo alla implementazione del venditore.

La figura 2.1 mostra la gerarchia di linguaggi in questione, attaverso un preprocessore, la compilazione ed i passi di link che generano una applicazione eseguibile.

2.1.3 Mappaggio del Modello ad Oggetti ODMG sul C++

Sebbene il C++ fornisca un modello di dati molto potente e vicino a quello presentato nel capitolo 2, vale la pena cercare di spiegare più precisamente come i concetti introdotti nel capitolo 2 vengano mappati in concreto nei costrutti C++.

Oggetti e Letterali

Un tipo di oggetto ODMG viene mappato in una classe C++. A seconda di come una classe C++ viene istanziata, il risultato può essere un oggetto ODMG oppure un letterale ODMG. Un oggetto C++ che sia membro interno di una classe viene trattato come un letterale ODMG. Questo è spiegato dal fatto che un blocco di memoria viene inserito nell'oggetto incluso ed appartiene interamente ad esso. Per esempio, non si può copiare l'oggetto incluso senza fare una copia dell'oggetto membro allo stesso tempo. In questo senso l'oggetto membro non può essere considerato come se avesse una identità, infatti esso si comporta come un letterale immutabile.

Strutture

La nozione di *struttura* del Modello ad Oggetti viene mappata nei costrutti C++ di *struct* e *class* interni in una classe.

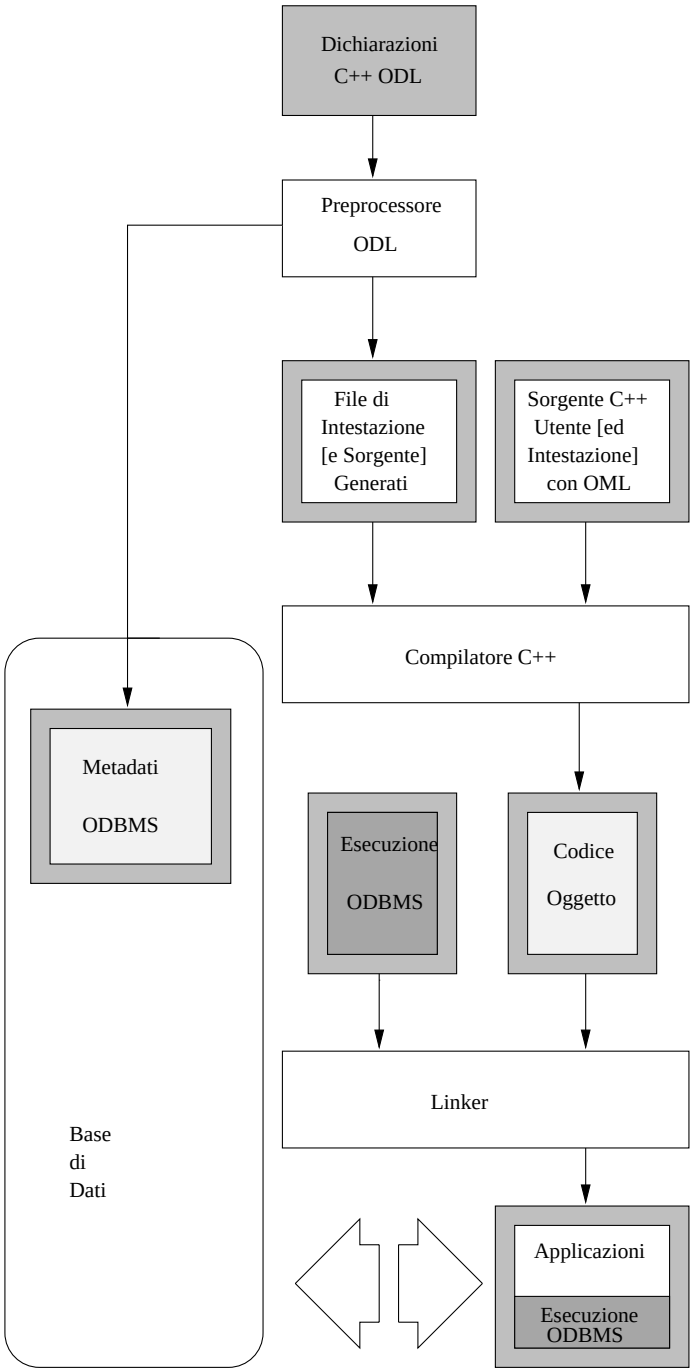

Figura 2.1: Gerarchia di Linguaggio

Implementazione

Il C++ possiede implicitamente la nozione di suddivisione della definizione di una classe in due parti: la sua interfaccia (parte pubblica) e la sua implementazione(membri privati e protetti e definizione delle funzioni). Comunque, in C++ è possibile specificare soltanto una implementazione per una data classe.

Classi Collezione

Il modello ad oggetti ODMG comprende dei generatori di tipi collezione, tipi collezione ed istanze collezione. I generatori di tipi collezione sono rappresentati come *classi template* in C++. I tipi collezione sono rappresentati come classi collezione e le istanze collezione sono rappresentate come istanze di queste classi collezione. Illustriamo queste tre categorie:

```
template<class T> class Set:public Collection<T>{...};
class Ship {...};
Set<Ref<Ship>> Cunard_Line;
```

Set<T> è una classe template collezione. Set<Ref<Ship>> è una classe collezione, e Cunard_Line è una particolare collezione, una istanza della classe Set<Ref<Ship>>.
La gerarchia sottotipo/supertipo di tipi collezione definita nel modello ODMG è direttamente trasportata nel C++. Il tipo Collection<T> è una classe astratta in C++ con nessuna istanza diretta. Risulta istanziabile soltanto attraverso le sue classi derivate. La sola differenza tre le classi collezione nel legame C++ e le loro controparti nello standard sono le seguenti:

- Le operazioni con un nome nel Modello ad Oggetti sono mappate in funzioni membro in C++.
- Per alcune operazioni, il legame C++ include sia le funzioni con nome che le operazioni sovrapposte, e.g, Set::union_with ha anche la forma operator+=. L'istruzione s1.union_with(s2) e s1+=s2 sono funzionalmente equivalenti.
- Le operazioni che nello standard restituivano un valore booleano sono modellate come funzioni membro che restituiscono un int nel legame C++. Questo viene fatto per venire incontro alla convenzione C che un intero di valore zero significa falso ed ogni altro valore significa vero.
- Le operazioni di create e delete definite nello standard sono state rimpiazzate coi costruttori e distruttori C++.

Array

Il C++ fornisce una sintassi per creare ed accedere ad una sequenza indicizzabile di oggetti. Questa è stata scelta per mappare parzialmente la collezione Array di ODMG. Per complementarla, viene fornita una classe Varray che implementa un array il cui estremo superiore può variare nel tempo.

Relazioni

Le relazioni non sono direttamente supportate dal C++. Comunque, esse sono supportate in ODMG generando automaticamente dei metodi C++ che attraversano le relazioni stesse.

La relazione in sé stessa viene implementata come un riferimento (relazione uno-a-uno) o come una collezione (relazione uno-a-molti) all'interno dell'oggetto.

Estensioni

Le estensioni non sono supportate direttamente dal C++. Il programmatore è responsabile della definizione di una collezione e della scrittura dei metodi per il suo mantenimento.

Chiavi

La dichiarazioni di chiavi non è supportata dal C++.

Nomi

Nel legame col C++ un oggetto può acquisire solo un nome, mentre nel modello ODMG più di un nome può riferirsi allo stesso oggetto. Un tentativo di associare più di un nome allo stesso oggetto causa un errore.

Amministrazione della Base di Dati

Alcune operazioni riguardanti l'amministrazione di una base di dati sono omesse intenzionalmente dal legame col C++. Per esempio il legame col C++ non fornisce un mezzo per creare una base di dati né per definire un indice su una collezione.

2.1.4 Utilizzo degli Aspetti del Linguaggio C++

Prefisso

I nomi globali nell'interfaccia ODMG avranno il prefisso _odb. L'intenzione è di evitare le collisioni con altri nomi nello spazio dei nomi.

Manipolazione delle Eccezioni

Si pianifica di usare la manipolazione delle eccezioni C++ in quanto essa risulta un mezzo generalmente disponibile nella maggior parte degli ambienti C++. Per il momento, quando una condizione di eccezione viene incontrata, una routine di errore sarà richiamata. Entrambe queste soluzioni saranno identificate come *origine di errori*.

Identificatore del Pre-processore

Un identificatore del pre-processore, __ODMG_93__, è definito per compilazioni sotto condizioni.

2.2 C++ ODL

Questa sezione definisce il Linguaggio di Definizione degli Oggetti del C++. il C++ ODL fornisce una descrizione dello schema della base di dati come un insieme di classi di oggetti - comprendente i loro attributi, relazioni ed operazioni - in uno stile sintattico che consiste della porzione dichiarativa di un programma C++. Le istanze di queste classi possono essere manipolate attraverso il C++ OML.

Di seguito si ha un esempio di dichiarazione del tipo Professor. Si noti l'introduzione della nuova parola chiave **inverse** nella dichiarazione di relazioni.

```
class Professor: public Persistent_Object {
public:
// proprieta':
   int                 age;
   int                 id_number;
   String              office_number;
   String              name;
   Ref<Department>     dept       inverse professors;
```

```
   Set<Ref<Student>>  advises   inverse Student::advisor;
// operazioni:
   void               grand_tenure();
   void               assign_course(Course &);
private:
        ...
};
```

questa sintassi per la dichiarazione di una classe per C++ ODL è identica alla dichiarazione di una classe in C++ tranne che essa include la dichiarazione di attributi e cammini attraverso relazioni. La dichiarazione di attributi mappa su di un insieme ristretto di dichiarazioni di attributi di una classe in C++. Le dichiarazioni di cammini attraverso relazioni sono sintatticamente distinti dagli attributi di una classe dalla presenza della clausola **inverse**.

Gli attributi statici delle classi non sono contenuti all'interno di ogni istanza ma sono classi staticamente allocate. Questi attributi statici non sono allocati nella base di dati, ma sono supportati per le classi capaci di persistenza. I supertipi sono specificati utilizzando la sintassi standard C++ all'interno dell'intestazione della classe, e.g., class Professor: public Person. Sebbene questa specifica utilizzi attributi e metodi pubblici per facilità e brevità, i corrispondenti attributi e metodi privati o protetti sono supportati.

2.2.1 Dichiarazioni di Attributi

Le dichiarazioni di attributi sono sintatticamente identiche alle dichiarazioni dei membri dato in C++. Siccome nozioni di attributi come oggetti non sono ancora definite ed incluse in questo standard, gli attributi ed i membri dato non sono e non possono essere distinti. In questo standard, un attributo non può avere proprietà (e.g, unità di misura) e non c'è modo di specializzare le operazioni get_value e set_value definite sul tipo (e.g., per scatenare un evento quando un valore è cambiato).

La sintassi e la semantica C++ standard per la definizione delle classi sono supportate. Comunque le implementazioni non necessitano di supportare i seguenti tipi dato all'interno delle classi persistenti:

- unioni
- campi di bit (bit fields)

- riferimenti (&)

come membri. Unioni e campi di bit pongono dei problemi quando si supportano ambienti eterogenei. Le semantiche dei riferimenti sono tali che essi sono inizializzati soltanto alla loro creazione: tutte le susseguenti operazioni sono dirette all'oggetto referenziato. I riferimenti all'interno di oggetti persistenti non posso essere re-inizializzati quando sono portati dalla base di dati in memoria e l'indirizzo che essi contengono da quando sono stati inizializzati potrebbe, in generale, essere non valido. Tutti i puntatori all'interno di un oggetto persistente sono trattati come puntatori a dati transienti. Un insieme di classi speciali è definito all'interno delle specifiche di ODMG per contenere riferimenti agli oggetti persistenti. In aggiunta a tutti i tipi di dato primitivi, eccettuati quelli accennati sopra, le strutture e le classi di oggetti possono essere attributi di una classe. Ci sono parecchi tipi strutturati di letterali che sono fornite. Questi includono:

- String
- Interval
- Date
- Time
- Timestamp

esempi:

```
class Student: public Persistent_Object {
public:
   String          name;
   Date            birth_date;
   Phone_Number    dorm_phone;
   struct{
           int     PO_box;
           String  university;
           String  city;
           String  state;
           String  zip_code;
   } university_address;
   List<String>    favorite_friend;
};
```

L'attributo name prende una String come suo valore. L'attributo dorm_phone prende un tipo Phone_Number, definito dall'utente, come suo valore. L'attributo university_address prende una struttura. L'attributo favorite_friends prende una List di String come suo valore.

Le sezioni seguenti contengono le descrizioni dei tipi letterali forniti.

String

La classe seguente definisce un tipo dato letterale da usarsi come attributo stringa. Resta inteso che questa classe è usata soltanto per allocare stringhe nella base di dati, quindi non è certamente una classe generica di stringhe con tutte le funzionalità di una classe di stringhe normalmente utilizzata per stringhe transienti in una applicazione.

Inizializzazione, assegnamento, copia e conversione da e alle stringhe di caratteri del C++ sono supportate. Gli operatori di comparazione sono definiti su String per comparare questa con un'altra String o una stringa di caratteri del C++. Si può anche accedere ad un elemento nella String tramite un indice, inoltre è possibile determinare la lunghezza della String.

Definizione:

```
class String {
 public:
                          String();
                          String(const String &);
                          String(const char *);
                          ~String();
      String &            operator=(const String &);
      String &            operator=(const char *);
                          operator const char *() const;
      char &              operator[](unsigned long index);
      unsigned long       length() const;
  friend int              operator==(const String &sL,
                                     const String &sR);
  friend int              operator==(const String &sL,
                                     const char *pR);
  friend int              operator==(const char *pL,
                                     const String &sR);
```

```
  friend int              operator!=(const String &sL,
                                     const String &sR);
  friend int              operator!=(const String &sL,
                                     const char *pR);
  friend int              operator!=(const char *pL,
                                     const String &sR);
  friend int              operator<(const String &sL,
                                    const String &sR);
  friend int              operator<(const String &sL,
                                    const char *pR);
  friend int              operator<(const char *pL,
                                    const String &sR);
  friend int              operator<=(const String &sL,
                                     const String &sR);
  friend int              operator<=(const String &sL,
                                     const char *pR);
  friend int              operator<=(const char *pL,
                                     const String &sR);
  friend int              operator>(const String &sL,
                                    const String &sR);
  friend int              operator>(const String &sL,
                                    const char *pR);
  friend int              operator>(const char *pL,
                                    const String &sR);
  friend int              operator>=(const String &sL,
                                     const String &sR);
  friend int              operator>=(const String &sL,
                                     const char *pR);
  friend int              operator>=(const char *pL,
                                     const String &sR);

  };
```

Interval

La classe Interval è utilizzata per rappresentare un intervallo di tempo. Viene anche usata per effettuare operazioni aritmetiche sulle classi Date, Time e Timestamp. Un giorno è il più ampio componente di tempo contenuto in un Interval. Questa classe corrisponde all'intervallo day-time come è definito nello standard SQL.

Inizializzazione, assegnamento, aritmetica e funzioni di comparazione sono definite nella classe, come anche le funzioni membro per accedere alle componenti di tempo del suo valore corrente.

Definizione:

```
class Interval {
public:
                Interval(int day=0, int hour=0, int min=0,
                          float sec=0.0);
                Interval(const Interval &);
     Interval & operator=(const Interval &);
     int        day() const;
     int        hour() const;
     int        minute() const;
     float      second() const;
     int        is_zero() const;
     Interval & operator+=(const Interval &);
     Interval & operator-=(const Interval &);
     Interval & operator*=(int);
     Interval & operator/=(int);
     Interval   operator-() const;
friend Interval operator+(const Interval &L,
                          const Interval &R);
friend Interval operator-(const Interval &L,
                          const Interval &R);
friend Interval operator*(const Interval &L, int R);
friend Interval operator*(int L, const Interval &R);
friend Interval operator/(const Interval &L, int R);
friend int      operator==(const Interval &L,
                           const Interval &R);
friend int      operator!=(const Interval &L,
                           const Interval &R);
friend int      operator<(const Interval &L,
                          const Interval &R);
friend int      operator<=(const Interval &L,
                           const Interval &R);
friend int      operator>(const Interval &L,
                          const Interval &R);
friend int      operator>=(const Interval &L,
```

```
                                const Interval &R);

};
```

Date

La classe Date contiene una rappresentazione di una data composta da anno, mese e giorno. Può essere inizializzata ed usata anche nella rappresentazione giuliana di una data. Fornisce inoltre enumerazioni per denotare giorni della settimana ed i mesi.

Inizializzazione, assegnamento, aritmetica e funzioni di comparazione sono fornite. Le implementazioni possono avere funzioni aggiuntive disponibili per supportare conversione da ed al tipo usato dal sistema operativo per rappresentare una data. Sono fornite funzioni per accedere alle componenti di una data. Ci sono anche funzioni per determinare il numero del giorno in un mese, ecc. La funzione statica current restituisce la data corrente. Le funzioni next e previous avanzano la data al prossimo giorno della settimana specificato.

Definizione:

```
class Date {
public:
       enum Weekday{
            Sunday=0,    Monday=1,    Tuesday=2,
            Wednesday=3, Thursday=4,  Friday=5,
            Saturnday=6
       };
       enum Month {
            January=1, February=2, March=3, April=4,
            May=5, June=6, July=7, August=8, September=9,
            October=10, November=11, December=12
       };

                        Date();
                        Date(unsigned short year,
                              unsigned short julian_day);
                        Date(unsigned short year,
                              unsigned short month=1,
                              unsigned short day=1);
```

```
                        Date(const Date &);
                        Date(const Timestamp &);
    Date &              operator=(const Date &);
    Date &              operator=(const Timestamp &);
    unsigned short      year() const;
    unsigned short      month() const;
    unsigned short      day() const;
    unsigned short      day_of_year() const;
    Weekday             day_of_week() const;
    int                 is_leap_year() const;
static int              is_leap_year(unsigned short year);
static Date             current();
    Date &              next(Weekday);
    Date &              previous(Weekday);
    Date &              operator+=(const Interval &);
    Date &              operator+=(int ndays);
    Date &              operator++();
    Date                operator++(int);
    Date &              operator-=(const Interval &);
    Date &              operator-=(int ndays);
    Date &              operator--();
    Date                operator--(int);
friend Date             operator+(const Date &L,
                                  const Interval &R);
friend Date             operator+(const Interval &L,
                                  const Date &R);
friend Date             operator-(const Date &L,
                                  const Interval &R);
friend int              operator==(const Date &L,
                                   const Date &R);
friend int              operator!=(const Date &L,
                                   const Date &R);
friend int              operator<(const Date &L,
                                  const Date &R);
friend int              operator<=(const Date &L,
                                   const Date &R);
friend int              operator>(const Date &L,
                                  const Date &R);
friend int              operator>=(const Date &L,
                                   const Date &R);
    int                 is_between(const Date &,
```

```
                                    const Date &) const;
  friend int              overlaps(const Date &psL,
                                   const Date &peL,
                                   const Date &psR,
                                   const Date &peR);
  friend int              overlaps(const Timestamp &psL,
                                   const Timestamp &peL,
                                   const Date &psR,
                                   const Date &peR);
  friend int              overlaps(const Date &psL,
                                   const Date &peL,
                                   const Timestamp &psR,
                                   const Timestamp &peR);
  static int              days_in_year(unsigned short year);
      int                 days_in_year() const;
  static int              days_in_month(unsigned short yr,
                                        unsigned short month);
      int                 days_in_month() const;
  static int              is_valid_date(unsigned short year,
                                        unsigned short month,
                                        unsigned short day);
      int                 is_valid() const;
  };
```

Le funzioni overlaps prendono due periodi (inizio e fine), ogni periodo denotato da un tempo iniziale e da uno finale, e determinano se i due periodi di tempo si sovrappongono. La funzione is_between determina se il valore della Date è interno ad una dato periodo.

Time

La classe Time viene usata per denotare un'orario specifico, che è internamente allocato come Ora Media di Greenwich (Greenwich Mean Time = GMT). Inizializzazione, assegnamento, aritmetica ed operatori di comparazione sono definiti. Ci sono inoltre funzioni per accedere ad ognuno dei componenti di un valore di orario. Le implementazioni possono avere funzioni aggiuntive disponibili a supportare conversioni da e al tipo usato dal sistema operativo per rappresentare il tempo.

L'enumerazione Time_Zone è resa disponibile per denotare una specifica zona temporale. Le zone temporali sono numerate in accordo col numero di ore che deve essere aggiunto o sottratto dall'orario locale per ottenere l'ora di Greenwich (Inghilterra GMT). Così il valore di GMT è 0. Una Time_Zone chiamata GMT6 indica un'orario di 6 ore avanti rispetto a GMT e così 6 deve essere sottratto da esso per ottenere GMT. Al contrario, GMT_8 significa che l'orario è 8 ore indietro rispetto a GMT (leggere il sottotratto come un meno). Una zona temporale di default viene mantenuta ed è inizialmente settata la valore della zona temporale locale. Risulta possibile cambiare la zona temporale di default come pure resettarla al valore locale.

Definizione:

```
class Time {
public:
       enum Time_Zone {
            GMT   = 0, GMT12 =12, GMT_12=-12,
            GMT1  = 1, GMT_1 =-1, GMT2  = 2, GMT_2 = -2,
            GMT3  = 3, GMT_3 =-3, GMT4  = 4, GMT_4 = -4,
            GMT5  = 5, GMT_5 =-5, GMT6  = 6, GMT_6 = -6,
            GMT7  = 7, GMT_7 =-7, GMT8  = 8, GMT_8 = -8,
            GMT9  = 9, GMT_9 =-9, GMT10 =10, GMT_10=-10,
            GMT11 =11, GMT_11=-11,
            USeastern=-5, UScentral=-6, USmountain=-7,
            USpacific=-8
       };

static void             set_default_Time_Zone(Time_Zone);
static void             set_default_Time_Zone_to_local();
                        Time(unsigned short hour,
                              unsigned short minute,
                              float sec);
                        Time(unsigned short hour,
                              unsigned short minute,
                              float sec, short tzhour,
                              short tzminute);
                        Time(const Time &);
                        Time(const Timestamp &);
     Time &             operator=(const Time &);
     Time &             operator=(const Timestamp &);
```

```
      unsigned short     hour() const;
      unsigned short     minute() const;
      float              second() const;
      short              tz_hour() const;
      short              tz_minute() const;
static Time              current();
      Time &             operator+=(const Interval &);
      Time &             operator-=(const Interval &);
friend Time              operator+(const Time &L,
                                   const Interval &R);
friend Time              operator+(const Interval &L,
                                   const Time &R);
friend Interval          operator-(const Time &L,
                                   const Time &R);
friend Time              operator-(const Time &L,
                                   const Interval &R);
friend int               operator==(const Time &L,
                                    const Time &R);
friend int               operator!=(const Time &L,
                                    const Time &R);
friend int               operator<(const Time &L,
                                   const Time &R);
friend int               operator<=(const Time &L,
                                    const Time &R);
friend int               operator>(const Time &L,
                                   const Time &R);
friend int               operator>=(const Time &L,
                                    const Time &R);
friend int               overlaps(const Time &psL,
                                  const Time &peL,
                                  const Time &psR,
                                  const Time &peR);
friend int               overlaps(const Timestamp &psL,
                                  const Timestamp &peL,
                                  const Time &psR,
                                  const Time &peR);
friend int               overlaps(const Time &psL,
                                  const Time &peL,
                                  const Timestamp &psR,
                                  const Timestamp &peR);
```

```
};
```

Le funzioni Overlaps prendono due periodi, ognuno denotato da un orario iniziale e da uno finale, e determinano se due periodi di tempo si sovrappongono.

Timestamp

Un Timestamp consiste di data e orario.

Definizione:

```
class Timestamp {
public:
                    Timestamp(unsigned short year,
                              unsigned short monty=1,
                              unsigned short day=1,
                              unsigned short hour=0,
                              unsigned short minute=0,
                              float sec=0.0);
                    Timestamp(const Date &);
                    Timestamp(const Date &, const Time &);
                    Timestamp(const Time &);
   Timestamp &      operator=(const Timestamp &);
   Timestamp &      operator=(const Date &);
   Date &           date();
   const Time &     time() const;
   unsigned short   year() const;
   unsigned short   month() const;
   unsigned short   day() const;
   unsigned short   hour() const;
   unsigned short   minute() const;
   float            second() const;
   short            tz_hour() const;
   short            tz_minute() const;

static Timestamp    current();
   Timestamp &      operator+=(const Interval &);
   Timestamp &      operator-=(const Interval &);
friend Timestamp    operator+(const Timestamp &L,
                              const Interval &R);
```

```
  friend Timestamp   operator+(const Interval &L,
                              const Timestamp &R);
  friend Timestamp   operator-(const Timestamp &L,
                              const Interval &R);
  friend int         operator==(const Timestamp &L,
                               const Timestamp &R);
  friend int         operator!=(const Timestamp &L,
                               const Timestamp &R);
  friend int         operator<(const Timestamp &L,
                              const Timestamp &R);
  friend int         operator<=(const Timestamp &L,
                               const Timestamp &R);
  friend int         operator>(const Timestamp &L,
                              const Timestamp &R);
  friend int         operator>=(const Timestamp &L,
                               const Timestamp &R);
  friend int         overlaps(const Timestamp &psL,
                              const Timestamp &peL,
                              const Timestamp &psR,
                              const Timestamp &peR);
  friend int         overlaps(const Timestamp &psL,
                              const Timestamp &peL,
                              const Date &psR,
                              const Date &peR);
  friend int         overlaps(const Date &psL,
                              const Date &peL,
                              const Timestamp &psR,
                              const Timestamp &peR);
  friend int         overlaps(const Timestamp &psL,
                              const Timestamp &peL,
                              const Time &psR,
                              const Time &peR);
  friend int         overlaps(const Time &psL,
                              const Time &peL,
                              const Timestamp &psR,
                              const Timestamp &peR);
  };
```

2.2.2 Dichiarazioni di Cammini Attraverso le Relazioni

Le relazioni non hanno delle definizioni separate sintatticamente. Comunque i *cammini trasversali* sono definiti all'interno dei corpi delle definizioni di ognuno dei due tipi di oggetti che hanno un ruolo nella relazione. Per esempio, se c'è una relazione Advises uno-a-molti tra Professor e Student, allora il cammino trasversale Advises è definito all'interno della definizione del tipo dell'oggetto Professor ed il cammino trasversale Advisor è definito all'interno della definizione del tipo dell'oggetto Student.

La dichiarazione di un cammino attraverso le relazioni è simile alla dichiarazione di un attributo, ma con le seguenti differenze. Il codominio di una relazione ha un cammino attraverso la relazione. Una dicharazione di cammino trasversale assomiglia ad una dichiarazione di attributo, ma è seguita dalla parola chiave **inverse** e dal nome del cammino attraverso la relazione nella classe all'altro capo della relazione. La dichiarazione dell'attributo deve essere del tipo Ref<T>, Set<Ref<T>>, or List<Ref<T>> per alcune classi persistenti T. Il cammino di attraversamento delle relazioni designato dopo la parola chiave **inverse** deve essere dichiarato nella classe T. Il percorso trasversale ad ogni capo della relazione deve riferirsi, nella sua clausola inversa, al corrispondente membro dell'altra classe coinvolta nella relazione. Il nome del cammino trasversale inverso può essere un nome completo. Lo studio della relazione negli esempi seguenti renderà questo concetto chiaro.

Esempi:

```
class Department: public Persistent_Object{
public:
  Set<Ref<Professor>> professors inverse Professor::dept;
};

class Professor: public Persistent_Object{
public:
 Ref<Department>   dept     inverse Department::professors;
 Set<Ref<Student>> advises inverse  Student::advisor;
};

class Student: public Persistent_Object{
public:
```

```
 Ref<Professor>    advisor  inverse Professor:advisees;
 Set<Ref<Course>>  classes  inverse Course::students_enrolled;
};

class Course: public Persistent_Object{
public:
 Set<Ref<Student>> students_enrolled inverse Student::classes;
};
```

L'integrità referenziale delle relazioni bidirezionali è mantenuta automaticamente dall ODBMS. Se una relazione esiste tra due oggetti ed uno degli oggetti viene cancellato, la relazione non viene più considerata come esistente ed il cammino trasversale sarà alterato per rimuovere la relazione.

2.2.3 Dichiarazioni di Relazioni Unidirezionali

Per adeguarsi ad una pratica comune nella programmazione C++, una forma degenerata di relazione viene permessa: quella che specifica un cammino a senso unico. In questo caso, la clausola **inverse** viene omessa. Si osservi che la sintassi per la dichiarazione di relazioni unidirezionali è attualmente indistinguibile dalla sintassi C++ utilizzata per dichiarare membri di tipo Ref o qualunque sottoclasse di Collection che referenzi ad un oggetto persistente.

Nel caso specifico di relazioni unidirezionali, è l'applicazione ad essere responsabile del mantenimento consistente della relazione quando l'oggetto (oggetti) alla fine del cammino è cancellato (sono cancellati). Nessun mantenimento di integrità referenziale viene fatto per le relazioni unidimensionali.

Tale dichiarazione di relazione unidirezionale viene permessa sia all'interno di una classe che all'interno di un attributo di una struttura.

Esempi:

```
struct Responsible {
    String          dept;
    Ref<Employee>   e;
    Date            due_date;
};
class Order {
```

```
public:
 Set<Ref<Client>>   who;
 String             what;
 Responsible        contact;
};
```

2.2.4 Dichiarazioni di Operazioni

Le dichiarazioni di operazioni in C++ sono sintatticamente identiche alle dichiarazioni dei **metodi di una classe**. Ad esempio, si guardi grant_tenure e assign_course definiti per la classe Professor nella sezione 2.2.

2.3 C++ OML

Questa sezione descrive il legame C++ per l'OML. Un principio guida nel progetto del C++ ODL è che la sintassi usata per creare, cancellare, identificare, leggere/assegnare i valori di proprietà ed invocare operazioni su di un oggetto persistente dovrebbero essere, per quanto possibile, senza differenze rispetto a quella usata per gli oggetti di vita più breve. Una singola espressione può liberamente mischiare riferimenti ad oggetti persistenti e transienti.

L'obiettivo a lungo termine è che niente possa essere fatto con oggetti persistenti che non possa essere fatto anche con oggetti transienti; in realtà questo standard tratta gli oggetti persistenti e transienti in maniera un poco differente. Le interrogazioni e la consistenza delle transazioni vengono applicate soltanto agli oggetti persistenti.

2.3.1 Creazione, Cancellazione, Modifica e Riferimenti ad Oggetti

Gli oggetti possono essere creati, cancellati e modificati. Gli oggetti sono creati in C++ OML usando l'operatore **new**, che è ridefinito per accettare argomenti aggiuntivi che specificano il tempo di vita dell'oggetto. Una pragma opzionale di allocazione permette al programmatore di specificare in che modo gli oggetti appena allocati debbano essere disposti nel rispetto agli altri oggetti.

La variabile statica Database::transient_memory, della classe Database, è definita per consentire a librerie che creino oggetti di essere usate indeifferentemente per creare sia oggetti transienti che persistenti. Questa variabile può essere usata come valore tra gli argomenti nell'operatore **new** della base di dati per creare oggetti con tempo di vita transiente.

```
static const Database * const Database::transient_memory;
```

Le forme in ODMG dell'operatore C++ **new** sono:

```
(1) void * operator new(size_t size, const char* typename=0);
(2) void * operator new(size_t size,
                        const Ref<Persistent_Object &clustering,
                        const char* typename=0);
(3) void * operator new(size_t size, const Database *database,
                        const char* typename=0);
```

Questi operatori possiedono il campo d'azione dei Persistent_Object per le cui implementazioni viene introdotta la classe Persistent_Object. (1) viene usata per creare oggetti transienti derivati dai Persistent_Object. (2) e (3) creano oggetti persistenti. In (2) l'utente specifica che l'oggetto appena creato dovrebbe essere piazzato "vicino" al cilindro esistente per gli oggetti. L'esatta interpretazione di "vicino" è definita dall'implementazione. Una possibile interpretazione potrebbe essere "nella stessa pagina se possibile". In (3) l'utente specifica che l'oggetto appena creato dovrebbe essere posto nella base di dati specificata, ma nessuna allocazione viene specificata.

L'argomento size, che appare come primo argomento in ogni riga, rappresenta la dimansione della rappresentazione dell'oggetto. Viene determinata dal compilatore come una funzione della classe della quale il nuovo oggetto è una istanza, e non passato come un argomento esplicito dal programmatore.

L'argomento opzionale tipename serve per specificare un nome per un oggetto e viene usato in alcune implementazioni.

Le implementazioni devono fornire al minimo queste tre forme e possono rendere disponibili delle varianti, con parametri aggiuntivi. Ogni parametro aggiuntivo deve avere un valore di default. Questo permette ad applicazioni che non usano i parametri aggiuntivi di essere portabili. Tipici usi dei parametri aggiuntivi dovrebbero essere richieste di allocazione in memoria

distribuita o di passaggio di una String per il nome della classe.

Esempi:

```
    Database *yourDB, *myDB;  // si assume che vengano
                              // inizializzati in modo corretto
(1)  Ref<Schedule> temp_sched1= new Schedule;
(2)  Ref<Professor> prof2=new(yourDB, "Professor") Professor;
(3)  Ref<Student> student1=new(myDB) Student;
(4)  Ref<Student> student2=new(student1) Student;
(5)  Ref<Student> temp_student=
                   new(Database::transient_memory) Student;
```

L'istruzione (1) crea l'oggetto transiente temp_sched1. Le istruzioni (2)-(4) creano oggetti persistenti. L'istruzione (2) crea una nuova istanza della classe Professor nella base di dati yourDB. L'istruzione (2) illustra anche il passaggio del nome di una classe come valore del terzo parametro opzionale. Forme particolari di questo parametro possono essere richieste da alcune implementazioni. L'istruzione (3) crea una nuova istanza della classe Student nella base di dati myDB. L'istruzione (4) fa la stessa cosa, ma specifica che il nuovo oggetto student2 dovrebbe essere posto vicino a student1. L'istruzione (5) crea l'oggetto transiente temp_student.

Cancellazione di Oggetti Gli oggetti, appena creati, possono essere cancellati in C++ OML usando il metodo Ref::delete_object. La cancellazione di un oggetto è permanente, sottomessa alla transazione coinvolta. L'oggetto viene rimosso dalla memoria e, se era persistente, dalla base di dati. L'istanza Ref esiste ancora in memoria ma il suo valore di riferimento è indefinito. Un tentativo di accedere a questo oggetto da qualunque istanza Ref fallisce e genera un errore.

Esempi:

```
Ref<anyType> obj_ref;
... // si setta obj_ref in modo che referenzi
    // un oggetto persistente
obj_ref.delete_object();
```

Il C++ richiede che l'operando di **delete** sia un puntatore, così la cancellazione di un oggetto referenziato da un Ref richiede che una funzione membro

tipo delete_object sia definita. L'invocazione di delete su di un Ref risulta essere un errore di compilazione, infatti Ref non è un puntatore.

Modifica di Oggetti Lo stato di un oggetto viene modificato cambiando le sue proprietà o invocando operatori su di esso. La modifica di oggetti persistenti viene resa visibile dagli altri utenti della base di dati quando viene eseguita una operazione di `commit` su di una transazione contenente la modifica.

Gli oggetti persistenti che sono stati distrutti o modificati devono comunicare, durante l'esecuzione del processo ODBMS, il fatto che il loro stato è cambiato. L'ODBMS modificherà la base di dati con questi nuovi dati quando la transazione verrà effettivamente eseguita. Il cambiamento di oggetti, definita nella sezione 2.3.4, viene usato nel modo seguente:

```
obj_ref->mark_modified();
```

La chiamata alla funzione mark_modified è inclusa nei metodi del costruttore e distruttore per le classi capaci di persistenza, i.e., per la classe Persistent_Object. Lo sviluppatore dovrebbe includere la chiamata in ogni altro metodo che modifica oggetti persistenti.

Per convenienza, il programmatore può omettere le chiamate a mark_modified sugli oggetti le cui classi siano state compilate usando un apposito parametro di un pre-processore C++ OML; il sistema individuerà automaticamente quando gli oggetti sono modificati. Nel caso di default, le chiamate a mark_modified sono richieste perché in alcune implementazioni sarebbe meglio che fosse il programmatore stesso a richiamare esplicitamente mark_modified. Comunque, ogni volta che un oggetto persistente viene modificato da un metodo fornito esplicitamente dalle classi ODMG, la chiamata a mark_modified non è necessaria in quanto questo viene fatto automaticamente.

Riferimenti ad Oggetti Gli oggetti, siano essi persistenti oppure no, possono referenziare altri oggetti attraverso riferimenti ad oggetti. In C++ OML i riferimenti ad oggetti sono istanze della classe template Ref<T> (vedi sezione 2.3.5). Tutti gli accessi ad oggetti persistenti sono fatti attraverso metodi

definiti nelle classi Ref, Persistent_Object e Database. L'operatore di dereferenziazione -> viene usato per accedere ai membri dell'oggetto persistente "puntato" da un dato riferimento ad oggetto. In che modo un riferimento ad oggetto viene convertito in un puntatore C++ all'oggetto è una definizione dell'implementazione.

Una operazione di dereferenziazione su un riferimento ad oggetto garantisce sempre che l'oggetto referenziato viene restituito oppure viene generato un errore. Il comportamento di un riferimento è il seguente. Se un riferimento ad oggetto si riferisce ad un oggetto persistente che esiste ma non è in memoria quando una dereferenziazione viene fatta, viene automaticamente prelevato da disco, mappato in memoria e restituito assieme al risultato della dereferenziazione. Se l'oggetto referenziato non esiste, un errore relativo viene generato. I riferimenti ad oggetti transienti funzionano esattamente allo stesso modo (o almeno in superficie).

Ogni riferimento ad oggetti può essere settato ad un riferimento nullo o *cancellato* per indicare un riferimento che non si riferisce a nessun oggetto.

Le regole per quando un oggetto durante la sua vita possa riferire ad un altro di un altro tempo di vita sono una diretta estensione delle regole C++ per le sue due forme di oggetti transienti - processi e procedure limitrofe. Un oggetto può sempre riferirsi ad un altro oggetto di tempo vita maggiore. Un oggetto può riferirsi ad un oggetto di tempo di vita minore solo mentre l'oggetto di tempo di vita più breve esiste.

Un oggetto persistente viene prelevato da disco alla sua attivazione. Risulta responsabilità dell'applicazione inizializzare i valori di ogni puntatore ad oggetto che punti ad oggetti transienti. Infatti, quando un oggetto persistente viene coinvolto, l'ODBMS setta i tutti suoi Ref interni ad oggetti transienti ed i puntatori interni al valore 0.

Nomi di Oggetti Un'applicazione per basi di dati generalmente inizierà processando tramite l'accesso uno o più oggetti critici e procederà da essi. Questi oggetti sono in un certo senso oggetti "radice", che guidano alla rete di interconnessione degli altri oggetti. La possibilità di nominare oggetti e recuperarli più tardi tramite il nome rende facilitate le capacità di inizializzazione. La denominazione di oggetti è anche conveniente in molte altre situazioni.

C'è un solo scopedname, in una base di dati; cosìtutti i nomi in una particolare base di dati sono unici. Un nome non è esplicitamente definito come un attributo di un oggetto. Le operazioni per manipolare i nomi sono definite nella classe Database nella sezione 2.3.8.

2.3.2 Proprietà

Attributi Il C++ OML utilizza lo standard C++ per accedere agli attributi. Per esempio, assumendo che `prof` sia stato inizializzato per referenziare un professore e noi vogliamo modificare il suo `id\_number`:

```
prof->id_number=next_id;
cout << prof->id_number;
```

La modifica di un valore di un attributo viene considerata come una modifica dell'intera istanza oggetto. Occorre chiamare mark_modified per l'oggetto prima di compiere la transazione.

Il legame col C++ permette alle classi persistenti di includere classi C++ ed altre classi persistenti. Tuttavia, gli oggetti interni non sono considerati "oggetti indipendenti" e non possiedono un'identità di oggetto personale. Agli utenti non viene permesso di manipolare un Ref ad un oggetto interno. Esattamente come un attributo, la modifica di un oggetto interno viene considerata come la modifica dell'intera istanza oggetto contenitore e mark_modified per l'oggetto contenitore deve essere chiamata prima di compiere la transazione.

Relazioni L'ODL specifica quali relazioni esistono tra classi di oggetti. Creazione, attraversamento e cancellazione di relazioni fra istanze sono definite in C++ OML. I cammini trasversali a-uno e a-molti sono supportati da OML. L'integrità delle relazioni viene mantenuta dall'ODBMS.

I diagrammi seguenti mostreranno graficamente gli effetti dell'aggiunta, modifica e cancellazione di relazioni fra le classi. Ad ogni diagramma viene dato un nome che riflette la cardinalità e l'effetto sulla relazione. Il nome comincerà con 1-1, 1-m o m-m per denotare la cardinalità e terminerà con

Figura 2.2: 1-1N: Nessuna relazione

N(Nessuna relazione), A(Aggiunta di una relazione) o M(Modifica di una relazione). Quando una relazione viene cancellata, si tornerà nello stato che non possiede relazioni (N).

Sebbene i cammini di attraversamento delle relazioni siano disegnati all'interno di un'istanza di una classe, questo non significa che l'implementazione inserisca fisicamente il cammino di attraversamento della relazione come un membro all'interno della classe. Una linea continua viene disegnata per indicare una operazione esplicita eseguita dal programma e una linea tratteggiata mostra l'effetto collaterale eseguito automaticamente dall' ODBMS per mantenere l'integrità referenziale.

Assumiamo che la seguente relazione 1-1 esista tra le classi A e B:

```
class A {
    Ref<B> rb   inverse B::ra;
};
class B {
    Ref<A> ra   inverse A::rb;
};
```

Si noti che la classe A e la classe B potrebbero essere la stessa classe. In ognuno dei diagrammi mostrati, ci sarà un'istanza di A chiamata a oppure aa ed un'istanza di B chiamata b o bb. Nello scenario 1-1N di figura 2.2, non ci sono relazioni tra a e b.

Quindi aggiungiamo una relazione tra a e b attraverso

```
a.rb=&b;
```

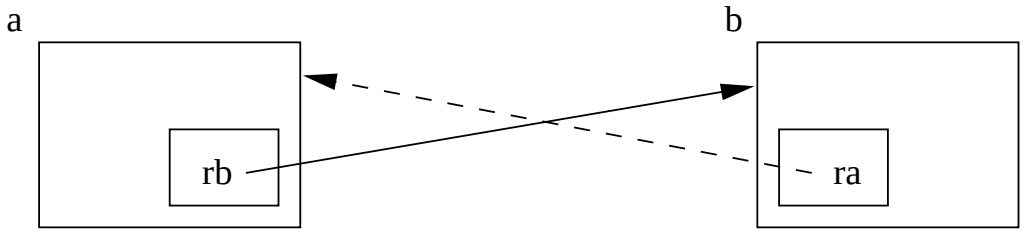

Figura 2.3: 1-1A: Aggiunta di una relazione

il risultato è quello mostarato in figura 2.3.
La freccia continua indica l'operazione specificata dal programma e quella tratteggiata mostra che operazione viene svolta automaticamente dall'ODBMS.

Prendiamo ora in considerazione il diagramma precedente (figura 2.3) e supponiamo che rappresenti lo stato corrente della relazione tra a e b. Se il programma esegue l'istruzione

```
a.rb.clear();
```

il risultato sarebbe l'assenza di relazioni come mostrato in figura 2.2.

Riprendiamo nuovamente in considerazione la relazione mostrata in figura 2.3. Se noi adesso eseguiamo

```
a.rb=&bb;
```

otteniamo il risultato mostrato in figura 2.4.

Si noti che b.ra non referenzia più a e bb.ra viene settato automaticamente come riferimento ad a.

Il cammino attraverso la relazione rb si comporta come se fosse un attributo della classe A il cui tipo potrebbe essere la classe seguente che definisce le operazioni disponibili su rb. Le operazioni che hanno effetto sulla relazione possiedono un commento che indica quale dei precedenti scenari 1-1 si avrebbe come risultato dopo che l'operazione fosse stata eseguita.

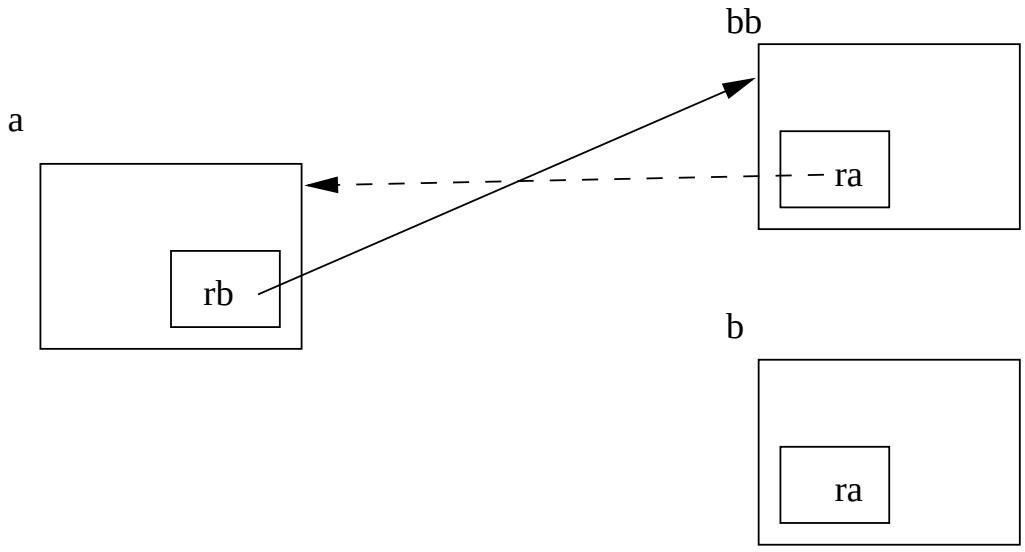

Figura 2.4: 1-1M: Modifica di una Relazione

```
class A_to_B {
public:
               A_to_B()                    // N
               A_to_B(const A_to_B &f);    // A, se
                                           //r non e nullo
               A_to_B(const Ref<B> &f);    // A, se
                                           //r non e nullo
               A_to_B(B *r);               // A, se
                                           //r non e nullo
               ~A_to_B()                   // N

// Attraversamenti della relazione
     Ref<B>    get() const;
               operator Ref<B>() const;
               operator Ref_Any<B>() const;
     B *       operator->() const;
     B &       operator*() const;

// Instaurazione della relazione
     A_to_B & set(const A_to_B &);       // A or M
     A_to_B & set(const Ref<B> &);       // A or M
     A_to_B & set(B *);                  // A or M
```

```
        A_to_B & operator=(const A_to_B &); // A or M
        A_to_B & operator=(const Ref<B> &); // A or M
        A_to_B & operator=(B *);            // A or M
        A_to_B & set(const A_to_B &);       // A or M

    // Cancellazione della relazione
        void       clear();                 //N
        int        is_null() const;
        void       delete_object()          //N

    friend int     operator==(const A_to_B &, const A_to_B &);
    friend int     operator==(const A_to_B &, const Ref<B> &);
    friend int     operator==(const Ref<B> &, const A_to_B &);
    friend int     operator==(const A_to_B &, const Ref_Any &);
    friend int     operator==(const Ref_Any &, const A_to_B &);
    friend int     operator==(const A_to_B &, const B *);
    friend int     operator==(const B *, const A_to_B &);
    friend int     operator!=(const A_to_B &, const A_to_B &);
    friend int     operator!=(const A_to_B &, const Ref<B> &);
    friend int     operator!=(const Ref<B> &, const A_to_B &);
    friend int     operator!=(const A_to_B &, const Ref_Any &);
    friend int     operator!=(const Ref_Any &, const A_to_B &);
    friend int     operator!=(const A_to_B &, const B *);
    friend int     operator!=(const B *, const A_to_B &);

};
```

Ogni qualvolta l'operando di inizializzazione o assegnamento rappresenta un riferimento nullo, il risultato sarà una assenza di relazioni come in 1-1N. Nel caso dell'assegnamento, se ci fosse una relazione, verrebbe rimossa. Se la relazione è attualmente nulla (is_null restituirebbe vero), un assegnamento potrebbe aggiungere una relazione, a meno che anche l'operando di assegnamento fosse nullo.

Se esiste già una relazione con un oggetto, il fare un assegnamento modifica la relazione come in 1-1M. Se l'operando di assegnamento è nullo, la relazione esistente viene rimossa.

Quando un oggetto coinvolto in una relazione viene cancellato, tutte le rela-

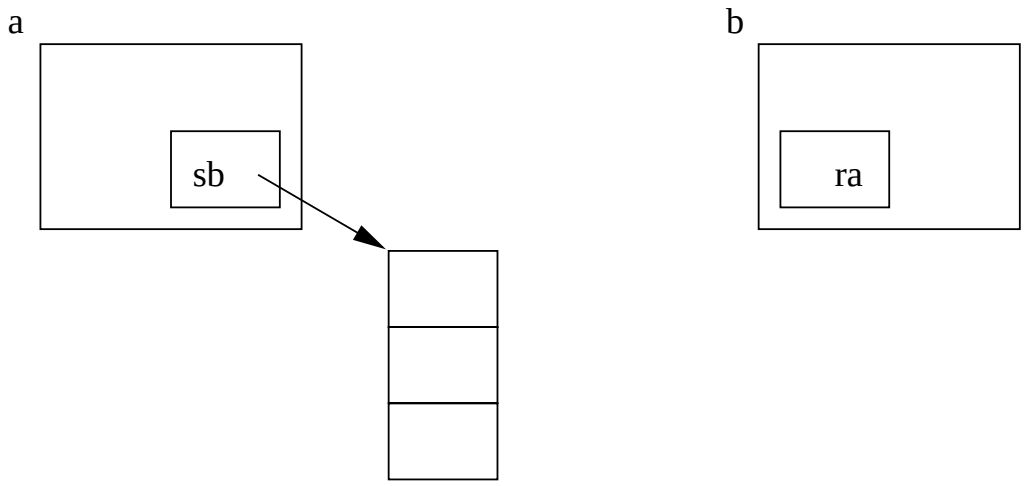

Figura 2.5: 1-mN: Nessuna relazione

zioni in cui l'oggetto era coinvolto saranno rimosse.

Ci sono altre due cardinalità da considerare: uno-a-molti e molti-a-molti. Dopo aver dato una descrizione grafica degli effetti delle operazioni su di una relazione di entrambe queste cardinalità, sarà specificata l'interfaccia del cammino di attraversamento della relazione, sia esso disordinato oppure posizionale.

Consideriamo una relazione disordinata di insieme uno-a-molti tra le classi A e B:

```
class A {
    Set<Ref<B>> sb   inverse B::ra;
};
class B {
    Ref<A> ra        inverse A::sb;
};
```

Supponiamo di avere le seguenti istanze a e b senza relazioni (figura 2.5)
a.sb possiede tre elementi, ma essi si riferiscono ad istanze di B diverse da b.

Ora supponiamo di aggiungere la relazione tra a e b eseguendo l'istruzione

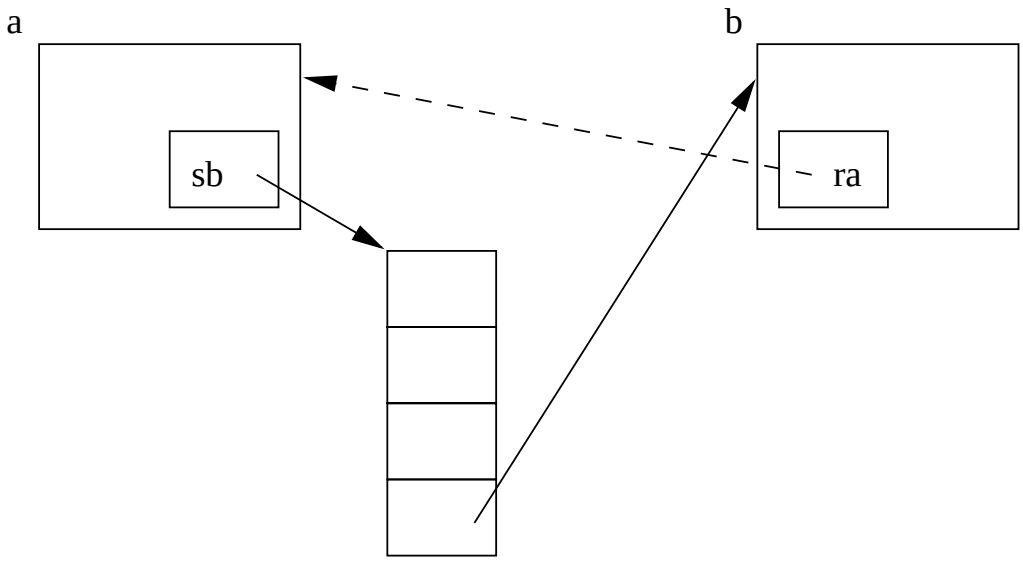

Figura 2.6: 1-mA: Aggiunta di una relazione

```
a.sb.insert_element(&b);
```

Il risultato è quello di figura 2.6:
Il cammino trasversale b.ra viene settato automaticamente per riferenziare a. Se invece noi eseguiamo l'istruzione

```
b.ra=&a;
```

un elemento verrà automaticamente aggiunto ad a.sb per riferirsi a b. Ma è necessario eseguire nel programma soltanto una delle due operazioni, l'ODBMS automaticamente genera il cammino trasversale inverso.

Data la situazione mostrata in figura 2.6, se noi eseguiamo

```
a.sb.remove_element(&b)      oppure      b.ra.clear();
```

il risultato sarebbe che la relazione tra a e b verrebbe rimossa e lo stato di a e b sarebbe nuovamente quello indicato in figura 2.5.

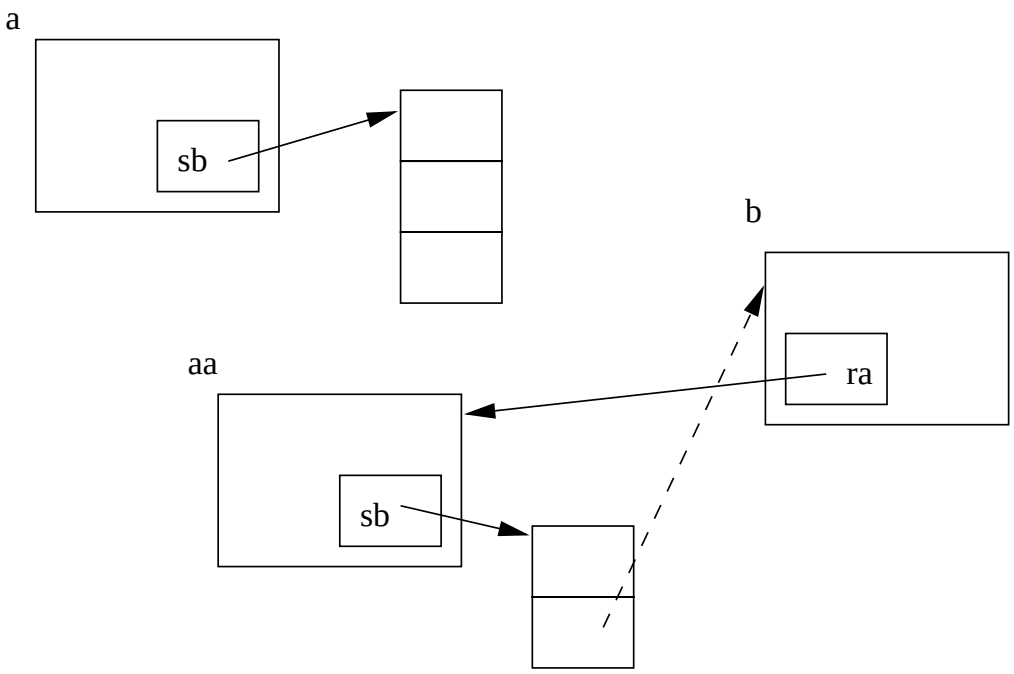

Figura 2.7: 1-mM: Modifica di una relazione

Adesso assumiamo di avere una relazione tra a e b come mostrato in figura 2.6. Se eseguiamo la seguente istruzione:

```
b.ra=&aa;
```

il risultato viene evidenziato in figura 2.7
Dopo che l'istruzione è stata eseguita, b.ra si riferisce ad aa e come effetto collaterale, l'elemento interno ad a.sb che referenziava b è stato rimosso mentre un elemento è stato aggiunto ad aa.sb in modo che si riferisca a b.

La terza cardinalità di una relazione da considerare è molti-a-molti. Supponiamo di avere la seguente relazione tra A e B.

```
class A {
    Set<Ref<B>> sb   inverse B::sa;
};
class B {
    Set<Ref<A>> sa   inverse A::sb;
```

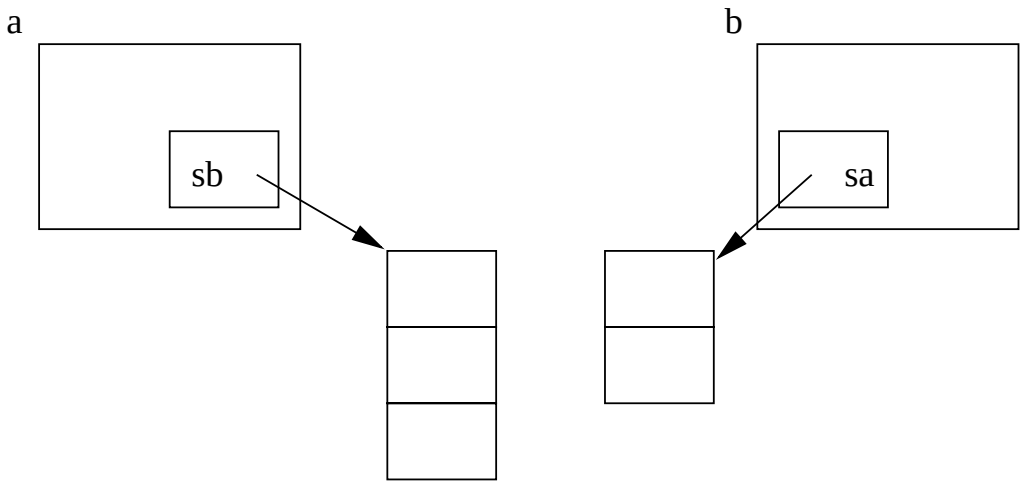

Figura 2.8: m-mN: Nessuna relazione

```
};
```

Inizialmente, non ci sono relazioni tra le istanze a e b, benché a e b abbiano relazioni con altre istanze (figura 2.8).

La seguente istruzione aggiunge una relazione tra a e b.

```
a.sb.insert_element(&b);
```

Il risultato è quello di figura 2.9:
In aggiunta al fatto che un elemento che referenzi b è stato aggiunto ad a.sb, c'è un elemento che referenzia a inserito automaticamente in b.sa.

Eseguendo ancora

```
a.sb.remove_element(&b)     oppure     b.sb.remove_element(&a)
```

risulta che la relazione tra a e b è stata rimossa e si è ritornati alla situazione m-mN.

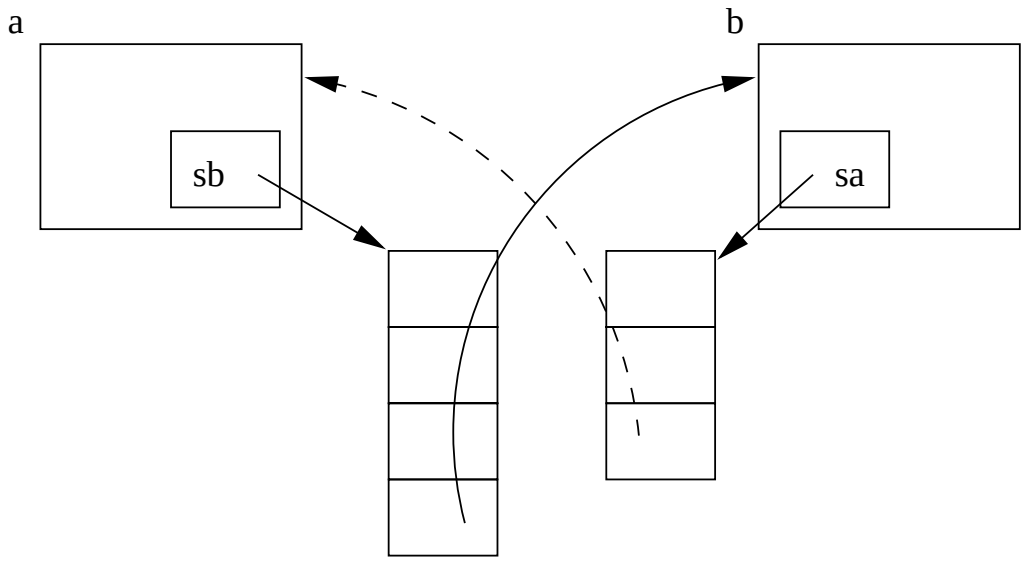

Figura 2.9: m-mA: Aggiunta di una relazione

Per ultimo consideriamo la modifica ad una relazione molti-a-molti. Assumiamo lo stato precedente mostrato in m-mA, ed assumiamo anche che sb rappresenti una relazione posizionale. La seguente istruzione modifica una relazione esistente tra a e b, facendo in modo che a venga posto in relazione con bb.

```
a.sb.replace_element_at(&bb,3);
```

Il risultato viene mostrato in figura 2.10:
Il risultato di questa operazione è che l'elemento in b.sa che referenziava a è stato rimosso ed un elemento che referenzia a è stato aggiunto a bb.sa.

Lo stesso concetto "generatore" applicato per cammini trasversali a-molti viene anche applicato per cammini trasversali a-uno. Nella sintassi C++ ODL, un cammino trasversale a-molti assomiglia ad un'istanza di una semplice classe collezione (dove ogni classe collezione è attualmente una classe template). Tuttavia, per implementare la semantica delle relazioni, il cammino trasversale deve essere convertito in un attributo C++ OML che è un'istanza di una classe che "capisce" il tipo di cammino trasversale che è stato dichiarato.

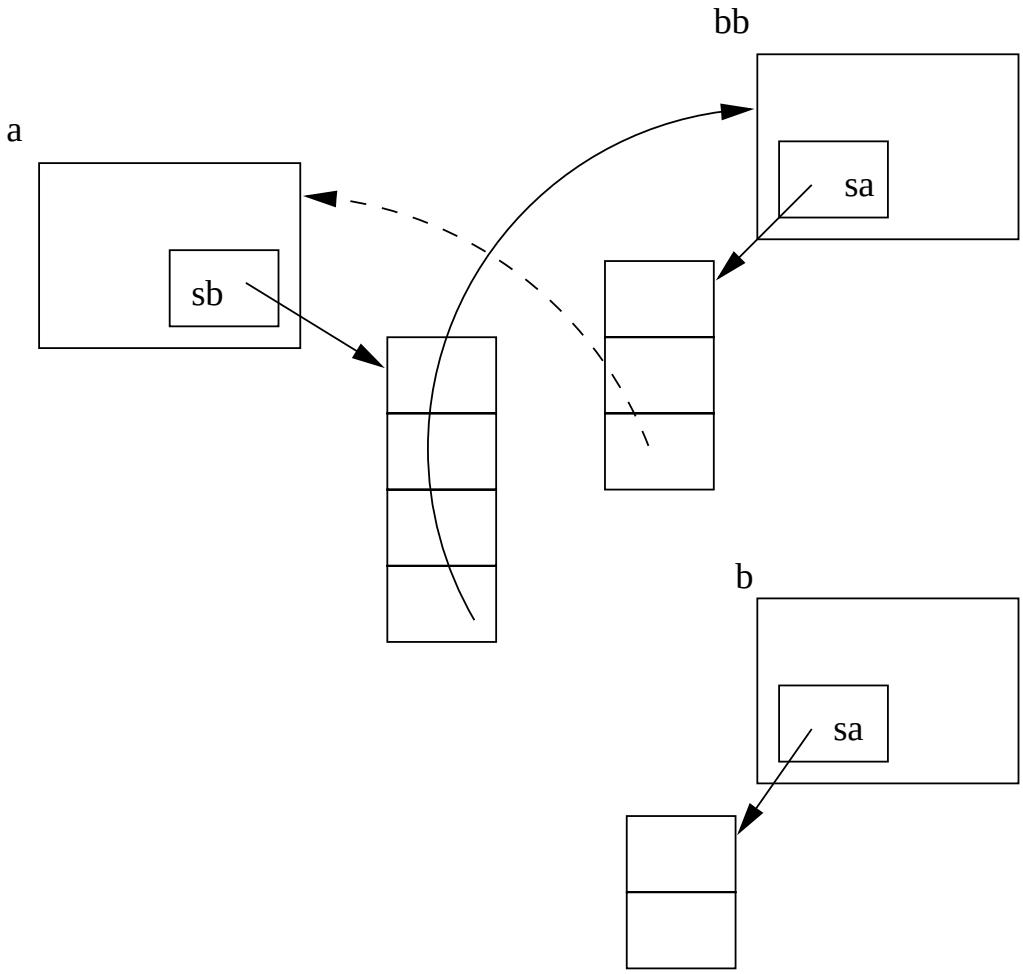

Figura 2.10: m-mM: Modifica di una relazione

Con relazioni uno-a-molti e molti-a-molti, l'insieme delle operazioni permesse è basato sul fatto che la relazione sia un insieme disordinato o posizionale.

Data una relazione a-molti tra A e B come in

```
class A {
    Set<Ref<B>> sb   inverse B::ra;
};
```

sb si comporta come se fosse un attributo della classe A del tipo della classe seguente A_to_Bs che definisce le operazioni disponibili su sb. Ogni operazione che altera la relazione ha un commento che indica quale scenario si avrebbe come risultato dell'operazione.

```
class A_to_Bs {
public:
                    A_to_Bs();                          // N
                    A_to_Bs(const A_to_Bs &s);          // A,
                             // N se s \‘e nullo
                    A_to_Bs(const Set<Ref<B>> &s);      // A,
                             // N se s \‘e nullo
                    ~A_to_Bs();                         // N

  Set<Ref<B>>       get() const;
  A_to_Bs &         set(const A_to_Bs &);               // M,
                             // A se nessun antecedente
  A_to_Bs &         set(const Set<Ref<B>> &);           // M,
                             // A se nessun antecedente
  A_to_Bs &         operator=(const A_to_Bs &);         // M,
                             // A se nessun antecedente
  A_to_Bs &         operator=(const Set<Ref<B>> &);     // M,
                             // A se nessun antecedente
  Iterator<Ref<B>>  create_iterator() const;

  unsigned long     cardinality() const;
  int               is_empty() const;
  int               contains_element(const Ref<B> &) const;
  void              insert_element(const Ref<B> &);   // A
```

```
    void              remove_element(const Ref<B> &);  // N
    void              remove_all();           // N per tutti

};
```

Le inizializzazioni e gli assegnamenti che hanno un argomento del tipo A_to_Bs oppure Set<Ref<B>> sono molto più coinvolte rispetto ai semplici diagrammi precedenti perché coinvolgono l'esecuzione delle corrispondenti operazioni per ogni elemento dell'insieme rispetto al farlo per un solo elemento. La funzione remove_all rimuove ogni membro della relazione, togliendo anche ogni riferimento all'indietro per ognuno dei membri referenziati. Se gli operatori di assegnamento hanno un argomento che rappresenta un insieme vuoto, l'assegnamento avrà lo stesso effetto della funzione remove_all.

Queste specifiche supportano il concetto di iteratore tramite una generica classe template chiamata Iterator, ma non ci sono certamente ragioni per cui una sottotemplate specializzata di questa classe non possa essere definita. L'iteratore viene utilizzato per attraversare tutti i membri della relazione.

La classe List rappresenta una collezione *posizionale*, mentre la classe Set è semplicemente una collezione disordinata. Assumendo una relazione a-molti posizionale tra una classe A ed una classe B:

```
class A {
    List<Ref<B>> sb   inverse B::ra;
};
```

sb si comporta come se fosse un attributo della classe A del tipo della classe seguente A_to_Bp che definisce le operazioni disponibili su sb.

```
class A_to_Bp {
public:
                    A_to_Bp();                          // N
                    A_to_Bp(const A_to_Bp &s);          // A,
                            // N se s \'e nullo
                    A_to_Bp(const List<Ref<B>> &s);     // A,
                            // N se s \'e nullo
                    ~A_to_Bp();                         // N

  List<Ref<B>>       get() const;
```

```
  A_to_Bp &         set(const A_to_Bp &);            // M,
                            // A se nessun antecedente
  A_to_Bp &         set(const List<Ref<B>> &);       // M,
                            // A se nessun antecedente
  A_to_Bp &         operator=(const A_to_Bp &);      // M,
                            // A se nessun antecedente
  A_to_Bp &         operator=(const List<Ref<B>> &); // M,
                            // A se nessun antecedente
  Iterator<Ref<B>>  create_iterator() const;

// Test di esistenza
  int               is_empty() const;
  unsigned long     cardinality() const;
  int               contains_element(const Ref<B> &) const;
  int               find_element(const Ref<B> &,
                                 unsigned long &pos) const;

// Recupero di relazioni
  Ref<B>            retrieve_element_at(
                            unsigned long pos) const;
  Ref<B>            operator[](int position) const;
  Ref<B>            retrieve_first_element() const;
  Ref<B>            retrieve_last_element() const;

// Instaurazione di relazioni
  void    insert_element(const Ref<B> &);           // A
  void    insert_element_first(const Ref<B> &);     // A
  void    insert_element_last(const Ref<B> &);      // A
  void    insert_element_after(const Ref<B> &,      // A
                               unsigned long pos);
  void    insert_element_before(const Ref<B> &,     // A
                                unsigned long pos);
  void    replace_element_before(unsigend long pos, // M
                                 const Ref<B> &);

// Rimozione di relazioni
  void    remove_element(const Ref<B> &);           // N
  void    remove_first_element();                   // N
  void    remove_last_element();                    // N
  void    remove_element_at(unsigned long pos);     // N
  void    remove_all();                             // N
```

```
};
```

Naturalmente le classi possono anche definire dei metodi specifici del venditore. Ma, più importante, essi devono gestire ogni integrità referenziale che sia richiesta ed ogni caratteristica di una relazione specifica del venditore che sia dichiarate nel C++ ODL. Una implementazione conforme può scegliere di usare una sola classe "base" per tutti i tipi di relazione o può definire classi di relazioni specializzate.

Esempi:

```
Ref<Professor> p;
Ref<Student> Sam;
Ref<Department> english_dept;
// Inizailizzazione dei riferimenti di p, Sam e english_dept
p->dept=english_dept;  // creazione di una relazione 1:1
p->dept.clear();       // cancella la relazione
p->advisees.insert_element(Sam); // aggiunge Sam
                       // all'insieme degli studenti
                       // consigliati da p; ha lo stesso
                       // effetto di ' Sam->advisor=p '
p->advisees.remove_element(Sam); // rimuove Sam dell'insieme
                                 // degli studenti
                                 // consigliati da p ed
                                 // inoltre cancella
                                 // Sam->advisor
```

2.3.3 Operazioni

Le operazioni sono definite in OML come in generale in C++. Le operazioni su oggetti transienti e persistenti si comportano interamente come definito nel contesto operazionale definito nello standard C++. Questo include le sovrapposizioni, l'invio, la struttura di chiamata e l'invocazione delle funzioni, la struttura di chiamata e l'invocazione dei metodi, il passaggio di parametri e la risoluzione, la gestione degli errori, e le regole di compilazione.

2.3.4 Classe Persistent_Object

La classe Persistent_Object viene introdotta e definita come segue:

Definizione:

```
class Persistent_Object {
public:
                Persistent_Object();
                Persistent_Object(const Persistent_Object &);
                ~Persistent_Object();
  void          mark_modified(); // marca l'oggetto
                                 // come modificato
  void *        operator new(size_t size,
                             const char *typename=0);
  void *        operator new(size_t size,
                             Ref<Persistent_Object> cluster,
                             const char *typename=0);
  void *        operator new(size_t size, Database *database,
                             const char *typename=0);
  virtual void  odb_activate();
  virtual void  odb_deactivate();
};
```

Questa classe viene introdotta per permettere a chi definisce i tipi di specificare quando una classe, definita allo stesso modo di quelle transienti, è in grado di rimanere persistente. Una classe A che sia persistente dovrebbe ereditare dalla classe Persistent_Object:

```
class My_Class:public: Persistent_Object{...};
```

Alcune implementazioni, anche se accettano la superclasse Persistent_Object come indicazione di persistenza potenziale, non necessitano dell'introduzione fisica di una classe Persistent_Object. Tutte le implementazioni accettano chiamate a mark_modified anche se la classe Persistent_Object non esiste.

Per permettere ad un'applicazione di inizializzare parti transienti di un oggetto persistente, la funzione obj_activate può essere ridefinita in ogni classe

persistente. Questa funzione è chiamata automaticamente quando un oggetto viene messo in memoria. Al contrario, quando un oggetto viene rimosso dalla memoria, la funzione odb_deactivate viene chiamata automaticamente. Risulta responsabilità dell'applicazione ridefinire queste funzioni in una classe capace di persistenza quando la contabilità è stata fatta per la parte transiente di un oggetto persistente.

2.3.5 Classe Ref

Gli oggetti possono riferirsi ad altri oggetti attraverso un puntatore oppure tramite un riferimento chiamato Ref. Un Ref<T> è un riferimento ad una istanza di tipo T. C'è anche una classe Ref_Any definita per fornire un generico riferimento ad ogni tipo.

Un Ref è una classe template definita come segue:

Definizione:

```
template <class T> class Ref {
public:
               Ref();
               Ref(T *fromPtr);
               Ref(const Ref<T> &);
               Ref(const Ref_Any &);
               ~Ref();
               operator Ref_Any() const;
  Ref<T> &     operator=(T*);
  Ref<T> &     operator=(const Ref<T>&);
  Ref<T> &     operator=(const Ref_Any &);
  void         clear();
  T *          operator->() const; // dereferenzia il
                                   // riferimento
  T &          operator*() const;
  T *          ptr() const;
  int          is_null() const;
  void         delete_object(); // cancella l'oggetto
                                // referenziato dalla memoria
                                // e dalla base di dati
```

```
    // questi Ref e puntatori possono riferirsi
    // agli stessi oggetti?

    friend int   operator==(const Ref<T> &refL,
                            const Ref<T> &refR);
    friend int   operator==(const Ref<T> &refL,
                            const T *ptrR);
    friend int   operator==(const T *ptrL,
                            const Ref<T> &refR);
    friend int   operator==(const Ref<T> &refL,
                            const Ref_Any &anyR);
    friend int   operator==(const Ref_Any &anyL,
                            const Ref<T> &refR);
    friend int   operator!=(const Ref<T> &refL,
                            const Ref<T> &refR);
    friend int   operator!=(const Ref<T> &refL,
                            const T *ptrR);
    friend int   operator!=(const T *ptrL,
                            const Ref<T> &refR);
    friend int   operator!=(const Ref<T> &refL,
                            const Ref_Any &anyR);
    friend int   operator!=(const Ref_Any &anyL,
                            const Ref<T> &refR);
};
```

I riferimenti, in molti aspetti, si comportano come puntatori C++, ma forniscono un meccanismo aggiuntivo che garantisce l'integrità nei riferimenti ad oggetti persistenti. Anche se la sintassi per dichiarare un Ref è differente da quella per dichiarare un puntatore, l'utilizzo è, nella maggior parte dei casi, lo stesso che se fossero sovrapposti; e.g., i Ref ad una classe possono essere dereferenziati con l'operatore *, assegnati con l'operatore =, ecc. Un Ref ad una classe può essere assegnato ad un Ref di una superclasse. I Ref possono essere specializzati per fornire comportamenti referenziali specifici.

Il puntatore o il riferimento restituito dall'operatore -> o dall'operatore * è valido solamente finché Ref non viene cancellato, la transazione più esterna termina, oppure finché l'oggetto a cui punta viene cancellato. Il puntatore restituito da ptr resta valido finché la transazione più esterna ha termine o finché l'oggetto a cui punta viene cancellato. Il valore di un Ref dopo l'ese-

cuzione di una transazione (sia essa andata o meno a buona fine) è indefinito.

Una classe Ref_Any è definta per supportare un riferimento ad ogni tipo. Il suo obiettivo principale è di manipolare riferimenti generici e permettere la conversione di Ref in un tipo gerarchico. Un oggetto Ref_Any può essere usato come un intermediario tra qualunque due tipi Ref<X> e Ref<Y> dove X e Y sono tipi differenti. Un Ref<T> può sempre essere convertito in un Ref_Any; c'è una funzione che svolge la conversione nel template Ref<T>. Ogni classe Ref<T> possiede un costruttore ed un operatore di assegnamento che prende un riferimento ad un Ref_Any. L'oggetto referenziato dall'argomento Ref_Any deve essere di tipo T o una sottoclasse di T, altrimenti viene segnalato un errore.

La classe Ref_Any è definita come segue:

Definizione:

```
class Ref_Any {
public:
              Ref_Any();
              Ref_Any(const Ref_Any &);
              Ref_Any(Persistent_Object *);
              ~Ref_Any();
  Ref_Any &   operator=(const Ref_Any &);
  Ref_Any &   operator=(Persistent_Object *);
  void        clear();
  int         is_null();
  void        delete_object();

friend int    operator==(const Ref_Any &,
                         const Ref_Any &);
friend int    operator==(const Ref_Any &,
                         const Persistent_Object *);
friend int    operator==(const Persistent_Object *,
                         const Ref_Any &);
friend int    operator!=(const Ref_Any &,
                         const Ref_Any &);
friend int    operator!=(const Ref_Any &,
                         const Persistent_Object *);
friend int    operator!=(const Persistent_Object *,
```

```
                                const Ref_Any &);
};
```

Le operazioni definite su Ref<T> non sono dipendenti da uno specifico tipo T che sia stato fornito nella classe Ref_Any.

2.3.6 Classi Collezione

I template collezione sono forniti per supportare la rappresentazione di una collezione i cui elementi sono di un tipo arbitratio. Una implementazione deve supportare almeno i seguenti sottotipi di Collection:

- Set
- Bag
- List
- Varray

Le definizioni di classi del C++ per ognuno di questi tipi sono definiti nelle sottosezioni seguenti. Gli iteratori sono definiti un una sottosezione finale.

La seguente discussione utilizza la classe Set per spiegare le collezioni, ma la descrizione può essere applicata a tutte le classi derivate da Collection.

Datto un oggetto del tipo T, la dichiarazione

```
Set<T> s;
```

definisce una collezione Set i cui elementi sono del tipo T. Se questo set viene assegnato ad un'altro set dello stesso tipo, entrambi gli oggetti Set ed ogni elemento del set vengono copiati. Gli elementi sono copiati usando la semantica di copia definita per il tipo T. Una convenzione comune sarà quella di avere collezioni che contengono Ref ad oggetti persistenti. Ad esempio,

```
Set<Ref<Professor>> faculty;
```

La classe Ref ha una semantica di copia poco profonda. Per un Set<T>, se T è di tipo Ref<C> per una classe persistente C, solo gli oggetti Ref sono copiati, non gli oggetti C che gli oggetti Ref referenziano.

Questo è valido in ogni campo; in particolare, se s è dichiarato come attributo all'interno di una classe, l'intero set sarà inserito all'interno come un'istanza della classe Set. Quando un oggetto di questa classe inclusa è copiato in un'altro oggetto della stessa classe, il set interno viene volta copiato seguendo la semantica di copia definita sopra. Questo deve essere differenziato dalla dichiarazione

```
Ref<Set<T>> ref_set;
```

che definisce un riferimento ad un Set. Quando un riferimento è definito come una proprietà di una classe, questo significa che il set è un oggetto indipendente che giace fuori da un'istanza della classe. Parecchi oggetti possono avere in comune lo stesso set, anche se copiando un oggetto non si copierà anche il set, ma soltanto il riferimento ad esso. Questo verrà illustrato in Figura 2.11.

Gli elementi di una collezione possono essere di ogni tipo. Ogni tipo T che diventa un elemento di una data collezione deve supportare le seguenti operazioni:

```
class T {
public:
                T();
                T(const T &);
                ~T();
     T &        operator=(const T &);
friend int      operator==(const T &, const T &);
};
```

Questo è l'insieme completo delle funzioni richieste per definire le semantiche di copia per un dato tipo. Per i tipi che richiedono un ordine, deve anche essere fornita la seguente operazione

```
friend int      operator<(const T &, const T &);
```

Si noti che il compilatore C++ genererà automaticamente un costruttore di copia ed un operatore di assegnamento se il progettista della classe non ne dichiara uno. Si noti anche che la classe Ref<T> fornisce queste operazioni, eccetto che per operator<.

Le collezioni di letterali, inclusi letterali atomici e strutturati, sono definite come parte dello standard. Queste includono primitive ed i tipi Set<*literal_type*> definiti dall'utente sono definiti con lo stesso comportamento; e.g., Set<int>, Set<Struct time_t>.

Il diagramma mostrato in figura 2.11 illustra vari tipi coinvolgenti Set, Ref, tipi letterali L (int per esempio) e classi persistenti T. L'oggetto Set è rappresentato da un rettangolo che referenzia ad un insieme di elementi del tipo specificato. Una freccia continua è usata per denotare il contenimento dell'insieme di elementi all'interno dell'oggetto Set. I Ref hanno delle frecce tratteggiate che puntano agli oggetti referenziati del tipo T.

Classe Collection La classe Collection è una classe astratta in C++ e non può avere istanze. Deriva da Persistent_Object, permette istanze di classi concrete derivate da essa. Come per il caso affrontato nella sezione 2.3.4 sui Persistent_Object, non tutte le implementazioni richiedono che la classe Persistent_Object venga introdotta per supportare la persistenza; queste implementazioni non deriveranno necessariamente Collection da Persistent_Object.

Definizione:

```
template <class T> class Collection:
  public Persistent_Object {
public:
                    Collection();
                    Collection(const Collection<T> &);
                    ~Collection();
  Collection<T> &   operator=(const Collection<T> &);
friend int          operator==(const Collection<T> &cL,
                               const Collection<T> &cR);
friend int          operator!=(const Collection<T> &cL,
                               const Collection<T> &cR);
  unsigned long     cardinality();
```

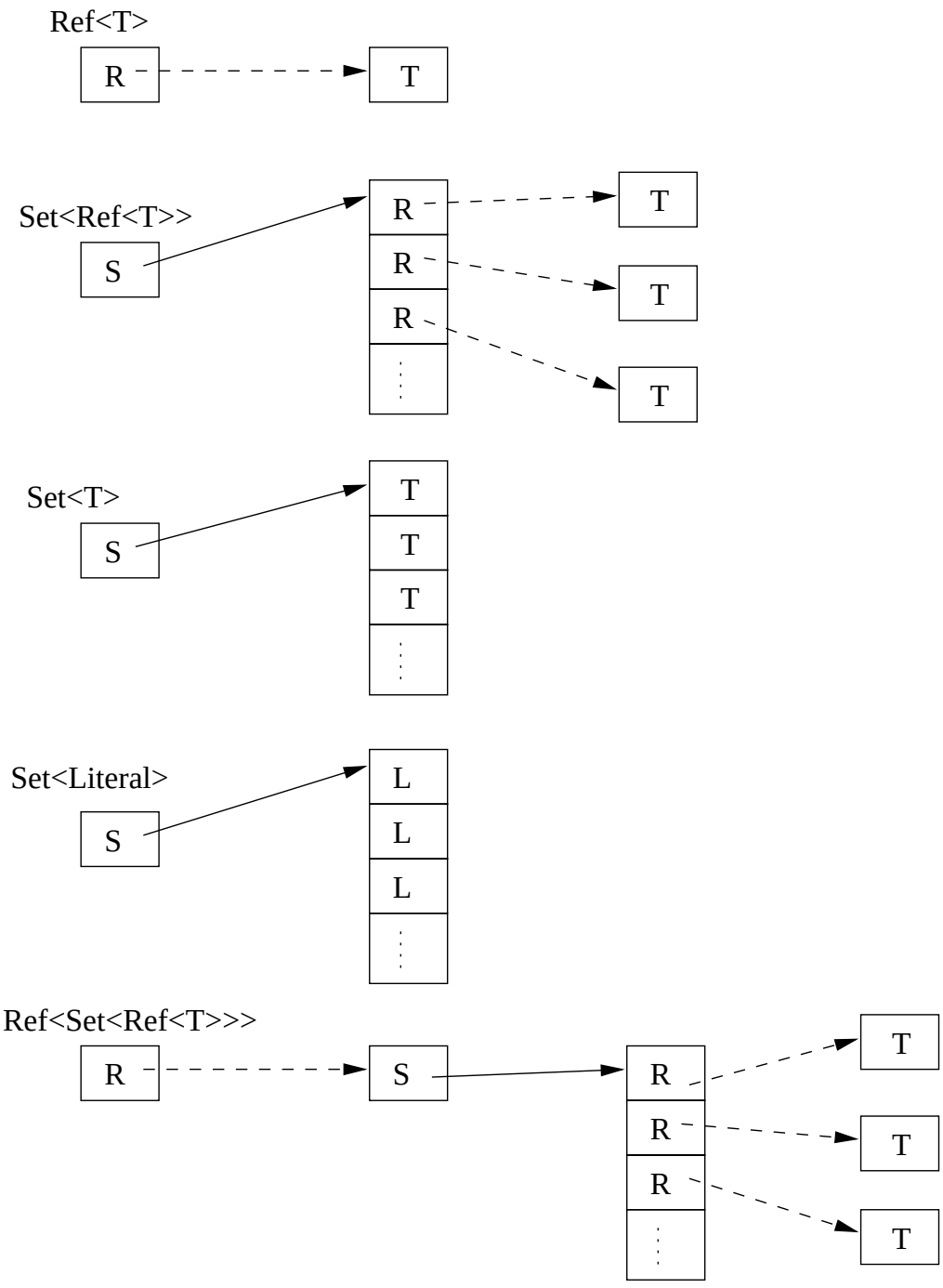

Figura 2.11: Collezioni e Riferimenti

```
  int              is_empty() const;
  int              is_ordered() const;
  int              allows_duplicates() const;
  int              contains_element(
                        const T &element) const;
  void             insert_element(const T &elem);
  void             remove_element(const T &elem);
  void             remove_all();
  const T &        retrieve_element_at(
                        const Iterator<T> &iter) const;
  void             remove_element_at(
                        const Iterator<T> &iter);
  void             replace_element_at(const T &,
                        const Iterator<T> &);
  Iterator<T>      create_iterator()const;
  const T &        select_element(
                        const char *OQL_predicate) const;
  Iterator<T>      select(const char *OQL_predicate) const;
  int              query(Collection<T> &,
                          const char *OQL_predicate) const;
  int              exist_element(
                           const char *OQL_predicate) const;
};
```

Classe Set Un Set<T> è una collezione non ordinata di elementi di tipo T senza duplicati.

Definizione:

```
template <class T> class Set:public Collection<T> {
public:
              Set();
              Set(const Set<T> &);
              ~Set();
  Set<T> &    operator=(const Set<T> &);
  Set<T> &    union_of(const Set<T> &sL,const Set<T> &sR);
  Set<T> &    union_with(const Set<T> &s2);
  Set<T> &    operator+=(const Set<T> &s2); // union_with
  Set<T>      create_union(const Set<T> &s) const;
```

```
  friend Set<T> operator+(const Set<T> &s1,
                          const Set<T> &s2);
    Set<T> &    intersection_of(const Set<T> &sL,
                                const Set<T> &sR);
    Set<T> &    intersection_with(const Set<T> &s2);
    Set<T> &    operator*=(const Set<T> &s2);
                          // intersection_with
    Set<T>      create_intersection(const Set<T> &s) const;
  friend Set<T> operator*(const Set<T> &s1,
                          const Set<T> &s2);
    Set<T> &    difference_of(const Set<T> &sL,
                              const Set<T> &sR);
    Set<T> &    difference_with(const Set<T> &s2);
    Set<T> &    operator-=(const Set<T> &s2);
                          // difference_with
    Set<T>      create_difference(const Set<T> &s) const;
  friend Set<T> operator-(const Set<T> &s1,
                          const Set<T> &s2);
    int         is_subset_of(const Set<T> &s2) const;
    int         is_proper_subset_of(
                           const Set<T> &s2) const;
    int         is_superset_of(const Set<T> &s2) const;
    int         is_proper_superset_of(
                           const Set<T> &s2) const;
  };
```

Si noti che tutte le operazioni definite nel tipo Collection sono ereditate dal tipo Set, e.g., insert_element, remove_element, select_element, e select.

Esempi:

- creazione:

```
Database db               // apriamo la base di dati
Ref<Professor> Guttag     // settiamo questo
                          // ad un professore
Ref<Set<Ref<Professor>>> my_profs=new8&db)
                              Set<Ref<Professor>>;
```

- inserimento:

```
my_profs->insert_element(Guttag);
```

- rimozione:

```
my_profs->remove_element(Guttag);
```

- cancellazione:

```
my_profs->delete_object();
```

Per ognuna delle operazioni sugli insiemi (unione, intersezione e differenza) ci sono tre modi di ottenere l'insieme risultante. Queste saranno spiegate utilizzando l'operazione di unione. Ognuna delle funzioni unione ha due operandi insieme e compie la loro unione. Esse variano nel modo in cui gli operandi insieme sono passati e rispetto al modo in cui il risultato viene restituito, per supportare i differenti stili di interfaccia. La funzione union_of è una funzione membro che ha due argomenti che sono riferimenti a Set<T>. Essa esegue l'unione di questi due insiemi e pone il risultato nell'oggetto Set con cui la funzione è stata chiamata, rimuovendo il contenuto originario dell'insieme. La funzione union_with è anch'essa un membro e pone il suo risultato nell'oggetto con cui è stata chiamata. La differenza è che union_with utilizza il contenuto dell'insieme corrente come uno dei due operandi che devono essere uniti, e quindi è richiesto il passaggio di un solo parametro. Sia union_of che union_with restituiscono un riferimento all'oggetto con cui sono state invocate. La funzione union_with ha una corrispondente funzione operator+= definita. Dall'altra parte, create_union crea e restituisce una nuova istanza Set per valore che contiene l'unione, lasciando inalterati i due insiemi originali. Anche questa funzione possiede la corrispondente funzione operator+ definita.

Classe Bag Una Bag<T> è una collezione disordinata di elementi di tipo T che permette valori duplicati.

Definizione:

```
template <class T> class Bag:public Collection<T> {
public:
            Bag();
            Bag(const Bag<T> &);
```

```
              ~Bag();
  Bag<T> &    operator=(const Bag<T> &);
  Bag<T> &    union_of(const Bag<T> &bL,
                       const Bag<T> &bR);
  Bag<T> &    union_with(const Bag<T> &b2);
  Bag<T> &    operator+=(const Bag<T> &b2); // union_with
  Bag<T>      create_union(const Bag<T> &b) const;
friend Bag<T> operator+(const Bag<T> &b1,
                        const Bag<T> &b2);
  Bag<T> &    intersection_of(const Bag<T> &bL,
                              const Bag<T> &bR);
  Bag<T> &    intersection_with(const Bag<T> &b2);
  Bag<T> &    operator*=(const Bag<T> &b2);
                         // intersection_with
  Bag<T>      create_intersection(const Bag<T> &b) const;
friend Bag<T> operator*(const Bag<T> &b1,
                        const Bag<T> &b2);
  Bag<T> &    difference_of(const Bag<T> &bL,
                            const Bag<T> &bR);
  Bag<T> &    difference_with(const Bag<T> &b2);
  Bag<T> &    operator-=(const Bag<T> &b2);
                         // difference_with
  Bag<T>      create_difference(const Bag<T> &b) const;
friend Bag<T> operator-(const Bag<T> &b1,
                        const Bag<T> &b2);
};
```

Le operazioni di unione, intersezione e differenza sono descritte nella sezione precedente relativa alla classe Set.

Classe List Una List<T> è una collezione ordinata di elementi di tipo T e permette valori duplicati. Il valore iniziale dell'indice di una lista è 0, seguendo la convenzione di C e C++.

Definizione:

```
template <class T> class List:public Collection<T> {
public:
```

```
                List();
                List(const List<T> &);
                ~List();
  List<T> &     operator=(const List<T> &);
  const T &     retrieve_first_element() const;
  const T &     retrieve_last_element() const;
  void          remove_first_element();
  void          remove_last_element();
  const T &     operator[](unsigned long position) const;
  int           find_element(const T &element,
                             unsigned long &position) const;
  const T &     retrieve_element_at(
                             unsigned long position) const;
  void          remove_element_at(unsigned long position);
  void          replace_element_at(const T &element,
                                   unsigned long position);
  void          insert_element_first(const T &element);
  void          insert_element_last(const T &element);
  void          insert_element_after(const T &element,
                                   unsigned long position);
  void          insert_element_before(const T &element,
                                    unsigned long position);
  List<T>       concat(const List<T> &listR) const;
friend List<T>  operator+(const List<T> &listL,
                          const List<T> &listR);
  List<T> &     append(const List<T> &listR);
  List<T> &     operator+=(const List<T> &listR);
};
```

La funzione insert_element (ereditata da Collection<T>) inserisce un nuovo elemento alla fine della lista.

Classe Array Il tipo Array definito nello standard è implementato dall'array costruito internamente al linguaggio C++. Questo è un array ad una dimensione e di lunghezza fissata.

Classe Varray Un Varray<T> è un array ad una dimensione di lunghezza variabile contenente elementi del tipo T. Il valore iniziale dell'indice del

Varray è 0, seguendo la convenzione di C e C++.

Definizione:

```
template <class T> class Varray:public Collection<T> {
public:
                Varray();
                Varray(unsigned long length);
                Varray(const Varray<T> &);
                ~Varray();
  Varray<T> &   operator=(const Varray<T> &);
  unsigned long upper_bound() const;
  void          resize(unsigned long length);
  const T &     operator[](unsigned long index) const;
  int           find_element(const T &element,
                             unsigned long &index) const;
  const T &     retrieve_element_at(
                             unsigned long index) const;
  const T &     remove_element_at(unsigned long index);
  const T &     replace_element_at(const T &element,
                                   unsigned long index);
};
```

La funzione insert_element (ereditata da Collection<T>) inserisce un nuovo elemento incrementando la lunghezza del Varray di uno e ponendo il nuovo elemento in questa nuova posizione del Varray.

Esempi:

```
Varray<double>          vector(100);
vector.replace_element_at(3.14159,97);
vector.resize(2000);
```

Classe Iterator Una classe template Iterator<T> definisce il generico comportamento per l'iterazione. Tutti gli iteratori usano un protocollo consistente per restituire sequenzialmente ogni elemento dalla collezione sulla quale l'iterazione è definita. Una classe template è stata usata per darci degli

iteratori di tipo sicuro, i.e., iteratori che garantiscono la restituzione di un'istanza del tipo dell'elemento della collezione sulla quale l'iteratore è definto. Normalmente, un iteratore è inizializzato tramite il metodo create_iterator su una classe collezione. La classe template Iterator<T> è definita come segue:

```
template <class T> class Iterator {
public:
                  Iterator(const Collection<T> &);
                  Iterator(const Iterator<T> &);
                  ~Iterator();
  Iterator<T> &   operator=(const Iterator<T> &);
  void            reset();
  int             not_done() const;
  void            advance();
  void            operator++();
  void            operator++(int);
  const T &       get_element() const ;
  int             next(T &objRef);
};
```

Quando un iteratore viene costruito, esso prende un riferimento ad una collezione (un'istanza di una classe concreta derivata da Collection) oppure un'altro iteratore. L'assegnamento dell'iteratore è comunque supportato. Una funzione di reset è fornita per re-inizializzare l'iteratore all'inizio dell'iterazione per la stessa collezione.

La funzione not_done permette di determinare se vi sono ancora elementi della collezione da visitare nella iterazione. Essa restituisce 1 se ci sono ancora elementi e 0 se l'iterazione è completa. La funzione advance muove l'iteratore avanti fino al prossimo elemento della collezione. Se l'iterazione non è ancora cominciata, essa setta l'iteratore al primo elemento dell'insieme. Le forme prefissa e postfissa dell'operatore ++ di incremento sono sovrapposte per fornire una operazione equivalente di avanzamento.

La funzione get_element restituisce un riferimento costante all'elemento corrente. All'inizio dell'iterazione, essa restituisce il primo elemento; altrimenti, durante l'iterazione, essa restituisce l'elemento corrente. Se non c'è l'elemento corrente viene sollevato un errore. Non c'è l'elemento corrente se l'iterazione è stata completata (not_done restituisce 0) oppure se la collezione non pos-

siede elementi.

La funzione next fornisce il mezzo per controllare la fine dell'iterazione, facendo avanzare l'iteratore e restituendo l'elemento corrente, se ce n'è uno. Si comporta nel modo seguente:

```
template <class T> int iterator<T>::next(T &objRef)
{
 if(!not_done()) return 0; // non ci sono altri elementi,
                           // restituisce falso
 advance();                // se all'inizio, si posiziona
                           // al primo elemento,
                           // altrimenti al prossimo
 objRef=get_element();     // assegna al parametro di
                           // uscita l'elemento
 return 1;                 // restituisce vero,
                           // si ha un elemento successivo
}
```

Si noti che la funzione next esegue l'assegnamento di un oggetto, mentre usando della funzione get_element si otterrebbe un semplice riferimento costante all'elemento. L'ultima potrebbe essere preferibile in situazioni in cui il tipo di elemento ha una semantica di copia profonda e l'applicazione non ha bisogno o non vuole pagare il tempo di esecuzione della copia dell'oggetto.

Queste operazioni permettono due tipi di stili iterativi, utilizzabili per cicli while o for.

Esempi:
Data una classe Student, ed una sua istanza esistente students:

```
(1) Iterator<Ref<Student>> iter=strudents.create_iterator();
    Ref<Student> s;
(2) while(iter.next(s)) {
        ...
    }
```

o in modo equivalente con un ciclo for:

```
(3) for(Iterator<Ref<Student>> iter=strudents;
                        iter.not_done(); iter++) {
(4)    Ref<Student> s=iter.get_element();
       ...
    }
```

L'istruzione (1) definisce un iteratore iter che si muove sulla collezione students. L'istruzione (2) itera attraverso questa collezione, restituendo un Ref ad uno Student ad ogni successiva chiamata di next, legando ciò al ciclo della variabile s. Il corpo dell'istruzione while è allora eseguita una volta per ogni studente nella collezione students. Le ciclo (3) l'iteratore viene inizializzato, l'iterazione viene controllata per vedere se risulta completa e l'iteratore viene avanzato. All'interno del ciclo for, la funzione get_element può essere chiamata per ottenere un riferimento all'elemento corrente.

2.3.7 Transazioni

La semantica delle transazioni è definita nel modello ad oggetti esposto nello standard.

Le transazioni possono essere fatte iniziare, si possono completare, abortire e controllare. Le transazioni possono anche essere innestate. Risulta importante osservare che *accesso, creazione, modifica e cancellazione di oggetti persistenti devono essere fatti mediante una transazione.*

Le transazioni sono implementate in C++ come oggetti della classe Transaction. La classe Transaction definisce le operazioni per iniziare, completare, abortire e controllare le transazioni. Queste operazioni sono:

```
class Transaction {
public:
                Transaction();
                ~Transaction();
  void          begin();
  void          commit();
  void          abort();
  void          checkpoint();
```

```
};
```

Le transazioni devono essere esplicitamente create ed iniziate; esse non vengono iniziate automaticamente con la base di dati aperta, dopo la creazione di un oggetto Transaction, nemmeno se seguita da un'operazione di commit o abort.

La funzione begin inizia una transazione. Chiamando begin molte volte sullo stesso oggetto transazione, senza l'intervento di commit o abort, si genera un errore sulla seconda e sulle susseguenti chiamate.

Chiamando commit si coinvolgono tutti gli oggetti persistenti modificati appartenenti alla base di dati (compresi quelli creati o cancellati) all'interno della transazione. L'operazione di commit setta i Ref, sia per gli oggetti persistenti agli oggetti transienti a 0 ed i puntatori interni ad oggetti persistenti, sia che puntino ad oggetti persistenti, sia che puntino ad oggetti transienti, vengono a loro volta settati a 0. In alternativa, le implementazioni possono scegliere di mantenere la validità dei puntatori tra oggetti persistenti attraverso i confini della transazione. L'operazione di commit non cancella gli oggetti della transazione.

Chiamando checkpoint si salvano su memeoria persistente gli oggetti modificati all'interno della transazione da quando è stato fatto l'ultimo controllo sulla base di dati. Tutti i Ref ed i puntatori restanoinvariati.

Chiamando abort, si abortiscono i cambiamenti agli oggetti, inoltre non viene cancellato l'oggetto transazione.

Il distruttore abortisce la transazione se risulta attiva.

Nello standard corrente, gli oggetti transienti non sono soggetti alla semantica delle transazioni. L'impegno di una transazione non rimuove gli oggetti transienti dalla memoria. L'aborto di una transazione non ripristina lo stato degli oggetti transienti modificati.

Gli oggetti transazione non hanno una vita lunga (oltre i confini del processo) e non possono essere salvati nella base di dati. Questo significa che gli oggetti transazione possono non essere persistenti e che la nozione di "transazione lunga" non è definita in questa specifica.

Le transazioni possono essere innestate. Il commit della transazione più interna è soltanto relativo alla transazione più esterna; i cambiamenti fatti in quella interna vengono coinvolti soltanto se anche le transazioni esterne raggiungono il commit. Se una transazione interna ha raggiunto il commit su un aborto di una transazione esterna, i cambiamenti fatti nella transazione interna sono abortiti a loro volta.

Riassumendo, le regole che si applicano alla modifica di un oggetto (necessariamente durante una transazione) sono:

1. I cambiamenti fatti sugli oggetti persistenti all'interno di una transazione possono essere "disfatti" abortendo la transazione.
2. Gli oggetti transienti sono oggetti C++ standard.
3. Gli oggetti persistenti creati nell'ambito di una transazione sono manipolati coerentemente ai confini della transazione: salvati nella base di dati e rimossi dalla memoria (al commit della transazione) oppure cancellati (come risultato dell'aborto di una transazione).

2.3.8 Operazioni sulla Base di Dati

C'è un tipo Database predefinito. Esso supporta i seguenti metodi:

```
class Database {
public:
  enum access_status {not_open, read_write,
                      read_only, exclusive};

  void            open(const char *database_name,
                       access_status status=read_write);
  void            close();
  void            set_object_name(const Ref_Any &theObject,
                                  const char *theName);
  const char *    get_object_name(const Ref_Any &) const;
  void            rename_object(const char *oldName,
                                const char *newName);
  Ref_Any         lookup_object(const char *name) const;
};
```

L'oggetto base di dati, come un oggetto transazione, è transiente. Le basi di dati non possono essere create da programma usando il C++ OML definito

da questo standard. Le basi di dati devono essere aperte prima di cominciare qualunque transazione che usi la base di dati stessa, e chiuse dopo la fine di queste transazioni.

Per aprire un base di dati, si usa Database::open, che prende il nome della base di dati come suo argomento. Questo inizializza l'istanza dell'oggetto Database.

```
database->open("myDB");
```

Il metodo open localizza la base di dati nominata ed esegue le appropriate connessioni alla base di dati. Occorre aprire la base di dati prima di poter accedere ai suoi oggetti. Eventuali estensioni al metodo open permetteranno ad alcuni ODBMS di implementare nomi di basi di dati di default e/o aprire una base di dati di default quando una sessione di una base di dati è cominciata. Alcuni ODBMS possono supportare l'apertura logica allo stesso modo di quella fisica. Alcuni ODBMS possono supportare la loro connessione a basi di dati multiple allo stesso istante.

Per chiudere una base di dati si usa Database::close:

```
database->close();
```

Dopo che si è chiusa una base di dati, ogni tentativo di accedere agli oggetti che contiene genererà un errore. Il comportamento al termine del programma, se le basi di dati non sono chiuse o le transazioni non hanno raggiunto il commit o sono state abortite, risulta indefinito.

I metodi *name* permettono la manipolazione dei nomi degli oggetti. Un oggetto può acquisire soltanto un nome. Il metodo set_object_name assegna un nome stringa di caratteri all'oggetto referenziato. Se la stringa fornita come argomento nome non risulta unica all'interno della base di dati, viene generato un errore. Se l'oggetto possiede già un nome, viene ribattezzato col nuovo nome. Il metodo rename_object cambia il nome di un oggetto. Se il nuovo nome è già utilizzato, viene sollevato un errore e viene lasciato il vecchio nome. Un oggetto nominato può avere il suo nome rimosso ponendo a 0 il puntatore al valore del nuovo nome. Quando un oggetto nominato viene cancellato, la sua voce viene automaticamente rimossa dall'elenco dei nomi.

Ad un oggetto si accede tramite nome utilizzando la funzione membro Database::lookup_object.

Esempio:

```
Ref<Professor> prof = myDatabase->lookup_object("Newton");
```

Se un'istanza chiamata "Newton" esiste, essa viene referenziata ed un Ref_Any viene restituito da lookup_object. Il valore restituito Ref_Any viene usato per inizializzare prof. Se invece l'oggetto chiamato "Newton" non è un'istanza di Professor o una sottoclasse di Professor, viene generato un errore durante questa inizializzazione.

2.4 C++ OQL

Nello standard viene descritto a grandi linee il Linguaggio di Interrogazione ad Oggetti. In questa sezione la semantica OQL viene mappata nel linguaggio C++. Vi sono in generale due opzioni per legare un sottoliguaggio di interrogazione ad un linguaggio di programmazione: accoppiamento lasco o accoppiamento stretto. Nell'approccio ad accoppiamento lasco, funzioni interrogazione sono introdotte per prendere stringhe contenenti interrogazioni come loro argomenti. Queste funzioni analizzano e valutano l'interrogazione durante l'esecuzione del programma, restituendo il risultato attraverso il parametro di output result. Nell'approccio ad accoppiamento stretto il sottolinguaggio di interrogazione viene integrato direttamente nel linguaggio di programmazione espandendo la definizione dei valori non terminali <term> ed <expression> come definiti nella BNF del linguaggio di programmazione. L'approccio ad accoppiamento stretto permette interrogazioni che possono essere ottimizzate durante la compilazione; nell'approccio ad accoppiamento lasco esse vengono in generale ottimizzate durante l'esecuzione. Il legame C++ per OQL supporta due varianti per l'accoppiamento lasco:

- un metodo di interrogazione è definito nella classe generica Collection
- una funzione oql libera al di fuori di ogni classe.

Le due varianti sono definite nelle sottosezioni seguenti.

2.4.1 Metodi di Interrogazione nella Classe Collection

La classe Collection possiede un metodo interrogazione la cui dichiarazione è:

```
int query(Collection<T> &result,
          const char* predicate) const;
```

Questa funzione filtra la collezione usando il predicato ed assegnando il risultato al primo parametro. Restituisce un codice diverso da zero, se l'interrogazione non è stata ben formulata. Il predicato è dato come una stringa con la sintassi della clausola *where* di OQL. La variabile predefinita this viene usata all'interno del predicato per denotare l'elemento corrente della collezione che viene filtrata.

Esempio:
Data la classe Student, dove Students è una collezione di oggetti della classe Student, si ricavi l'insieme degli strudenti che seguono i corsi di matematica:

```
Set<Ref<Student>> mathematicians;  // estensione di matematici
Students->query(mathematicians,
                "exist s in this.takes: \
                 s.section_of.name=\"math\"");
```

La funzione Collection::select_element prende un predicato OQL come argomento e restituisce l'elemento che soddisfa l'interrogazione. Se non c'è un solo elemento da restituire viene generato un errore. La funzione Collection::select passa un predicato OQL e restituisce un iteratore per attraversare la collezione di elementi che soddisfano l'interrogazione. La funzione Collection::exist_element viene passata come un predicato OQL e restituisce un valore non nullo se c'è almeno un elemento nella collezione che soddisfa l'interrogazione. Se l'argomento predicato di un qualunque di queste funzioni membro è mal posto, viene generato un errore.

2.4.2 Funzione OQL

La funzione oql permette al programmatore di ottenere l'accesso a tutte le funzionalità OQL da un programma C++. Essa è una funzione isolata, cioè non risulta parte di nessuna definizione di classe. Prende come parametri un riferimento ad una variabile in cui porre il risultato, una proposizione OQL ed una lista variabile di espressioni C++ i cui valori sono operandi di ingresso per l'interrogazione. All'interno dell'interrogazione questi operandi vengono identificati dalla sintassi data sotto. La funzione restituisce una valore non nullo se l'interrogazione non risulta ben posta.

Le dichiarazioni sovrapposte di questa funzione sono:

- Un'interrogazione che restituisce un oggetto.

```
template<class T> int oql(Ref<T> &result,
                          const char *query,...);
```

- Un'interrogazione che restituisce una collezione.

```
template<class T> int oql(Collection<T> &result,
                          const char *query,...);
```

- Un'interrogazione che restituisce un valore atomico.

```
int oql(int &, const char *query,...);
int oql(char &, const char *query,...);
int oql(double &, const char *query,...);
```

- Un'interrogazione che restituisce una stringa. In questo caso, il risultato è allocato dinamicamente tramite OQL. Il programma è responsabile della deallocazione della stringa.

```
int oql(char *&, const char *query,...);
```

Di seguito ai primi due parametri, oql accetta qualunque numero di parametri opzionali. Ognuno di essi può essere una espressione C++. La proposizione di interrogazione si riferisce all'i-esimo parametro di questa lista tramite la notazione di legame con questa sintassi:

```
$<range><type>
```

<range> è un intero che fornisce la posizione del parametro (partendo da 1). <type> è un carattere che definisce il tipo della espressione C++:

- i indica un intero (int)
- c indica un carattere (char)
- r indica un reale (double)
- s indica una stringa (char *)
- k indica una collezione (oggetto di una sottoclasse di Collection<T>)
- o indica un oggetto (oggetto referenziato da una classe o sottoclasse Ref<T>)

- i indica un intero (int)

Il controllo del tipo dei parametri di input in accordo col loro uso nell'interrogazione viene fatto durante l'esecuzione. Similmente il tipo del risultato dell'interrogazione viene controllato per accordarsi col tipo del primo parametro. Ogni violazione di tipo genera un errore.

Se l'interrogazione restituisce un oggetto mutabile di tipo T, la funzione restituisce un Ref<T>. Se l'interrogazione restituisce un letterale strutturato, il valore di questo è assegnato al valore dell'oggetto o collezione denotato dalla variabile risultato, assicurando un accordo dei tipi.

Esempio:
Tra gli strudenti di matematica (ottenuti prima nella sezione 2.4.2 nella variabile mathematicians) c'è chi è un assistente all'insegnamento e che guadagna più di x, trovare l'insieme dei professori da essi assistiti. Si suppone che esista una estensione per gli assistenti all'insegnamento che viene chiamata "TA".

```
Set<Ref<Student>> mathematicians;  // ottenuto come
                                   // in precedenza
Set<Ref<Professor>> assisted_profs;
double x=...

oql(assisted_prof, "select t.assist.taught_by \
                        from t in TA where t.salary >$1r \
                        and t in $2k",
                        x, mathematicians);
```

2.5 Esempio

Questa sezione fornisce un esempio completo di una piccola applicazione C++. Questa applicazione gestisce record di persone. Una istanza di Person può essere introddotta nella base di dati. Quindi alcuni eventi speciali possono essere registrati: il suo matrimono, la nascita dei figli, il suo trasferimento ad un nuovo indirizzo.

L'applicazione comprende due transazioni: la prima popola la base di dati, mentre la seconda la consulta e la aggiorna.

La prossima sezione definisce lo schema della base di dati come classi C++

ODL. Il programma C++ viene dato nella sezione susseguente.

2.5.1 Definizione dello Schema

Per la spiegazione della semantica di questo esempio, si veda lo standard. Di seguito vi è la sintassi C++ ODL:

```
// Definizione dello schema in C++ ODL
class City;              // dichiarazione successiva

struct Address {
    int        number;
    String     street;
    Ref<City>  city;
               Address();
               Address(int, const char*, const Ref<City> &);
};

class Person: public Persisten_Object {
public:
// Attributi (tutti pubblici, in questo esempio)
   String                name;
   Address               address;

// Relazioni
   Ref<Person>           spouse   inverse spouse;
   List<Ref<Person>>     children inverse parents;
   List<Ref<Person>>     parents  inverse children;

// Operazioni
                         Person(const char *pname);
   void                  birth(const Ref<Person> &child);
                                 // un figlio nasce
   void                  marriage(const Ref<Person> &with);
                                 // uno sposo/a
   Ref<Set<Ref>person>>> ancestors() const;
                                 // restituisce gli
                                 // antenati di Person
   void                  move(const Address &);
```

```
                                    // trasferisce
                                    // l'indirizzo

  // Estensioni
  static Ref<Set<Ref<Person>>> people; // riferimento alla
                                       // classe estensione:
                                       // questa variabile
                                       // statica (transiente)
                                       // viene inizializzata
                                       // quando inizia
                                       // la transazione (si
                                       //veda l'applicazione)
  static const char * const    extent_name;

  };

  class City:public Persistent_Object {
  public:
  // Attributi
     int                       city_code;
     String                    name;
     Ref<Set<Ref<Person>>>     population;
                                 // gente che vive in qui
  // Operazioni
                               City(int, const char *);
  // Estensioni
  static Ref<Set<Ref<City>>>  cities; // riferimento alla
                                      // classe estensione
  static const char * const   extent_name;
  };
```

2.5.2 Implementazione dello Schema

Definiamo adesso il codice delle operazioni dichiarate nello schema. Questo viene scritto in C++. Si assume che un preprocessore C++ ODL abbia generato un file chiamato "schema.hxx" e che contenga le definizioni standard C++ equivalenti alle classi C++ ODL.

```
// Implementazione delle Classi in C++
#include "schema.hxx"
```

```
//Struttura Address

Address::Address(int pnumber, const char * pstreet,
                  const Ref<City> &pcity)
   : number(pnumber);
     street(pstreet);
     city(pcity);
{}

Address::Address()
   : number(0);
     street(0);
     city(0);
{}

// Classe Person

const char * const Person::extent_name="people";

Person::Person(const char * pname)
   : name(pname)
{
 people->insert_element(this); // mette questa
                               // Person nella
                               // estensione
}

void Person::birth(const Ref<Persono> &child)
{                         // aggiunge un nuovo
                          // figlio alla lista
                          // dei figli
 children.insert_element_last(child);
 if(spouse!=0)
  spouse->children.insert_element_last(child);
}

void Person::marriage(const Ref<Persono> &with)
{                         // Inizializza la
                          // relazione spouse
 spouse=with;             // with->spouse viene
```

```
                          // automaticamente
                          // settato a
                          // questa Person
}

Ref<Set<Ref<Person>>> Person::ancestors()
{          // construisce l'insieme
           // degli antenati di questa Person
 Ref<Set<Ref<Person>>> the_ancestors=
                        new Set<Ref<Person>>;
 int i;
 for(i=0;i<2;i++)
  if (parents[i]!=0) {
   // antenati = genitori uniti
   // agli antenati dei genitori
   the_ancerstos->insert_element(parents[i]);
   Ref<Set<Ref<Persono>>> grand_parents=
                       parents[i]->ancestors();
   the_ancestors->union_with(*grand_parents);
   grand_parents.delete_object();
  }
 return the_ancestors;
}

void Person::move(const Address &new_address)
{   // aggiorna l'attributo indirizzo di Persona
 If(address_city!=0)
  address_city->population->remove_element(this);
 new_address.city->population->insert_element(this);
 address=new_address;
 mark_modified();      // DA NON DIMENTICARE!
                       // Si noti che risulta
                       // necessario solo nel
                       // caso in cui venga
                       // modificato l'attributo
                       // di un'oggetto. Quando
                       // si modifica una relazione,
                       // l'oggetto viene marcato
                       // modificato automaticamente
}
```

```
\\Classe City

const char * const City::extent_name="cities";

City::City(int code, const char *cname)
   : city_code(code),
     name(cname);
{
 cities->insert_element(this);
}
```

2.5.3 Una Applicazione

Adesso abbiamo l'intero schema ben definito ed implementato. Siamo in grado di popolare la base di dati e manipolarla. Nella seguente applicazione, la transazione Load costruisce alcuni oggetti e li inserisce nella base di dati. La transazione Consult li legge, stampa alcuni rapporti su di essi, e li marca aggiornati. Ogni transazione è implementata all'interno di una funzione C++.
La base di dati è aperta dal programma principale, che poi inizia le transazioni.

```
#include<iostream.h>
#include "schema.hxx"

static Database dbobj;
static Database *database = &dbobj;

void Load()
{             // Transazione che popola la base di dati
 Transaction load;
 load.begin();

 // Creazione le estensioni Persone e Citta' e li nomina.

 Person::people=new(database) Set<Ref<Person>>;
 City::cities=new(database) Set<Ref<City>>;

 database-set_object_name(Person::people,
                          Person::extent_name);
```

```
 database-set_object_name(City::cities,
                          City::extent_name);

 // Costruzione  di 3 oggetti
 // persistenti della classe Person.

 Ref<Person> God, Adam, Eve;

 God=new(database) Person("God");
 Adam=new(database) Person("Adam");
 Eve=new(database) Person("Eve");

 // Costruzione di una struttura Address, Paradise,
 // come (7 Apple Street Garden)
 // e settaggio degli attributi address di Adam e Eve.

 Address Peradise(7,"Apple",new(database) City(0,"Garden");

 Adam->move(Paradise);
 Eve->move(Paradise);

 // Definizione delle relazioni famigliari.

 God->birth(Adam);
 Adam->marriage(Eve);
 Adam->birth(new(database) Person("Cain"));
 Adam->birth(new(database) Person("Abel"));

 load.commit();  // impegna la transizoine, prendendo gli
                 // oggetti dalla dase di dati.
}

static void print_persons(const Collection<Ref<Person>> & s)
{       // Funzione di servizio per
        // stampare una collezione di Person
 Ref<Person> P;
 Iterator<Ref<Person>> it=s.create_iterator();
 while(it.next(p)) {
  cout << ""---" << p->name << "lives in";
  if(p->address.city!=0)
   cout << p->address.city->name;
```

```
   else
    cout << "Unknown";
   cout << endl;
  }
 }

 void Consult()
 {   // Transazione che consulta e aggiorna la base di dati
  Transaction consult;
  List<Ref<Person>> list;
  consult.begin();

  // I riferimenti agli oggetti o alle collezioni devono
  // essere ricalcolati dopo un impegno

  Person::people=database->
                  lookup_object(Person::extent_name);
  City::cities=database->
                  lookup_object(City::extent_name);

  // Adesso inizia la transazione;

  cout << "All the people ...:" << endl;
  print_persons(*Person::people);

  cout << "All the people sorted by name ...:" << endl;
  oql(list,"sort p in people by p.name");
  print_persons(list);

  cout << "People having 2 children\
           and living in Paradise ...:" << endl;
  oql(list, "select p from p in people\
             where  p.address.city !=nil and \
             p.address.city.name=\"Garden\"\
             and count(p.children)=2");
  print_persons(list);

  // Adam e Eve traslocano

  Address Earth(13,"Macadam",
                 new(database) City(1, "St-Croix"));
```

```
 Ref<Person> Adam;
 oql(Adam, "element(select p from p in people \
            where p.name = \"Adam\")");
 Adam.move(Earth);
 Adam->spouse->move(Earth);

 cout << "Cain's ancestors ...:" << endl;
 Ref<Person> Cain=Adam->children.retrieve_element_at(0);
 print_persons(*(Cain->ancestors()));
 consult.commit();
}

main()
{
 database->open("family");
 Load();
 Consult();
 database->close();
}
```

2.6 Future Estensioni del Legame C++ ODL/OML

Questa sezione definisce il legame C++ ODL/OML che il gruppo di standardizzazione ODMG-93 offrirà in futuro. Ci si aspetta che anche la tecnologia C++ evolva per supportare il grado di trasparenza descritta in questa sezione o che un pre-processore C++ possa essere introdotto per permettere questa trasparenza.

Semplicemente, il legame futuro è una più completa realizzazione del principio che non dovrebbe esservi differenza nel modo in cui un programmatore C++ tratta gli oggetti persistenti e transienti. Allo stesso tempo, questo richiede una più complessa implementazione. In assenza dei cambiamenti discussi nella sezione 2.6.2, il legame futuro richiede un pre-processore capace di maneggiare l'intero linguaggio C++, e allo stesso modo vi sia una integrazione attenta con i maggiori debugger ed ambienti di programmazione per il C++. Riconoscendo le difficoltà pratiche di conseguire delle implementazioni robuste di questo legame in prodotti correntemente in uso nel 1993, i membri del gruppo di ODMG hanno scelto di porre l'obiettivo del legame,

spiegato dalla sezione 2.2 alla sezione 2.5, come uno standard basato sulle linee per la portabilità di applicazioni di intersistemi oggi, piuttosto che il legame spiegato in questa sezione 2.6. Il legame futuro viene definito in questo standard comunque, per prevenire un problema che il gruppo ODMG-93 - assieme a venditori con clienti di base coinvolti con otto differenti legami, alcuni abbastanza simili, altri veramente diversi - ha riscontrato, in accordo col legame corrente. Definendo il futuro legame adesso, precedendo differenti implementazioni fornite da differenti venditori, tentiamo di evitare l'allargamento di questo problema.

Le maggiori differenze tra lo standard corrente ed il legame futuro sono in due aree: come maneggiare i riferimenti ad oggetti, e come il linguaggio delle interrogazioni è integrato all'interno del linguaggio di programmazione.

Ponendo l'attenzione sulla prima di queste, il legame corrente introduce una nuova forma per i riferimenti (Ref<T>) per classi persistenti e per le istanze transienti. Queste classi sono indicate come "classi capaci di persistenza" dalla Sezione 2.2 alla Sezione 2.5 per distinguerle dalle "classi C++". Risulta chiaro che il principio che la persistenza sia ortogonale al tipo è limitato, nel legame attuale, per queste classi capaci di persistenza. Una sintassi differente viene usata per riferirsi ad istanze delle classi capaci di persistenza rispetto a quella usata per riferirsi alle altre classi C++.

Il legame futuro, al contrario, permetterà ad ogni tipo, sia una classe C++ che un tipo definito dall'utente, di avere istanze sia persistenti che transienti e permetterà al programmatore di usare lo stesso puntatore e la sintassi referenziale C++ che è sempre stata usata, per referirsi ad istanze persistenti nello stesso modo di quelle transienti, e.g.,

```
Professor *p;
p->office_number="4818";
```

in aggiunta a

```
Ref<Professor> p;
p->office_number="4818";
```

La seconda area nella quale il legame futuro differisce dal legame corrente è nel sottolinguaggio delle interrogazioni. Il legame futuro consentirà ancora

maggior completa integrazione col C++, ma ancora a spese di una più sofisticata implementazione da parte del venditore ODBMS. Nel legame corrente un programmatore invoca una funzione di interrogazione e passa il testo dell'interrogazione come una stringa a questa funzione (si veda la Sezione 2.4). Il legame delle variabili del linguaggio di programmazione con le variabili interne alla stringa di interrogazione viene fatto come se si fosse nella funzione printf: $1 viene inserito nella stringa di interrogazione e si riferisce al valore dell'argomento passato dopo la stringa di interrogazione. Il legame degli argomenti alle variabili dell'interrogazione è per posizione, e.g, $2 si riferisce al valore del secondo argomento; e gli argomenti sono opzionali, così queste interrogazioni senza di essi o con pochi di essi non necessitano degli argomenti fittizi. Sta al programmatore vedere che i tipi degli argomenti passati combacino con quelli che ci si aspetta dalle variabili dell'interrogazione. Il risultato dell'interrogazione viene restituito come un insieme oppure attraverso un iteratore. Ancora, il linguaggio permette a funzioni arbitrarie definite dal programma di essere invocate all'interno del corpo dell'interrogazione, e questo richiede un interprete OQL abbastanza sofisticato che, durante l'esecuzione, localizzi, leghi ed esegua i metodi che implementano queste funzioni. Ci si aspetta però che molti sistemi possano restringere queste capacità.

Il legame futuro, al contrario, non richiederà che l'interrogazione venga passata ad una funzione come una stringa; invece estenderà la definizione di espressioni nella grammatica del linguaggio di programmazione di base per supportare l'identificazione di oggetti tramite la descrizione come se fossero nomi di variabili. Queste descrizioni sono nella forma di predicati definiti sugli attributi di oggetti, sulle loro relazioni con altri oggetti, e sui risultati di operazioni applicate agli oggetti stessi. Questo significa che le espressioni C++ possono essere usate liberamente all'interno del sottolinguaggio delle interrogazioni, e le espressioni del linguaggio delle interrogazioni possono essere incluse come sottoespressioni all'interno di normali espressioni C++. Non vi è la necessità di uno stile di stampa per i processi di legame degli argomenti, e le espressioni di interrogazione possono essere ottimizzate durante la compilazione del programma, invece che in esecuzione.

Si noti che il futuro legame col C++ è soltanto una semplificazione del linguaggio di legame per la funzionalità corrente definito in ODMG-93. L'accrescimento dello standard per includere maggiori funzionalità avanzate (e se necessario, sintassi per supportarle) sarà il soggetto di una futura revisione dello standard ODMG.

Il corpo di questa Sezione 2.6 discute le differenze tra il legame col C++

corrente e quello futuro. Un sommario delle differenze è il seguente:

- Tutti i tipi di primitive costruite internamente e i tipi definiti dal programmatore, possono avere istanze persistenti. Non è necessario designare un insieme speciale di classi capaci di persistenza.
- I puntatori C++ standard e i riferimenti possono essere utilizzati per referenziare oggetti; non è necessario introdurre un nuovo tipo di riferimento.
- La sintassi per creare e cancellare gli oggetti è pienamente concorde col C++:(1) l'allocazione automatica degli oggetti viene supportata per gli oggetti persistenti allo stesso modo delle richieste di allocazione del programma (new), e (2) l'operatore standard C++ delete può essere usato per oggetti persistenti e transienti, una forma separata per oggetti persistenti referenziati con la sintassi Ref<T> non è richiesta.
- Le interrogazioni possono essere integrate in modo più naturale nel linguaggio di programmazione.

Le prime tre differenze in questa lista sono discusse nella Sezione 2.6.1 (ODL). Le ultime due sono discusse nella Sezione 2.6.2 (OML). Per aiutare a distinguere i componenti ODL e OML del legame futuro dalle loro controparti del legame corrente, ci si riferisce ad essi come ODL/future e OML/future nel corpo di questa sezione.

2.6.1 ODL

La sola differenza tra le dichiarazioni di classi estese in C++ accettate dal pre-processore C++ per il legame corrente ed il pre-processore ODL/OML per il legame futuro è nell'uso della sintassi standard C++ per i puntatori (*) invece di avere il tipo riferimento Ref<T> introdotto nel legame corrente.

La definizione di classe seguente mostra la versione ODL/future della definizione di una classe.

```
class Professor {
public:
// proprieta':
 int            age;
```

```
 char *         name;
 Department *   dept     inverse Department::professors;
 Set<Student *> advisees inverse Student::advisor;
// operazioni:
 void       grant_tenure();
 void       assign_course(Course *);
private:
          ...
};
```

Le dichiarazioni di attributo sono le stesse nei due legami.

Le dichiarazioni di relazione differiscono soltanto per l'uso dei puntatori standard C++ invece che per la sintassi Ref<T>, e.g,

```
class Professor {
public:
 Department *   dept     inverse Department::professors;
 Set<Student *> advisees inverse Student::advisor;
      ...
};
```

invece di

```
class Professor {
public:
 Ref<Department>   dept     inverse Department::professors;
 Set<Ref<Student>> advisees inverse Student::advisor;
          ...
};
```

La dichiarazione delle operazioni differisce soltanto nel fatto che il legame futuro utilizza il puntatore ad oggetto standard C++ (*) e la sintassi (&) per i riferimenti ad oggetto. Non è necessario usare la forma Ref<T> per referenziare a "oggetti della base di dati".

2.6.2 OML

L'OML/future non è principalmente un insieme di nuove espressioni o istruzioni (eccettuata la nuova forma per i riferimenti) aggiunto al C++. Esso lascia invariate le forme sintattiche esistenti ma ne espande le interpretazioni. Invece di introdurre un sottolinguaggio separato di cui servirsi coi dati persistenti, il C++ OML espande l'interpretazione delle forme della sintassi esistente in modo che possano essere applicate uniformemente ai dati persistenti e transienti. Il linguaggio di programmazione C++ possiede già oggetti con due tempi di vita: quelli che vivono per il tempo di vita della procedura nella quale sono stati creati e (2) quelli che rimangono in vita per tutta la durata del processo. Queste differenze nel tempo di vita sono implicite in due differenti forme sintattiche usate per creare gli oggetti: un forma dichiarativa per quelli che vivono per la durata di una procedura ed una chiamata esplicita ad una funzione (**new**) per quelle che sopravvivono alla procedura. Un volta che gli oggetti con questi due tempi di vita sono stati creati, la sintassi usata per manipolarli è la stessa. Il legame ODBMS definito attraverso C++ OML semplicemente aggiunge un terzo tempo di vita per gli oggetti - persistenti. Un oggetto persistente è tale che sopravvive al processo che lo ha creato. Esso vive per il tempo di vita della base di dati in cui viene riposto. Una volta che un oggetto persistente è stato creato, il legame OML/future permette di referenziarlo e manipolarlo usando la stessa sintassi che il C++ utilizza per gli oggetti che vivono meno.

Il legame OML/future coinvolge soltanto:

1. Estensioni verso l'alto compatibili con la sintassi C++ per la crazione degli oggetti. Le estensioni permettono al programmatore di specificare un terzo tempo di vita (persistente) in aggiunta ai due già supportati dal linguaggio di programmazione.

2. Estensioni verso l'alto compatibili con la sintassi C++ per le espressioni con cui riferirsi agli oggetti. Le estensioni permettono la selezione degli oggetti basata sulle descrizioni di questi oggetti e sulle relazioni a cui essi partecipano invece che soltanto attraverso i nomi delle variabili a cui essi sono stati assegnati.

Il legame continua ad usare le due classi autocostruite definite dal legame corrente:

1. Un insieme di classi collezione autocostruite per supportare la capacità di interrogazione

2. Un insieme di classi transazione e basi di dati che supportano la concorrenza controllata, l'integrità e le garanzie di recupero del DBMS

L'insieme di funzioni membro esportate da queste classi è invocato utilizzando la sintassi C++ standard. Le sole estensioni alla sintassi sono legate alla creazione degli oggetti persistenti ed alle espressioni di interrogazione.

I dettagli di queste estensioni alla sintassi e le loro differenze con le loro equivalenti nel legame OML corrente sono spiegate nelle sottosezioni seguenti.

Creazione, Cancellazione e Riferimenti ad Oggetti

Il C++ supporta delle implicite creazioni di oggetti nell'ambito di una dichiarazione, come pure quelle esplicite, creazioni e cancellazioni di oggetti controllate dal programmatore usando gli operatori new e delete. Il legame OML corrente supporta soltanto le dichiarazioni esplicite di oggetti persistenti. Il legame futuro supporta creazioni implicite ed esplicite di oggetti persistenti.

Creazione di Oggetti

La creazione di oggetti persistenti viene fatta sovrapponendo l'operatore new affinché accetti un argomento aggiuntivo che permette la creazione di oggetti persistenti come pure di quelli transienti. Sono supportate due forme sintattiche per l'operatore. La seconda forma include una *storage pragma* che permette al programmatore di specificare in che modo gli oggetti appena allocati debbano essere posizionati rispetto agli altri oggetti. Per permettere a librerie di creare oggetti che possano essere usati uniformemente come oggetti di un qualunque tempo di vita, una variabile membro statica Database::transient_memory viene definita e può essere usata come valore per un argomento della base di dati. Un'invocazione dell'operazione new con questo valore di argomento risulterà come una normale allocazione dell'oggetto fuori dallo heap transiente del C++.

Sintassi:

```
class_name * pointer_name =
                    new (database* [,class_descriptor*])
                    class_name [[array_element]]
                    [new_initializer]
                    new (database*, object*
```

```
                    [,class_descriptor*]) class_name
                    [[array_element_count]] [new_initializer]
```

Esempi:

```
Student *s1 = new(university_database) Student:
Student *s1 = new(university_database,s2) Student:
Student *s1 = new(transient_memory) Student:
```

Cancellazione di Oggetti
Il legame OML/future mantiene il modello C++ per la cancellazione esplicita degli oggetti. La sola differenza tra la cancellazione di oggetti nel legame OML/current e quella nel legame OML/future è la sintassi della cancellazione esplicita. Il C++ definisce un operatore delete che viene usato per gli oggetti allocati fuori dello heap. Questa funzione richiede un void* come suo argomento e quindi non può essere usata nel legame corrente. Infatti il legame corrente definisce una funzione delete_object separata, per la cancellazione di oggetti persistenti. Siccome il legame OML/futuro usa i puntatori standard C++ per referenziare oggetti persistenti come per quelli transienti, non vi è necessità di una funzione delete_object aggiuntiva nel legame futuro. Il legame OML/futur utilizza semplicemente l'operatore delete dello standard C++ per tutti gli oggetti creati esplicitamente, senza badare al loro tempo di vita.

Riferimenti ad Oggetti
Nell'OML/future gli oggetti referenzieranno altri oggetti usando i puntatori standard C++ (*) oppure i riferimenti standard C++ (&). Queste forme possono essere usate per referenziare sia oggetti transienti che persistenti. I puntatori possono essere settati ad un valore nullo per indicare che essi non referenziano alcun oggetto. Tale valore nullo è definito 0 come in C++.

Questo significa che la sintassi C++ standard può essere usata per ricavare e settare il valore di attributi, per creare, cancellare ed attraversare relazioni, e per invocare operazioni; e.g.,

Esempi di assegnamento di una valore ad un attributo:

```
p.age=32;
```

```
x=p.age;
```

La prima istruzione assegna il valore 32 all'attributo age di p. La seconda istruzione assegna il valore dell'attributo age di p alla variabile x.

Esempi di creazione, cancellazione ed attraversamento di relazioni:
Date le definizioni:

```
Department *english_dept;
Student *   Sam;
Professor * p;
```

e considerando la classe Professor come mostrato nella sezione 2.2,

```
p->dept=english_dept;
p->dept=0;
p->advisees.insert_element(Sam);
p->advisees.remove_element(Sam);
Sam->advisor=p;
```

La prima di queste istruzioni crea una relazione uno-a-uno tra il professore denotato da p ed il dipartimento di inglese. La seconda istruzione cancella la relazione. La terza istruzione aggiunge Sam nell'insieme degli studenti consigliati da p. La quarta istruzione elimina Sam dall'insieme degli studenti consigliati da p. L'ultima istruzione ha lo stesso risultato della terza - nominalmente per creare una nuova relazione tra lo studente Sam ed il professore p. La sola differenza è che essa usa un'istruzione che risulta appropriata e vantaggiosa dal punto di vista di chiunque guardi alla base di dati dal punto di vista dello studente.

Esempio di invocazione di operazione:

```
s.register_for(c);
```

Interrogazioni

La forma basilare per una interrogazione integrata è:

collectionobject=collection[predicate]

L'espressione sul lato destro di questa istruzione seleziona dalla collezione quegli elementi che soddisfano il predicato.

I predicati possono essere definiti su attributi, relazioni oppure operazioni.

I predicati possono includere un solo termine oppure delle congiunzioni booleane di termini, e.g.,

```
Professor x;
Set<Professor *> ee_professors;
x=professors [name=="Guttag"];
ee_professors=professors{dept==electrical_engineering};
```

Le parentesi quadre, per esempio in professors[name==”Guttag”], sono usate per indicare che la sottoespressione dell'interrogazione dovrebbe restituire un solo valore. Le parentesi graffe, per esempio in professors{dept==electrical_engineering}, sono usate per indicare che la sottespressione dell'interrogazione dovrebbe restituire un insieme di oggetti.

Le interrogazioni possono anche attraversare una relazione tra oggetti, e.g.,

```
Set<Student *> guttags_students;
guttags_students=students {take Course
               [taught_by Professor [name=="Guttag"]]
                         };
```

Questa interrogazione restituisce un insieme di strudenti che partecipano al corso tenuto dal professore che si chiama Guttag.

La sintassi del predicato usata negli esempi di questa sezione rappresentano una restrizione della sintassi OQL completa. Altre restrizioni sono possibili, come il supporto dell'intera sintassi.

Questa è un'area del legame futuro in cui ci aspettiamo sia permesso al mercato di aiutarci a fare una scelta durante il tempo che il legame OML/future diventerà la parte principale del C++. La domanda basilare è se una sintassi simile a SQL o qualcosa di ancora più vicino all'inglese strutturato mostrato negli esempi di questa sottosezione vincerà.

Le interrogazioni non sono limitate agli oggetti persistenti; esse funzionano bene anche con oggetti transienti.

Trasazioni e Operazioni sulla Base di Dati

Le transazioni e le basi di dati nel legame OML futuro, sono modellate come le classi costruite internamente come per quello corrente. La sintassi per referenziare una istanza di questi tipi costruiti internamente è anch'essa la stessa per entrambi i legami, dato che né le transazioni, né gli oggetti "canale" usati per rappresentare la connessione tra un processo ed una base di dati sono oggetti persistenti. Nella versione corrente del Modello ad Oggetti, gli oggetti transienti e quelli canale sono transienti e quindi referenziati usando dei puntatori C++ in entrambi i legami.

L'operazione di commit setta i puntatori ad oggetti persistenti e ad oggetti di vita più breve al valore 0. Ogni puntatore tra oggetti stabilito all'interno di una transazione, da oggetti transienti ad altri persistenti, o viceversa, rimane valido dopo l'esecuzione dell'operazione checkpoint.

2.6.3 Problemi di Migrazione

Ci sono due cammini nel legame futuro: (1) evolve la definizione di operatore sovrapposto all'interno della definizione del linguaggio C++; (2) introduce un pre-processore che possa manipolare i corpi dei metodi C++ allo stesso modo delle dichiarazioni di classe.

I passi di pre-processing, compilazione e link richiesti per una implementazione di pre-processore di un legame futuro col C++ sono mostrati in Figura 2.12. Il diagramma è stato strutturato in modo tale che possa essere direttamente confrontato con l'analoga figura per il legame corrente, Figura 2.1. La sola differenza è che il pre-processore per il legame corrente si occupa soltanto della dichiarazione delle classi C++. Il pre-processore del legame futuro si occupa al corpo dei metodi che implementano le funzioni

membro definite sulle classi . Ovvero, nel linguaggio delle due figure, il legame corrente richiede un "pre-processore ODL", il legame futuro richiede un "pre-processore ODL/OML".

La ragione è che il pre-processore per il futuro legame deve occuparsi dei corpi dei metodi e che le istruzioni in questi metodi referenzino oggetti persistenti allo stesso modo degli operatori puntatore usati per referenziare oggetti transienti sempre presenti in memoria virtuale. Questi riferimenti devono essere sostituiti con chiamate di funzioni su di un tipo riferimento che possa determinare se l'oggetto referenziato è già in memoria virtuale uppure no. Se l'oggetto è transiente, la risposta sarà sempre si. Se l'oggetto è persistente, la risposta può essere si oppure no. Se la risposta è no, allora deve essere chiamata una operazione che porti l'oggetto in memoria virtuale prima di completare la dereferenziazione. La seconda ragione per cui si rende necessario il pre-processore è il permettere la stretta integrazione del linguaggio delle interrogazioni col linguaggio di programmazione di base.

Un grande sottoinsieme di questo legame - tutto eccetto l'integrazione delle espressioni di interrogazione con le espressioni C++ - può essere fatto con (1) un pre-processore solo ODL (come contemplato nel legame corrente) più alcune generalizzazioni di sovrapposizione di operatori per supportare gli operatori freccia ("->") e punto (".") nella definizione del C++, oppure (2) un'architettura di implementazione ODBMS che permetta il normale codice di dereferenziazione generato dal compilatore C++ per lavorare allo stesso modo con oggetto persistenti e transienti. Alcuni venditori possono supportare sistemi nei quali la creazione, la cancellazione ed il riferimento ad oggetti, lettura e settaggio dei valori degli attributi, attraversamento delle relazioni, e l'invocazione delle operazioni usano la sintassi standard C++ per lavorare sugli oggetti transienti, ma il loro linguaggio di interrogazione interno chiama la funzione oql passando l'interrogazione come una stringa.

Vi sono due cammini per spostare le applicazioni scritte nel legame corrente verso il legame futuro: (1) non fare cambiamenti, infatti le implementazioni che supportano il legame futuro continuano a supportare il legame corrente, oppure (2) usare tool che automaticamente traducano il codice sorgente al legame susseguente.

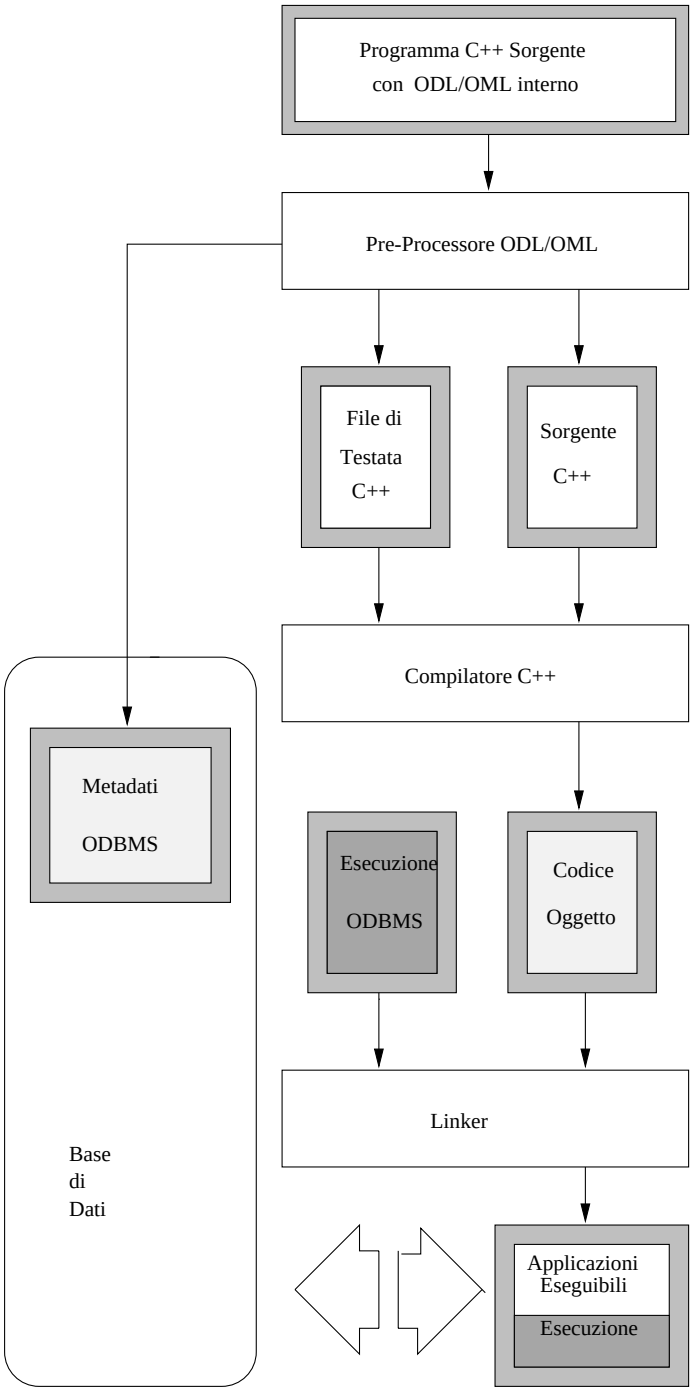

Figura 2.12: Passi di un Pre-Processore futuro

Capitolo 3

L'ambiente ODB-Tools Preesistente

Riporto nel seguito una breve descrizione dell'ambiente ODB-Tools preesistente dal quale il pre-processore ODL_{Rule}/C++ trae spunti e col quale coopera.

D'ora in avanti, salvo diverso avviso, ogni riferimento a ODL_{Rule}, deve intendersi come un riferimento a ODL esteso alla definitione di regole di integrità succintamente descritto in questo capitolo.

3.1 L'architettura di 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. Si basa su algoritmi che derivano da tecniche dell'intelligenza artificiale.

Il risultato della ricerca svolta nell'ambito di questo progetto è un prototipo che realizza l'ottimizzazione di schemi e l'ottimizzazione semantica delle interrogazioni.

In questa sezione si intende presentare in modo schematico ODB-Tools ed i suoi componenti maggiormente collegati al lavoro svolto in questa tesi[1].

In figura 3.1 sono rappresentati i vari moduli che compongono tale prototipo.

[1] Per una trattazione più approfondita si veda [Cor97]

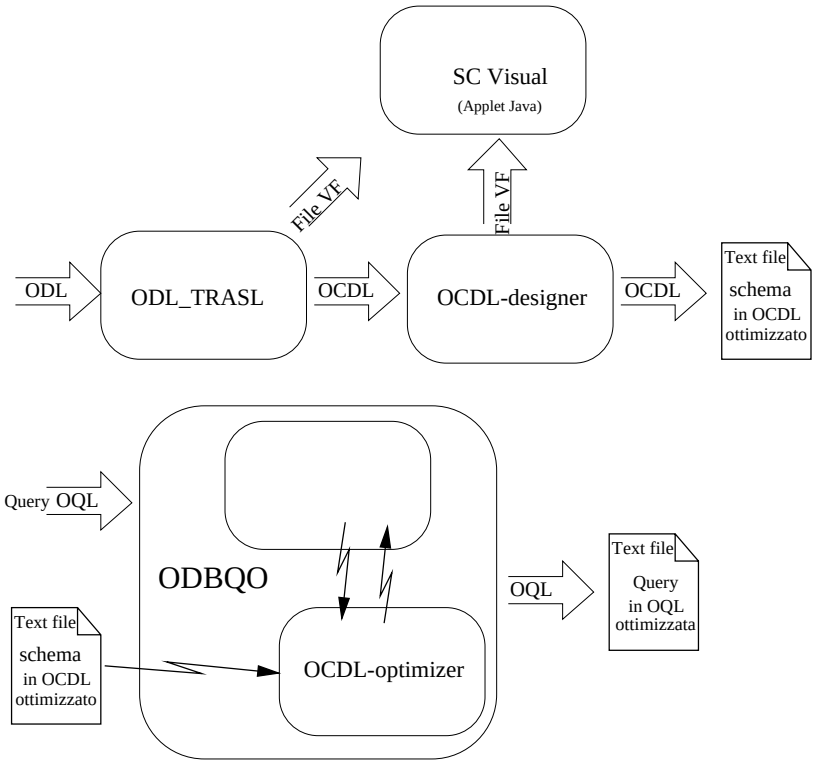

Figura 3.1: Componenti ODB-Tools

3.1.1 Schemi e regole di integrità

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

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

Per poter inserire nel modello *regole di integrità* è stata necessaria una sua estensione.

Le regole di integrità hanno la forma:

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

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

Riporto di seguito la sintassi delle regole di integrità di ODL$_{Rule}$ accettate dal pre-processore ODL$_{Rule}$/C++ :

```
< 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 < TypeSpec > |
                             < ForAll > < Identifier > in < DottedName > :
                             < RuleBodylist > |
                             exists < Identifier > in < DottedName > :
                             < RuleBodylist > |
                             < DottedName > =
                             < TypeSpec > < OpDclRule >
< RuleConstOp >          ::= = | >= | <= | < | >
< RuleCast >             ::= ( < TypeSpec > )
< DottedName >           ::= < Identifier > |
                             < Identifier > . < DottedName >
< ForAll >               ::= for all | forall
```

```
< OpDclRule >                ::=  < Identifier >
                                  (  < OptParameterListRule >  )
< OptParameterListRule >     ::=  [< ParameterListRule >]
< ParameterListRule >        ::=  < DottedName > |
                                  < DottedName >  ,
                                  < ParameterListRule >
```

Osservazioni sulla sintassi La sintassi appena presentata si discosta leggermente dalla sintassi ODL-estesa presentata in [Ric98] per quello che riguarda le operazioni.
Risulta comunque possibile fare la seguente considerazione:

- 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
  - ogni parametro, se specificato, si considera passato per valore in quanto non può e non deve essere modificato.

3.2 Il linguaggio OCDL

In questa sezione viene riportato qualche cenno al linguaggio OCDL in quanto parte dell'output del pre-processore ODL_{Rule}/C++ sarà utilizzato come input per ODL_Trasl[2] che traduce un sorgente ODL_{Rule} in formato OCDL.

OCDL (Object and Costraint Definition Language) estensione di ODL (del progetto ODB-Tools) con regole, è il linguaggio attraverso il quale è possibile descrivere uno schema di dati. OCDL è supportato da `OCDL-Designer` che *valida* ed ottimizza schemi di basi di dati.

[2]Si veda[Ric98]

3.2.1 Sintassi

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

`prim`: classe primitiva;

`virt`: classe virtuale;

`type`: tipo valore;

`btype` : tipo base;

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

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

Regole di integrità Come si è visto, una regola di integrità ha la forma

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

I tipi $tipo^a$ e $tipo^c$ vengono descritti con due definizioni separate, essi possono essere solo classi virtuali e tipi-valore.

Introduciamo cosí quattro nuove tipologie che descrivono le regole di integrità:

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

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

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

3.3 Validatore di schemi

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

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

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

Per ulteriori informazioni sul *validatore* vedere la tesi [Gar95].

Capitolo 4

Architettura Funzionale del Pre-Processore ODL$_{Rule}$/C++

Nello schema di figura 4.1, viene presentato lo schema generale dell'architettura del pre-processore ODL$_{Rule}$/C++ il quale riceve in input un file sorgente scritto secondo una opportuna sintassi ODL$_{Rule}$/C++ che consente anche l'utilizzo di dichiarazioni tipiche dell'ODL pur mantenendosi il più possibile vicina alla sintassi di un sorgente C++ standard.

4.1 Estensioni Rispetto al C++

Brevemente vengono riportate tutte le estensioni previste dal pre-processore ODL$_{Rule}$/C++ che non rientrano nel C++ standard.

Tipi Dati supportati: oltre a tutti quelli resi disponibili dal C++ si hanno i seguenti tipi resi disponibili sotto forma di classi di oggetti:

- Range
- Persistent_Object
- Collection
- Set
- Bag
- List
- Array
- Varray

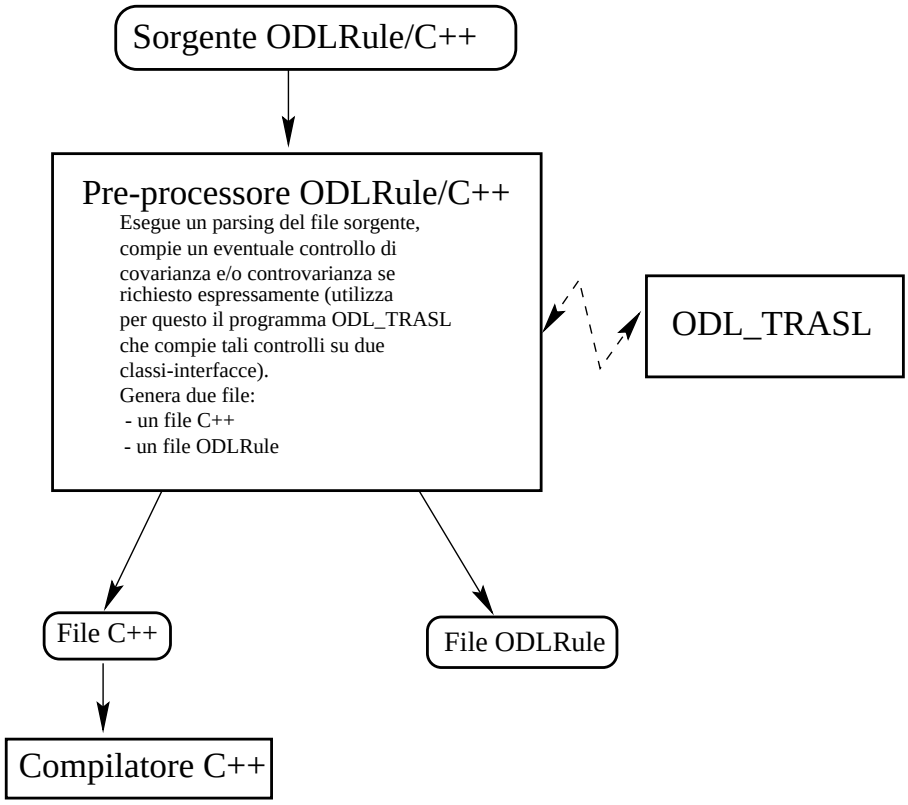

Figura 4.1: Schema generale del pre-processore ODL$_{Rule}$/C++ e del suo ambiente

Dichiarazioni delle chiavi di una classe, siano esse primarie oppure no.

Dichiarazioni delle estensioni delle classi.

Dichiarazioni delle relazioni di una classe , necessariamente tutte binarie.

Dichiarazioni delle Rule , relative alle varie classi di oggetti.

4.2 Tipi Dati introdotti

Di seguito viene presentata una breve ma esauriente spiegazione relativa ai nuovi tipi dati supportati dal pre-processore ODL_{Rule}/C++ [1].

Range: intervallo. Si tratta di un tipo dato che consente alle variabili di assumere un valore compreso tra quelli espressi dagli estremi del range. Attualmente i range sono relativi a valori interi e reali. Per semplicitá i range non sono definiti come una classe, bensí come un nuovo tipo di dato.

PersistentObject: serve per definire le interfacce delle classi che popolano la base di dati, le cui istanze si vuole rimangano memorizzate in modo persistente nella base di dati stessa.

Collection: è una classe astratta in C++ e non può avere istanze. Deriva da **Persistent_Object** e permette a classi concrete che a loro volta derivano da essa di possedere oggetti persistenti. La classe **Collection** contempla alcune semplici operazioni quali l'inizializzazione, l'introduzione e la rimozione di oggetti, il calcolo della cardinalitá ed altro ancora.

Set: é una collezione non ordinata di elementi che non permette duplicati. Deriva dalla classe **Collection** e possiede a sua volta operazioni tipiche degli insiemi quali l'unione, l'intersezione, la differenza, il controllo di uguaglianza.

Bag: risulta del tutto simile alla classe **Set**, senonché ammette elementi duplicati al proprio interno.

[1] Per un approfondimento si rimanda alle sezioni 2.3.4, 2.3.5 e 2.3.6

List: è una collezione ordinata di elementi e non permette duplicati. Anch'essa deriva dalla classe **Collection** e quindi ne eredita dati e metodi. Il primo valore dell'indice di una **List** è zero per seguire la convenzione del C e del C++.

Array: viene fornita una classe Array, derivata dalla classe **Collection** che permette di definire oggetti di tipo array ad una sola dimensione e di lunghezza assegnata (può contenere al più un certo numero prefissato di elementi) ed inoltre il suo indice inizia dal valore zero in accordo con la convenzione C e C++.

Varray: è un **Array** di lunghezza variabile, ma a differenza di questo possiede alcuni metodi aggiuntivi rispetto alla classe **Collection** da cui deriva anch'esso. Tali metodi permettono di alterare il numero di elementi del Varray quando questo è già stato creato.

4.3 ODL$_{Rule}$

Riferendosi ad ODL$_{Rule}$, si intende parlare del linguaggio ODL standard al quale è stata aggiunta la sintassi relativa alle regole di integrità.
Schematicamente si può quindi scrivere:

$$\text{ODL-Rule} = \text{ODL} + \text{Regole di Integrità}$$

Si è assunta come sintassi di tali regole quella definita nell'ambito dell'ambiente ODB-Tools come estensione del linguaggio ODL[2]. Questa scelta è stata fatta per evitare di introdurre una nuova sintassi equivalente a quella già esistente.

4.4 Architettura Funzionale del Pre-Processore ODL$_{Rule}$/C++

Passiamo ora ad una visione piú particolareggiata del pre-processore ODL$_{Rule}$/C++ e del suo reale funzionamento. Occorre premettere che, onde consentire la massima portabilitá possibile, si è preferito utilizzare il C come linguaggio di programmazione.
Come si vede dalla figura 4.2 lo schema del pre-processore ODL$_{Rule}$/C++

[2]Si veda la sezione 3.1.1

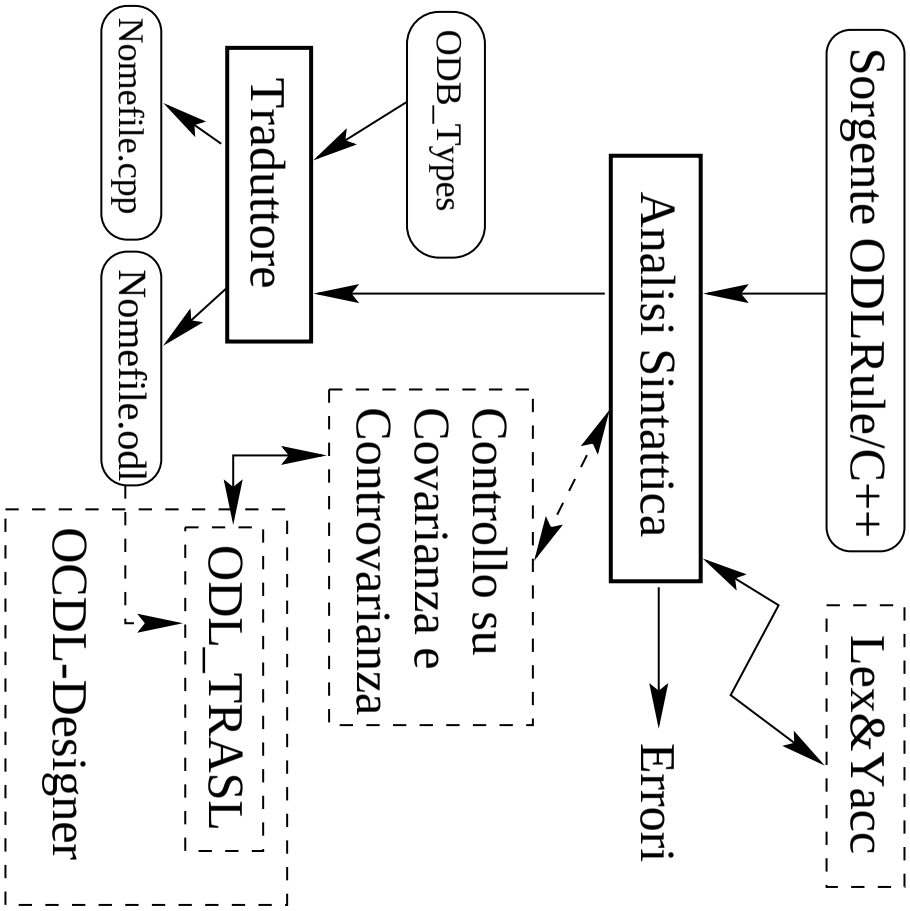

Figura 4.2: Schema a blocchi dell'architettura del pre-processore ODL$_{Rule}$/C++

risulta fortemente integrato all'ambiente che lo circonda. I componenti col bordo tratteggiato fanno parte di tale ambiente, mentre quelli a bordo continuo sono moduli del pre-processore ODL$_{Rule}$/C++ . I file, siano essi di input, di output o di libreria, hanno il bordo continuo con gli angoli arrotondati.

L'analisi sintattica è basata sull'utilizzo di LEX&YACC[3]. In particolare occorre leggere e capire il sorgente che viene accettato in ingresso in modo da poterlo poi tradurre. Se richiesto dall'utente tramite appositi parametri passati dalla linea di comando, è possibile richiedere il controllo di covarianza e/o controvarianza sui metodi delle varie classi definite nel sorgente. Attualmente questo controllo non è disponibile e per un approfondimento in tal

[3]Si veda l'Appendice A

senso si veda l'appendice B.

Nel caso siano riscontrati errori di sintassi o di qualunque altro genere, ne viene data notizia all'utente e l'esecuzione del programma ha termine. Viceversa, se tutto è andato bene, il programma passa alla parte successiva ovvero la traduzione vera e propria. La traduzione in linguaggio C++, viene effettuata grazie un file di libreria che definisce tutti i tipi dati e le classi introdotte (ODB_Types); il file C++ risultante non avrà quindi errori di compilazione dovuti alla traduzione, ma si potrebbero avere altri tipi di errori in quanto nessun controllo effettivo viene effettuato sulla correttezza dei dati introdotti, i.e., dichiarazione di due classi diverse con lo stesso nome.

Le relazioni e le dichiarazioni di chiavi vengono eliminate nel file C++ così ottenuto. Per quanto riguarda le regole di integrità esse possono essere trasformate in funzioni esterne o in metodi (operazioni) interni alle classi a cui le regole si riferiscono.

La traduzione in ODL$_{Rule}$, e quindi la produzione del file ODL$_{Rule}$, risulta abbastanza semplice in quanto la sintassi accettata dal pre-processore ODL$_{Rule}$/C++ si mantiene molto vicina alla sintassi ODL$_{Rule}$, pur consententendo varie estensioni verso il linguaggio C++. In genere è sufficiente sostituire o aggiungere particolari parole chiave.

4.4.1 Esecuzione del pre-processore ODL$_{Rule}$/C++ dalla linea di comando

L'utilizzo del pre-processore ODL$_{Rule}$/C++ deve seguire la seguente sintassi da digitare alla linea di comando:

odlcpp_prep nomefile [-v] [-t]

dove:

odlcpp_prep è il nome dell'eseguibile;

nomefile identifica il file sorgente scritto in linguaggio ODL$_{Rule}$/C++. Nessuna estensione deve essere specificata in quanto viene considerata per default l'estensione "**.cpo**". Questo parametro è necessario e la sua mancanza causa l'arresto immediato del programma;

-v parametro opzionale che richiede il controllo della covarianza;

-t parametro opzionale che richiede il controllo della controvarianza;

4.4.2 Files di Output

Il prodotto del pre-processore ODL_{Rule}/C++ , ammesso che non si siano verificati errori, risulta consistere, come giá detto in due file i cui nomi sono uguali a quello specificato come parametro, mentre ne varia l'estensione: .cpp per il file C++ e .odl per il file ODL_{Rule}.

Capitolo 5

Il Pre-processore ODL$_{Rule}$/C++

5.1 Il Traduttore

5.1.1 Struttura del programma

La sintassi di ODL estesa verso il linguaggio C++ è stata diffusa in formato *lex&yacc*[1], tale scelta è in accordo con la politica dell'ODMG di rendere ODL, e quindi tutti i suoi derivati, facili da implementare.

Alla sintassi Lex&Yacc sono state aggiunte:

1. Le *actions* della parte Yacc. Nelle actions vengono memorizzati tutti i concetti letti dal pre-processore ODL$_{Rule}$/C++ , vengono inoltre eseguiti alcuni controlli semantici.

2. Routine di stampa in formato ODL$_{Rule}$ e C++. Queste ultime comprendono le routine di traduzione vera e propria e le routine di trasformazione delle regole di integrità in metodi e funzioni C++.

Il programma è composto dai seguenti moduli

- modulo principale (`preprocm.c`):
  Si ha l'inizializzazione delle variabili globali, e chiama in sequenza: *il parser, la routine di traduzione in ODL$_{Rule}$, la routine di traduzione in C++.*

- parser della sintassi ODL$_{Rule}$/C++:
  È composto dal modulo *lex* (`preproc.l`) e dal modulo *yacc* (`preproc.y`).

[1] si veda appendice A

Svolge il controllo sintattico della sintassi ODL$_{Rule}$/C++ grazie alle routine generate con Lex&Yacc. Durante tale controllo riempie le strutture dati in memoria.

- routine di traduzione in ODL$_{Rule}$:
  Dalle strutture dati allocate dal parser estrae la forma ODL$_{Rule}$.

- routine di traduzione in C++:
  Dalle strutture dati allocate dal parser estrae la forma C++ oltre ad eseguire fisicamente la trasformazione delle regole di integrità dalla sintassi ODL$_{Rule}$ a metodi e funzioni C++.

Di seguito saranno descritte solo le parti principali del programma ed in particolar modo la trattazione delle regole di integrità al fine della loro trasformazione. Per eventuali chiarimenti sulla struttura originale del traduttore si veda la Tesi di laurea di S. Riccio [Ric98] alla quale la presente tesi si è ispirata.

5.1.2 La Sintassi ODL$_{Rule}$/C++

La sintassi ODL$_{Rule}$/C++ è fortemente legata a quella ODL$_{Rule}$, ma ne aumenta la flessibilità concedendo al progettista di basi di dati ad oggetti nuove potenzialità. Tale sintassi è stata ispirata dallo standard ODMG riportato nel capitolo 2 con alcune modifiche rese necessarie, a mio avviso, per la chiarezza espressiva della stesura di un sorgente.
Di seguito riporto soltanto le differenze rispetto a tale standard.

Dichiarazioni di File Inclusi

Sebbene lo standard non ne faccia parola, è di uso comune dichiarare inclusioni di file in un sorgente C++. A tale scopo il pre-processore ODL$_{Rule}$/C++ consente tali dichiarazioni e le riporta nel file C++ di output, mentre le ignora semplicemente durante la costruzione del file ODL$_{Rule}$.

Dichiarazioni di Classi di Oggetti

Onde rendere più chiara la dichiarazione di una classe, sono state aggiunte due nuove parole chiave: **relationship** e **operation** da porre rispettivamente davanti ad ogni dichiarazione di relazione e di operazione (per operazione si intenda ciò che cita lo standard ovvero metodo di una classe o funzione esterna).

Risulta anche prevista la possibilità di dichiarare chiavi relative alle classi. Discostandosi dallo standard per tenersi il più vicino possibile ad ODL, la sintassi per la dichiarazione di una classe prevede la dichiarazione della estensione della classe stessa nel modo seguente.

$$
\begin{aligned}
< \text{ClassDcl} > \quad ::= \quad & < \textbf{class} > \ < \text{Identifier} > \ [< \text{InheritanceSpec} >] \\
& [< \text{KeySpec} >] \ < \text{ExtentSpec} > \\
& \{ \ [< \text{ClassBody} >] \ \}; \\
< \text{ExtentSpec} > \quad ::= \quad & < \textbf{extent} > \ < \text{Identifier} >
\end{aligned}
$$

Dichiarazione di Regole di Integrità

Lo standard non le menziona neppure, quindi ho deciso di mantenere la sintassi ODL_{Rule}[2] per la dichiarazione di tali regole onde non costringere il progettista di basi di dati ad imparare una nuova sintassi del tutto equivalente ad una già esistente.

5.1.3 Le strutture Dati

Riporto di seguito alcune delle strutture dati nelle quali viene memorizzato lo schema di un sorgente ODL_{Rule}/C++.

Rappresentazione di una dichiarazione *include* Vediamo come sono memorizzate le informazioni relative alle dichiarazioni *include*.

```
struct s_include_list
{
  char type;   /* 'q' -> ""
                  'b' -> <> */
  char *nomefile;
  struct s_include_list *next;
} *Include_list;
```

Le informazioni vengono memorizzate sotto forma di lista di nomi di file e tipo di inclusione.

[2]Per un approfondimento dettagliato si vedano [Cor97] e [Ric98]

Rappresentazione di una *class* Vediamo come sono memorizzate le informazioni di una *class*.

```
struct s_class_type
{
  char type;  /* 'f' -> forward
                 'b' -> body
              */
  char *name;
  char *extent_name;
  struct s_iner_list *iner;
  char *key;
  struct s_prop_list *prop;
} *Class_type;
```

Il campo `iner` punta alla lista dei nomi delle classi da cui deriva la classe in questione.
Il campo `prop` punta alla lista delle proprietà della classe medesima: attributi, relazioni ed operazioni.

Rappresentazione delle *proprietà* Vediamo in che modo sono memorizzate le informazioni relative alle *proprietà* di una classe.

```
struct s_prop_list
{
  struct s_prop_type *p;
  struct s_prop_list *next;
} *Prop_list;

struct s_prop_type
{
  char type;     /* 'c' -> costante
                    't' -> tipo
                    'a' -> attributo
                    'r' -> relazione
                    'o' -> operazione
                 */
  union s_prop
  {
```

```
    char *c;
    struct s_type_type *t;
    char *a;
    struct s_rel_type *r;
    struct s_op_type *o;
  } prop;
} *Prop_type;
```

Per comodità si è preferito tenere separate la lista delle proprietà dalla singola proprietà. I campi dell'unione `prop` hanno il seguente significato:

- c: la proprietà è una costante e viene memorizzata in una stringa di caratteri che contiene anche il tipo ed il valore.
- t: la proprietà è una dichiarazione di tipo interna alla classe in questione.
- a: la proprietà è una costante o lista di costanti e viene memorizzata in una stringa di caratteri che contiene anche il tipo.
- r: la proprietà è una dichiarazione di relazione.
- o: la proprietà è una dichiarazione di operazione (metodo).

Rappresentazione delle *regole di integrità* Per la rappresentazione delle regole di integrità, accettando il pre-processore ODL_{Rule}/C++ la sintassi ODL_{Rule}, ho utilizzato le strutture relative al traduttore ODL_Trasl[3] con alcune leggere varianti:

```
struct s_rule_body_list
/* ---------------------------- */
/* gestione rules  */
{
  char type;
  /* flag che puo' valere:
   *   'c'  dichiarazione di costante
   *   'i'  dichiarazione di tipo
   *   'f'  la regola e' un forall o un'exist
   */
  char fg_ok;
```

[3]Si veda [Ric98]

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

```
    {
      /* dottedname e' la variabile su cui imporre
       * il tipo
       *  Nel caso di funzioni, 'type' rappresenta il
       * tipo di ritorno
      */
      char *type;         /* identif. tipo  */
                          /* lista dei parametri della funzione
                           * se NULL non si fa uso dei funzioni
                           */
      struct s_op_type *param_list;
    } i;
    struct
    {
      /* dottedname e' la lista su cui iterare */
      char fg_forall;    /*  puo valere:
                         *   'f'  forall
                         *   'e'  exist
                         *        significa che il
                         *        tipo e' un'exist
                         * questo flag e' stato
                         * introdotto in quanto i
                         * tipi EXIST e FORALL
                         * hanno moltissime cose
                         * in comune
                         */

      char *iterator;    /* nome iteratore */
      struct s_rule_body_list *body;
    } f;
  } r;
  struct s_rule_body_list *next;
} *Rule_body_list;
```

L'unica differenza rispetto alla struttura originale sta nell'attributo `param_list` che in questo caso risulta essere un puntatore al tipo `s_op_type` invece che al tipo `s_operation_param`. Il nuovo tipo possiede la seguente struttura:

```
struct s_op_type
{
  char *returnvalue;
  char *name;
```

```
  struct s_param_list *param;
} *Op_type;

struct s_param_list
{
  char *tipo;
  char *param;
  struct s_param_list *next;
} *Param_list;
```

Tale modifica è stata fatta in quanto la struttura `s_operation_param` risulta ridondante per gli scopi prefissati. Il significato dell'attributo rimane comunque invariato.

Per quel che concerne la struttura `s_rule_type`, sono state effettuate alcune semplificazioni:

```
struct s_rule_type
{
  char   *name;         /* nome della rule */
  struct s_rule_body_list *ante;
  struct s_rule_body_list *cons;
  /* lista globale di tutte le regole */
  struct s_rule_type      *next;
} *Rule_type;
```

Rispetto all'originale sono stati eliminati gli attirbuti `ocdlante` e `ocdlcons` in quanto inutili nell'applicazione sviluppata nella presente tesi.

5.2 Descrizione delle Funzioni

Utilizzando il parser generato da Lex&Yacc è possibile riempire le strutture dati in memoria con le informazioni relative allo schema presentato in input. Di seguito si presentano le funzioni che leggono queste strutture dati per eseguire la traduzione e la manipolazione di questi dati onde poi restituire in output il file in formato ODL$_{Rule}$ e quello in formato C++.

Faccio presente che nessun controllo di coerenza viene effettuato dal pre-processore ODL$_{Rule}$/C++ in quanto ridondante. Tali controlli vengono eseguiti da OCDL_Designer e dal compilatore C++ ai quali i file di output sono destinati. Viene invece garantita correttezza sintattica e grammaticale di

entrambi file prodotti.

Di seguito vengono riportate le principali funzioni che realizzano l'algoritmo di traduzione.
Sono due le funzioni fondamentali: una si occupa della traduzione del sorgente in ODL$_{Rule}$; l'altra produce l'equivalente C++ e converte le regole di integrità in operazioni.

Funzione per la Traduzione in ODL$_{Rule}$

La funzione principale di traduzione è `traduci_odl` ed è contenuta nel modulo `preodl.c` assieme a tutte le funzioni minori specializzate. Essa esegue una lettura sequenziale della struttura di definizioni presente in memoria dopo che è stato effettuato il parsing. A seconda del tipo di definizione viene richiamata una specifica sottofunzione specializzata a trattare il genere di dato in questione.

- Costanti: viene semplicemente riportata la dichiarazione ODL$_{Rule}$/C++.
- Classi: viene richiamata la funzione `print_class_odl`.
- Regole di Integrità: viene richiamata la funzione `print_rule_odl`.
- Tipi Dato: viene richiamata la funzione `print_type_odl`.
- Operazioni: viene richiamata la funzione `print_op_odl`.

La sintassi ODL$_{Rule}$/C++ risulta volutamente simile alla sintassi ODL$_{Rule}$ di modo che le operazioni di traduzione risultino semplificate.

Sono da citare comunque alcune notevoli differenze:

- Lista di Ereditarietà: la sintassi ODL$_{Rule}$/C++ prevede che in tale lista vi sia la specifica del tipo di ereditarietà (`public`, `private` o `protected`). Durante la traduzione, tale tipo di ereditarietà viene omesso.
- Liste di Attributi: nella sintassi ODL$_{Rule}$/C++ viene consentita una dichiarazione di attributi tipica del C++, secondo la sintassi:

$$
\begin{aligned}
&< \text{AttrDcl} > \;::=\; < \text{TypeSpec} > \; < \text{IdentifierList} > \\
&< \text{IdentifierList} > \;::=\; < \text{Identifier} > \;| \\
&\qquad\qquad\qquad\quad < \text{Identifier} > \;\;,\;\; < \text{IdentifierList} >
\end{aligned}
$$

Di conseguenza, al fine di poter tradurre correttamente la volontà dichiarativa del progettista, è stato necessario tradurre il dato di questo tipo nella sintassi ODL$_{Rule}$:

< AttrDcl > ::= < **attribute** > < TypeSpec > < Identifier >

ripetendo tale dichiarazione tante volte quanto dovuto.

- Tipi Dato: il C++ (ed anche ODL$_{Rule}$/C++) impone di dichiarare variabili struttura anteponendo al nome del tipo la parola chiave **struct** mentre questo viene considerato un errore nella sintassi ODL$_{Rule}$. La funzione `printf_attr_odl` e le sottofunzioni ad essa collegate, eliminano tale parola chiave.

- Liste di Parametri: la sintassi ODL$_{Rule}$/C++ consente il passaggio ad una funzione di parametri per valore e per riferimento rifacendosi alla sintassi tipica del C++ e quindi utilizzando l'operatore & posto d'innanzi al nome di un parametro per indicare che esso deve essere passato per riferimento e non per valore.
  La funzione `print_op_odl` compie appunto tale controllo ed inserisce la parola chiave **in** se non si riscontra la presenza di &, in caso contrario viene introdotta la parola chiave **inout** in accordo con la sintassi ODL$_{Rule}$[4].

  Un'osservazione risulta evidente:
  utilizzando il pre-processore ODL$_{Rule}$/C++ non sarà mai possibile dichiarare parametri di sola uscita per una funzione come invece previsto da ODL$_{Rule}$. Il solo parametro di sola uscita risulta essere il valore di ritorno della funzione stessa.

Un'ultima osservazione: avendo accettato la sintassi ODL$_{Rule}$ per la dichiarazione delle regole di integrità, essa viene riportata quasi immutata in uscita. In realtà il significato della regola resta comunque inalterato, ma a causa della memorizzazione della sua struttura, aventuali condizioni poste in **and** tra loro vengono riportate nel file di output in ordine invertito.

[4] Si veda [Ric98]

Esempio Di seguito riporto un esempio di traduzione, da un sorgente ODL_{Rule}/C++ ad un file in sintassi ODL_{Rule} corretta, compiuta dal preprocessore ODL_{Rule}/C++ .

```
#include <stream.h>

struct Indirizzo{
 string via,numero,citta;
};

class Persona
keys codfis
extent Persone
{
 string nome,cognome,codfis;
 struct Indirizzo indirizzo;
 relationship Persona sposato_con inverse Persona::sposato_con;

};

class Manager:public Persona
extent Managers
{
 range {20000,140000} salario;
 range {1,13} livello;
 set<Persona> collaboratori;
 range {0,200000} premi;
 int anno_assunzione;

 relationship set<Azienda> lavora_per inverse Azienda::dipendente;

 operation range {0,200000} premio(int anno_assunzione,int contratti);
};

class TopManager:public Manager
extent TopManagers
{
 range {10,13} livello;
};

class Azienda
keys partita_iva
extent Aziende
```

```
{
 string nome,partita_iva;
 struct Indirizzo indirizzo;

 relationship set<Manager> dipendente inverse Manager::lavora_per;
};
```

La cui traduzione risulta essere:

```
struct Indirizzo{
string  via,numero,citta;
};
interface Persona
{
attribute string  nome;
attribute string  cognome;
attribute string  codfis;
attribute Indirizzo indirizzo;
relationship Persona sposato_con inverse Persona::sposato_con;
};
interface Manager:Persona
{
attribute range {20000,140000} salario;
attribute range {1,13} livello;
attribute set<Persona> collaboratori;
attribute range {0,200000} premi;
attribute int  anno_assunzione;
relationship set<Azienda> lavora_per inverse Azienda::dipendente;
range {0,200000} premio(in int  anno_assunzione,in int  contratti);
};
interface TopManager:Manager
{
attribute range {10,13} livello;
};
interface Azienda
{
attribute string  nome;
attribute string  partita_iva;
attribute Indirizzo indirizzo;
relationship set<Manager> dipendente inverse Manager::lavora_per;
};
```

Funzione per la Traduzione in linguaggio C++

Occorre fare un discorso preliminare.
Il C++ standard non consente la dichiarazioni di oggetti di tipo collezione quali **set, bag, list**, ecc. Lo standard presentato nel capitolo 2 introduce però le definizioni di tali strutture proprio per un legame con C++. È stato quindi necessario creare un file di libreria C++ contenente tali definizioni oltre al concetto di `range` che ODL$_{Rule}$/C++ accetta.

La funzione principale di traduzione è `traduci_cpp` ed è contenuta nel modulo `precpp.c` assieme a tutte le funzioni minori specializzate. Per prima cosa essa inserisce nel file di output una inclusione del file di libreria "`cpolib.cpp`", quindi aggiunge aventuali inclusioni accessorie definite dal progettista di basi di dati nel file sorgente.

Conseguentemente viene fatta una prima scansione sequenziale dei dati presenti in memoria onde creare nuove strutture di dati di servizio utilizzate nella traduzione e nella gestione delle regole di integrità.

Noti i nomi di tutte le classi, viene scritto sul file di output una intestazione speciale, richiesta dal compilatore C++ della GNU, che ne consentirà poi la compilazione. Tale intestazione consiste nella definizione, per ogni classe dichiarata, di speciali tipi relativi alle classi collezione.
La funzione esegue quindi una seconda lettura sequenziale della struttura di definizioni presente in memoria dopo che è stato effettuato il parsing. A seconda del tipo di definizione viene richiamata una specifica sottofunzione specializzata a trattare il genere di dato in questione similmente a quanto avviene per la traduzione in ODL$_{Rule}$.

- Costanti: viene semplicemente riportata la dichiarazione ODL$_{Rule}$/C++.
- Classi: viene richiamata la funzione `print_class_cpp`.
- Regole di Integrità: viene richiamata la funzione `print_rule_cpp`.
- Tipi Dato: viene richiamata la funzione `print_type_cpp`.
- Operazioni: viene richiamata la funzione `print_op_cpp`.

Anche in questo caso sono da citare alcune notevoli differenze tra il sorgente ODL$_{Rule}$/C++ ed il file C++ di output:

- Dichiarazioni di Relazioni fra le Classi: tali dichiarazioni non vengono riportate nel file di output in quanto non sono gestibili direttamente tramite il linguaggio C++.

- Classi e Collezioni di Classi: come già detto, il C++ non supporta la gestione di classi collezione. In accordo con lo standard e con le esigenze compilative del C++, vengono fatte alcune sostituzioni. Quando un nome di una classe viene utilizzato per definire un attributo, a tale nome viene sostituito un riferimento a tale classe. In modo analogo si opera con le collezioni di classi.

  Esempio:

```
class Person
{
};

class Manager:Person
{
 Person mother;
 Set<Person> children;
};
```

  viene tradotto in:

```
class Person
{
};

class Manager:Person
{
 RerPerson mother;
 RefSetRefPerson children;
};
```

- Regole di integrità: vengono trasformate in funzioni e metodi interni alle classi per le quali vengono specificate. Ai dettagli di tale trasformazione verrà dedicato la prossima sezione.

Esempio Di seguito riporto un esempio di traduzione, da un sorgente ODL_{Rule}/C++ ad un file in linguaggio C++, compiuta dal pre-processore ODL_{Rule}/C++ . Il sorgente ODL_{Rule}/C++ è lo stesso dell'esempio relativo alla traduzione in ODL_{Rule}.

```
#include "cpolib.cpp"
#include<stream.h>
class Persona;
typedef Ref<Persona> RefPersona;
typedef Set<RefPersona> SetRefPersona;
typedef Ref<SetRefPersona> RefSetRefPersona;
typedef Bag<RefPersona> BagRefPersona;
typedef Ref<BagRefPersona> RefBagRefPersona;
typedef List<RefPersona> ListRefPersona;
typedef Ref<ListRefPersona> RefListRefPersona;
typedef Array<RefPersona> ArrayRefPersona;
typedef Ref<ArrayRefPersona> RefArrayRefPersona;
typedef Varray<RefPersona> VarrayRefPersona;
typedef Ref<VarrayRefPersona> RefVarrayRefPersona;
class Manager;
typedef Ref<Manager> RefManager;
typedef Set<RefManager> SetRefManager;
typedef Ref<SetRefManager> RefSetRefManager;
typedef Bag<RefManager> BagRefManager;
typedef Ref<BagRefManager> RefBagRefManager;
typedef List<RefManager> ListRefManager;
typedef Ref<ListRefManager> RefListRefManager;
typedef Array<RefManager> ArrayRefManager;
typedef Ref<ArrayRefManager> RefArrayRefManager;
typedef Varray<RefManager> VarrayRefManager;
typedef Ref<VarrayRefManager> RefVarrayRefManager;
class TopManager;
typedef Ref<TopManager> RefTopManager;
typedef Set<RefTopManager> SetRefTopManager;
typedef Ref<SetRefTopManager> RefSetRefTopManager;
typedef Bag<RefTopManager> BagRefTopManager;
typedef Ref<BagRefTopManager> RefBagRefTopManager;
typedef List<RefTopManager> ListRefTopManager;
typedef Ref<ListRefTopManager> RefListRefTopManager;
typedef Array<RefTopManager> ArrayRefTopManager;
typedef Ref<ArrayRefTopManager> RefArrayRefTopManager;
typedef Varray<RefTopManager> VarrayRefTopManager;
```

```
typedef Ref<VarrayRefTopManager> RefVarrayRefTopManager;
class Azienda;
typedef Ref<Azienda> RefAzienda;
typedef Set<RefAzienda> SetRefAzienda;
typedef Ref<SetRefAzienda> RefSetRefAzienda;
typedef Bag<RefAzienda> BagRefAzienda;
typedef Ref<BagRefAzienda> RefBagRefAzienda;
typedef List<RefAzienda> ListRefAzienda;
typedef Ref<ListRefAzienda> RefListRefAzienda;
typedef Array<RefAzienda> ArrayRefAzienda;
typedef Ref<ArrayRefAzienda> RefArrayRefAzienda;
typedef Varray<RefAzienda> VarrayRefAzienda;
typedef Ref<VarrayRefAzienda> RefVarrayRefAzienda;
struct Indirizzo{
string  via,numero,citta;
};
class Persona
{
public:
string  nome,cognome,codfis;
struct Indirizzo indirizzo;
static const char * const  extent_name;
};
static RefSetRefPersona Persone;
class Manager:public Persona
{
public:
IRange  salario;
IRange  livello;
RefSetRefPersona collaboratori;
IRange  premi;
int  anno_assunzione;
IRange  premio(int  anno_assunzione,int  contratti);
static const char * const  extent_name;
};
static RefSetRefManager Managers;
class TopManager:public Manager
{
public:
IRange  livello;
static const char * const  extent_name;
};
static RefSetRefTopManager TopManagers;
```

```
class Azienda
{
public:
string  nome,partita_iva;
struct Indirizzo indirizzo;
static const char * const  extent_name;
};
static RefSetRefAzienda Aziende;
```

Occorre osservare che il file di output in linguaggio C++ non contiene i corpi delle funzioni e dei metodi, ed inoltre non contiene la funzione "**main**" e quindi non è direttamente compilabile. Risulta necessario creare un file C++ entro il quale includere il prodotto della traduzione.

5.3 Traduzione delle regole di Integrità

Con riferimento alla sintassi ODL_{Rule}/C++ presentata nella sezione 3.1.1, viene qui mostrato l'algoritmo di traduzione di regole di integrità in una sintassi ODL_{Rule} ed in una funzione C++ standard.

5.3.1 La struttura dati

Di seguito viene presentata nel dettaglio la struttura nella quale sono memorizzati i dati riguardanti una regola di integrità.

```
struct s_rule_type
{
   char   *name;          /* nome della rule */
   struct s_rule_body_list *ante;
   struct s_rule_body_list *cons;
   /* lista globale di tutte le regole */
   struct s_rule_type      *next;
}
```

La struct `s_rule_type` serve per memorizzare le varie regole di integrità. Tutte le regole di integrità dichiarate in uno schema sono unite in una lista. Come si vede, una `rule` è descritta da una parte antecedente ed una conseguente. Le due condizioni antecedente e conseguente sono descritte come liste di condizioni.

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

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

```
                */
               char *operator;
               char *cast;        /* NULL se manca il cast */
               char *value;
        } c;
        struct
        {      /* in questo caso
                * dottedname e' la variabile su cui imporre
                * il tipo
                */
               char *type;         /* identif. tipo  */
               /* puntatore alla operazione */
               struct s_op_type *param_list;
        } i;
        struct
        {      /* in questo caso
                * dottedname \'e la lista su cui iterare
                */
               char fg_forall;    /*  puo valere:
                                   *   'f'  forall
                                   *   'e'  exists
                                   *        significa che il
                                   *        tipo \'e un'exists
                                   * questo flag \'e stato
                                   * introdotto in quanto i
                                   * tipi EXISTS e FORALL
                                   * hanno quasi la stessa traduzione
                                   * in comune
                                   */
               char *iterator;    /* nome iteratore */
               struct s_rule_body_list *body;
        } f;
   } r;
   struct s_rule_body_list *next;
}
```

Descrizione della struttura :

- **type** indica il tipo di parametro, il quale puo' valere:
  - 'c' dichiarazione di costante
  - 'f' la regola è un forall o un'exists
  - 'i' dichiarazione di tipo
    In questo caso particolare, puo' essere inserita nella condizione una

dichiarazione di operazione. Infatti nella struttura **i** è presente un campo, denominato **param_list**, che punta alla lista dei parametri di una operazione (**s_op_type**).

5.3.2 Funzioni di gestione delle regole di integrità

Di seguito sono presentate le funzioni che realizzano l'algoritmo di trattazione delle regole di integrità.

- **print_rule_odl**
  La funzione `print_rule_odl` effettua semplicemente la trascrizione della rule in linguaggio ODL$_{Rule}$ senza effettuare alcun tipo di controllo (ogni controllo viene lasciato al tool OCDL_Designer che accetta in ingresso il file generato dal pre-processore ODL$_{Rule}$/C++).
  Questa funzione scandisce semplicemente la struttura dati in memoria onde ricostruire la regola dichiarata.
  Si può osservare che in caso di condizioni multiple poste sugli attributi di un oggetto, a causa della particolare struttura dati utilizzata, esse vengono riportate in uscita in ordine inverso. Questo non pone problemi eccessivi per quel che riguarda il significato della regola di integrità in quanto le varie condizioni sono poste in "`and`" logico tra loro.

- **print_rule_cpp**
  La funzione `print_rule_cpp` trasforma una regola di integrità in una funzione C++ con lo stesso nome, che restituisce un valore "`vero`" se le condizioni imposte sono rispettate, un valore "`falso`" in caso contrario. Tale valore viene contenuto in una variabile chiamata "**flag**", inizialmente posta a 1 ("`vero`"), che fornirà poi il valore da restituire. Alla descrizione nel dettaglio dell'algoritmo espletato da questa funzione viene dedicata la prossima sezione del capitolo.

5.3.3 Algoritmo di Trasformazione delle Regole di Integrità

Occorre osservare che per motivi di coerenza con la sintassi ODL$_{Rule}$/C++, si è deciso di trasformare una regola di integrità in una funzione esterna alle classi anzichè in un metodo relativo alla classe interessata. Nulla vieta però di effettuare la scelta opposta. In questa sezione verranno anche brevemente descritte le modifiche all'algoritmo necessarie per la generazione di metodi

anzichè funzioni.

Onde chiarire al meglio l'algoritmo di trasformazione, riporto di seguito alcuni esempi spiegando passo passo come l'algoritmo interpreta i dati e agisce di conseguenza.

Come scrivere una regola di integrità: vediamo alcune regole semplici ma particolarmente esplicative:

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

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* `AgiatePerson` ".

```
rule r1 forall X in Person:
      ( X in AgiatePerson )
   then
      X.taxes = "High".
```

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

Dot notation (nomi con punti)

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

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

interface Class2()
```

```
   {
   attribute real           c2a1;
   attribute Class1         c2a2;
   };

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

Dato un oggetto X di tipo `Class3` si ha che:
`X.c3a1`: è di tipo `long`, fa riferimento direttamente all'attributo definito nella classe `Class3`.
`X.c3a2`: è un oggetto della *classe* `Class2`.
`X.c3a2.c2a1`: è di tipo `real`, fa riferimento all'attributo definito nella classe `Class2`, questo è possibile in quanto l'attributo `c3a2` è un oggetto della classe `Class2`.
`X.c3a2.c2a2.c1a1`: è di tipo `string`, e fa riferimento all'attributo definito nella classe `Class1`.
Nelle rule sono possibili operazioni di confronto tra valori, infatti se si scrive `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.

A tale proposito, siccome in C++ ci si riferisce agli oggetti delle classi dichiarate tramite puntatori di tipo "`Ref`" (si veda a tale proposito il capitolo 2), è stato necessario creare una apposita funzione **dta()** (acronimo di "dot to arrow", ovvero "dal punto alla freccia") che sostituisce ogni punto presente in un parametro con una freccia (operatore C "->" che referenzia un tipo dato tramite un puntatore).

Esempio:

`X.c3a2.c2a2.c1a`
diventa:
`X->c3a2->c2a2->c1a1`

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 AgiatePerson`
ove X individua un oggetto, la condizione è *vera* se X fa parte dell'estensione di `AgiatePerson`

La funzione di traduzione risale alla estensione della classe AgiatePerson che supponiamo si chiami AgiatePeople (sempre definita) e quindi semplicemente sostituisce tale costrutto con

```
AgiatePeople->contains_element(X)
```

OSS: il metodo "`contains_element`" appartiene alla classe "`Collection`" definita nello standard (si veda la Sezione 2.3.6).

Allo stesso modo:

`X1.is_section_of in Course`
ove X1 individua un oggetto, la condizione è verificata se X1 fa parte di `Course`.

Supponendo Courses estensione di Course la funzione sostituisce tale condizione con:

```
Course->contains_element(X1->is_section_of)
```

- 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`, intero o reale. La condizione è verificata se `X.age` ha un valore compreso tra 18 e 25 inclusi.

La funzione riconosce il tipo range e lo sostituisce con una doppia condizione in sintassi C++:

```
X->age>=18 && x->age<=25
```

- 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.

Esempi:

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

Analogamente al caso riguardante i tipi, la funzione di trasformazione sostituisce tali condizioni con le equivalenti C++.
Si osservi che il segno di uguaglianza "`=`" viene sostituito con il corrispondente operatore C++ "`==`" e la congiunzione "**and**" con l'equivalente "`&&`".

```
X->tass == "High"
X->age > 18
X->salary > lo_limit && X->salary < hi_limit
```

Un'altra osservazione risulta necessaria. Il confronto tra due stringhe in C++ viene eseguito attraverso la funzione di libreria "**strcmp**", ma nello standard viene dichiarata una classe "**String**" che contempla fra i suoi metodi, l'operatore "`==`" sovrapposto e quindi viene permesso il controllo di uguaglianza sopra riportato.

- 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 )
```

In questo caso, la funzione di trasformazione costruisce un ciclo "`while`" che permette la iterazione all'interno della classe collezione specificata. Viene fatto ancora uso dei metodi definiti per le classi collezione dello standard. La funzione esegue una nuova iterazione delle classi dichiarate onde ricavare il tipo di collezione dell'attributo di X. A questo punto viene dichiarata una variabile X1 del tipo ricavato ed un iteratore (appartenente alla classe template **Iterator**) definita sulla collezione `X.teaches`.
Per quanto riguarda le condizioni, esse vengono trattate richiamando una funzione dedicata.

```
RefProfessor X1;
Iterator<RefProfessor> it=X->teaches->create_iterator();
while (it.next(X1))
{
 if (!CONDIZIONE) return 0; /* valore falso */
};
```

Viene restituito un valore falso se anche una sola delle instanze contenute in `X.teaches` non rispetta la condizione.

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`

Anche questo caso, la funzione di trasformazione costruisce un ciclo "`while`" che permette la iterazione all'interno della classe collezione specificata. Viene ancora fatto ancora uso dei metodi definiti per le classi collezione dello

standard. La funzione esegue una nuova iterazione delle classi dichiarate onde ricavare il tipo di collezione dell'attributo di X. A questo punto viene dichiarata una variabile X1 del tipo ricavato ed un iteratore (appartenente alla classe template **Iterator**) definita sulla collezione `X.teaches`.
Per quanto riguarda le condizioni, esse vengono trattate richiamando una funzione dedicata.

```
RefProfessor X1;
Iterator<RefProfessor> it=X->teaches->create_iterator();
while (it.next(X1))
{
 if (CONDIZIONE) return 1; /* valore vero */
};
```

Viene restituito un valore vero se almeno una delle instanze contenute in `X.teaches` rispetta la condizione.

Esempio completo

```
#include <stream.h>

struct Indirizzo{
 string via,numero,citta;
};

class Persona
keys codfis
extent Persone
{
 string nome,cognome,codfis;
 struct Indirizzo indirizzo;
 relationship Persona sposato_con inverse Persona::sposato_con;

};

class Manager:public Persona
extent Managers
{
 range {20000,140000} salario;
 range {1,13} livello;
```

```
 set<Persona> collaboratori;
 range {0,200000} premi;
 int anno_assunzione;

 relationship set<Azienda> lavora_per inverse Azienda::dipendente;

 operation range {0,200000} premio(int anno_assunzione,int contratti);
};

class TopManager:public Manager
extent TopManagers
{
 range {10,13} livello;
};

class Azienda
keys partita_iva
extent Aziende
{
 string nome,partita_iva;
 struct Indirizzo indirizzo;

 relationship set<Manager> dipendente inverse Manager::lavora_per;
};


rule m1 forall X in Manager:
(X in TopManager)
then
(X.salario>=100000);

rule m2 forall X in Manager:
(X.salario in range {40000,80000})
then
 exists X1 in X.collaboratori:
 (X1.nome="Mario" and X1.cognome="Rossi");

rule m3 forall X in Manager:
(X.cognome="Neri")
then
 X.premi=range {0,200000} premio(X.anno_assunzione,30);
```

La versione ODL$_{Rule}$ risultante è:

```
struct Indirizzo{
string  via,numero,citta;
};
interface Persona
{
attribute string  nome;
attribute string  cognome;
attribute string  codfis;
attribute Indirizzo indirizzo;
relationship Persona sposato_con inverse Persona::sposato_con;
};
interface Manager:Persona
{
attribute range {20000,140000} salario;
attribute range {1,13} livello;
attribute set<Persona> collaboratori;
attribute range {0,200000} premi;
attribute int  anno_assunzione;
relationship set<Azienda> lavora_per inverse Azienda::dipendente;
range {0,200000} premio(in int  anno_assunzione,in int  contratti);
};
interface TopManager:Manager
{
attribute range {10,13} livello;
};
interface Azienda
{
attribute string  nome;
attribute string  partita_iva;
attribute Indirizzo indirizzo;
relationship set<Manager> dipendente inverse Manager::lavora_per;
};
rule m1 forall X in Manager:
X in TopManager
then (X.salario>=100000);
rule m2 forall X in Manager:
X.salario in range {40000,80000}
then exists X1 in X.collaboratori:
(X1.cognome="Rossi" and
X1.nome="Mario");
rule m3 forall X in Manager:
(X.cognome="Neri")
then X.premi = range {0,200000} premio(X.anno_assunzione,30);
```

La versione C++ risultante è:

```
#include "cpolib.cpp"
#include<stream.h>
class Persona;
typedef Ref<Persona> RefPersona;
typedef Set<RefPersona> SetRefPersona;
typedef Ref<SetRefPersona> RefSetRefPersona;
typedef Bag<RefPersona> BagRefPersona;
typedef Ref<BagRefPersona> RefBagRefPersona;
typedef List<RefPersona> ListRefPersona;
typedef Ref<ListRefPersona> RefListRefPersona;
typedef Array<RefPersona> ArrayRefPersona;
typedef Ref<ArrayRefPersona> RefArrayRefPersona;
typedef Varray<RefPersona> VarrayRefPersona;
typedef Ref<VarrayRefPersona> RefVarrayRefPersona;
class Manager;
typedef Ref<Manager> RefManager;
typedef Set<RefManager> SetRefManager;
typedef Ref<SetRefManager> RefSetRefManager;
typedef Bag<RefManager> BagRefManager;
typedef Ref<BagRefManager> RefBagRefManager;
typedef List<RefManager> ListRefManager;
typedef Ref<ListRefManager> RefListRefManager;
typedef Array<RefManager> ArrayRefManager;
typedef Ref<ArrayRefManager> RefArrayRefManager;
typedef Varray<RefManager> VarrayRefManager;
typedef Ref<VarrayRefManager> RefVarrayRefManager;
class TopManager;
typedef Ref<TopManager> RefTopManager;
typedef Set<RefTopManager> SetRefTopManager;
typedef Ref<SetRefTopManager> RefSetRefTopManager;
typedef Bag<RefTopManager> BagRefTopManager;
typedef Ref<BagRefTopManager> RefBagRefTopManager;
typedef List<RefTopManager> ListRefTopManager;
typedef Ref<ListRefTopManager> RefListRefTopManager;
typedef Array<RefTopManager> ArrayRefTopManager;
typedef Ref<ArrayRefTopManager> RefArrayRefTopManager;
typedef Varray<RefTopManager> VarrayRefTopManager;
typedef Ref<VarrayRefTopManager> RefVarrayRefTopManager;
class Azienda;
typedef Ref<Azienda> RefAzienda;
typedef Set<RefAzienda> SetRefAzienda;
typedef Ref<SetRefAzienda> RefSetRefAzienda;
```

```
typedef Bag<RefAzienda> BagRefAzienda;
typedef Ref<BagRefAzienda> RefBagRefAzienda;
typedef List<RefAzienda> ListRefAzienda;
typedef Ref<ListRefAzienda> RefListRefAzienda;
typedef Array<RefAzienda> ArrayRefAzienda;
typedef Ref<ArrayRefAzienda> RefArrayRefAzienda;
typedef Varray<RefAzienda> VarrayRefAzienda;
typedef Ref<VarrayRefAzienda> RefVarrayRefAzienda;
struct Indirizzo{
string  via,numero,citta;
};
class Persona
{
public:
string  nome,cognome,codfis;
struct Indirizzo indirizzo;
static const char * const  extent_name;
};
static RefSetRefPersona Persone;
class Manager:public Persona
{
public:
IRange  salario;
IRange  livello;
RefSetRefPersona collaboratori;
IRange  premi;
int  anno_assunzione;
IRange  premio(int  anno_assunzione,int  contratti);
static const char * const  extent_name;
};
static RefSetRefManager Managers;
class TopManager:public Manager
{
public:
IRange  livello;
static const char * const  extent_name;
};
static RefSetRefTopManager TopManagers;
class Azienda
{
public:
string  nome,partita_iva;
struct Indirizzo indirizzo;
```

```
static const char * const  extent_name;
};
static RefSetRefAzienda Aziende;


int m1 ()
{
int flag=1;
RefManager X;
Iterator<RefManager> it=Managers->create_iterator();
while (it.next(X))
{
if(TopManagers->contains_element((RefTopManager)X))
{

if(!((X->salario>=100000))) flag=0;
};
};
return flag;
};



int m2 ()
{
int flag=1;
RefManager X;
Iterator<RefManager> it=Managers->create_iterator();
while (it.next(X))
{
if(X->salario>=40000 && X->salario<=80000)
{

RefPersona X1;
Iterator<RefPersona> it=X->collaboratori->create_iterator();
while (it.next(X1))
{
if(((X1->cognome=="Rossi" &&
X1->nome=="Mario"))) return true;
};
};
};
return flag;
```

```
};



int m3 ()
{
int flag=1;
RefManager X;
Iterator<RefManager> it=Managers->create_iterator();
while (it.next(X))
{
if((X->cognome=="Neri"))
{

if(!(X->premi ==  X->premio(X->anno_assunzione,30))) flag=0;
};
};
return flag;
};
```

Cenni sulla Trasformazione di Regole di Integrità in Metodi

Tenendo presente l'esempio appena presentato, e riferendosi ad esempio alla funzione `m2()`, si può osservare che il blocco di istruzioni interno al ciclo `while` più esterno, si riferisce all'istanza `X` della classe `Manager`. Tali istruzioni potrebbero quindi formare il corpo di un metodo della classe `Manager` con poche varianti, quali la sostituzione di ogni riferimento all'oggetto `X` con l'operatore `this` del C++.

Affinchè la regola venga applicata a tutte le istanze di una classe, occorre comunque che tale metodo venga invocato per ognuna di esse e quindi si avrebbe comunque la presenza di una funzione esterna che effettua tale operazione.

Capitolo 6

Note Conclusive

La presente tesi ha realizzato un ambiente software, ispirato all'approccio standard ODMG-93, per i progettisti di basi di dati. Tale ambiente sfrutta un nuovo linguaggio di programmazione ibrido, progettato e sviluppato nell'ambito della tesi stessa. Tale linguaggio risulta possedere sia le potenzialità del linguaggio di alto livello ODL che le potenzialità espressive del linguaggio C++.

Tale ambiente software consente:

- la scrittura di un file sorgente che segua la sintassi ODL_{Rule}/C++;
- la generazione di due file di output, uno in linguaggio C++, l'altro in linguaggio ODL_{Rule}, sintatticamente corretti;
- la trattazione delle regole di integrità, espresse in sintassi ODL_{Rule}, direttamente da un ambiente C++, effettuando la trasformazione di tali regole in funzioni o metodi in linguaggio C++.

Inoltre è prevista la predisposizione al controllo di covarianza e controvarianza sui metodi delle classi di oggetti dichiarate.

6.1 Sviluppi Futuri

Il prodotto realizzato in questa tesi può e dovrebbe essere migliorato per fornire maggiori aiuti al progettista di basi di dati.

Alcuni sviluppi potrebbero riguardare:

- l'interfaccia utente: attualmente tale interfaccia è pressoché nulla, limitandosi alla riga di comando;
- l'estensione ulteriore del linguaggio ODL_{Rule}/C++: il linguaggio riconosciuto dal pre-processore ODL_{Rule}/C++ ha alcuni limiti dovuti all'ambiente che lo circonda e allo standard seguito. Sarebbe utile proiettarsi verso un legame futuro tra ODL_{Rule} e C++ come auspicato dallo standard stesso.

Appendice A

Lex & Yacc

Lex e *Yacc* sono due utility molto usate per la realizzazione di analizzatori sintattici. Di seguito è riportata una breve descrizione dei due programmi.

In realtà in questa tesi sono stati utilizzati altri due programmi *flex&bison*, diversi ma compatibili con Lex&Yacc.

Flex e Bison sono due strumenti software che facilitano la scrittura di programmi in linguaggio C per l'analisi e l'interpretazione di sequenze di caratteri che costituiscono un dato testo sorgente.
Entrambi questi strumenti, partendo da opportuni file di specifica, generano direttamente il codice in linguaggio C, che può quindi essere trattato allo stesso modo degli altri moduli sorgenti di un programma.

Flex

Flex legge un file di specifica che contiene delle espressioni regolari per il riconoscimento dei token (componenti elementari di un linguaggio) e genera una funzione, chiamata **yylex()**, che effettua l'analisi lessicale del testo sorgente.

La funzione generata estrae i caratteri in sequenza dal fusso di input. Ogni volta che un gruppo di caratteri soddisfa una delle espressioni regolari viene riconosciuto un token e, di conseguenza, viene invocata una determinata azione, definita opportunamente dal programmatore.
Tipicamente l'azione non fa altro che rendere disponibile il token identificato al riconoscitore sintattico. Per spiegare meglio il meccanismo di funzionamento ricorriamo ad un esempio: l'individuazione, nel testo sorgente, di un numero intero

```
[0-9]+          {
                  sscanf( yytext, "%d", &yylval );
                  return( INTEGER );
                }
```

l'espressione regolare `[0-9]+` rappresenta una sequenza di una o più cifre comprese nell'intervallo `{0-9}`. La parte compresa tra parentesi `{...}` specifica invece, in linguaggio C, l'azione che deve essere eseguita.
In questo caso viene restituito al parser il token `INTEGER` poiché è stato riconosciuto un numero intero.

Bison

Bison è un programma in grado di generare un parser in linguaggio C partendo da un file di specifica che definisce un insieme di regole grammaticali.

In particolare Bison genera una funzione, chiamata **yyparse()**, che interpreta una sequenza di token e riconosce la sintassi definita nel file di input.
La sequenza di token può essere generata da un qualunque analizzatore lessicale; di solito però Bison viene utilizzato congiuntamente a Flex.

Il vantaggio principale che deriva dall'utilizzo di Bison è la possibilità di ottenere un vero e proprio parser semplicemente definendo, in un apposito file, la sintassi da riconoscere.
Ciò avviene utilizzando una notazione molto simile alla *Bakus-Naur Form* (BNF).
Occorre però notare che i parser generati in questo modo sono in grado di riconoscere soltanto un certo sottoinsieme di grammatiche, dette *non contestuali*. A prima vista ciò potrebbe sembrare una limitazione; in realtà questo tipo di grammatica è in genere sufficiente[1] per definire la sintassi di un linguaggio di programmazione.

Per illustrare meglio il funzionamento di questo software utilizziamo un esempio di un possibile input per Bison:

```
var_declaration:    VAR var_list ':' type_name ';' ;
variable_list:      variable_name |
                    variable_list ',' variable_name ;
variable_name:      STRING ;
type_name:          INTEGER | FLOAT | BOOLEAN ;
```

[1] una trattazione più dettagliata e formale è data in [ASU86, FRJL88]

Ogni regola consiste di un nome, o simbolo non terminale, seguito da una definizione, che presenta a sua volta uno o più simboli terminali o non terminali (ovvero nomi di altre regole).

I simboli terminali, rappresentati nell'esempio in carattere maiuscolo, sono i token ottenuti dal riconoscitore lessicale. Il riconoscimento della grammatica avviene con un procedimento di tipo *bottom-up*[2], includendo ogni regola che viene riconosciuta in regole più generali, fino a raggiungere un particolare simbolo terminale che include tutti gli altri.
A questo punto il testo sorgente è stato completamente riconosciuto e l'analisi sintattica è terminata.

In realtà un parser deve svolgere anche altri compiti, come l'analisi semantica e la generazione del codice. Per questo motivo Bison consente al programmatore di definire un segmento di codice, detto *azione*, per ogni regola grammaticale.
Ogni volta che una regola viene riconosciuta il parser invoca l'azione corrispondente, permettendo, ad esempio, di inserire i nomi delle variabili nella symbol table durante l'analisi della sezione dichiarativa di un linguaggio:

```
var_declaration:     VAR var_list ':' type_name ';'
                     {
                         Push( $2 );
                     }
                     ;
```

Nell'esempio illustrato `Push()` è una funzione in linguaggio C che si occupa di inserire una lista di variabili nella `symbol table`.
Il codice che si occupa della traduzione vera e propria può allora essere integrato nel parser attraverso il meccanismo delle azioni semantiche.

Ulteriori informazioni si possono reperire in rete (GNU GENERAL PUBLIC LICENSE) la documentazione di flex&bison. Il libro di riferimento per Lex&Yacc è *Lex&Yacc[MB91]*.

[2]descritto ampiamente in [MB91]

Appendice B

Principio di covarianza e controvarianza di metodi

Il pre-processore ODL$_{Rule}$/C++ illustrato in questa tesi, offre la possibilità di effettuare un controllo di covarianza e/o controvarianza sui metodi delle classi dichiarate in ODLRule/C++ e sui metodi aggiuntivi dovuti alla traduzione delle regole di integrità, avvalendosi dell'ausilio di un modulo di ODL-Designer. Tale modulo, al momento della presentazione della tesi, non risulta ancora disponibile, ma l'architettura funzionale del pre-processore ODL$_{Rule}$/C++ ne prevede comunque la presenza (seppur facoltativa) e quindi guarda al futuro quando una cooperazione sarà resa possibile.

B.1 Principio di covarianza e controvarianza

In letteratura sono presentati due approcci contrastanti per controllare la consistenza delle interfaccie dei metodi:

1. Un primo approccio applica ai parametri delle operazioni il medesimo principio applicato agli attributi delle classi, ossia il principio di covarianza.

2. Un secondo approccio applica ai parametri delle operazioni il principio opposto a quello applicato agli attributi delle classi, ossia il principio di controvarianza.

Di seguito viene riportata la definizione di sottotipo secondo la teoria dei tipi di dati(E. Bertino e L.D. Martino [EM92]; Bruce and Wegner 1986 [BW86]; Cardelli 1984 [Car84]; Albano et al. 1985 [ACO85]).

Definizione 1 (sottotipo) *Un tipo t e' sottotipo del tipo t' ($t \leq t'$) se:*
(1) le proprieta' di t' sono un sottoinsieme di quelle di t
(2) per ogni operazione m'_t di t' esiste la corrispondente operazione m_t di t tale che:
(a) m_t e m'_t hanno lo stesso nome
(b) m_t e m'_t hanno lo stesso numero di argomenti
(c) l'argomento i di m'_t e' un sottotipo dell'argomento i di m_t **(regola di controvarianza)**
(d) m_t e m'_t restituiscono un valore o entrambi non hanno alcun parametro di ritorno (e) se m_t e m'_t restituiscono un valore allora il tipo del risultato di mt e' un sottotipo del tipo del risultato di m'_t **(regola di covarianza)**

In base a tale definizione è valido il *principio di sostituibilita'*, secondo il quale una istanza di un sottotipo puo' essere sostituita da un supertipo in ogni contesto nel quale quest'ultimo puo' essere legalmente usato.
Altri gruppi di ricerca, come gli autori che hanno sviluppato il sistema O_2 [BDK96], hanno adottato la regola di covarianza anche sui singoli parametri delle operazioni: in questo modo però si possono produrre degli errori in fase di run-time a fronte di una correttezza sintattica.

In Bruce and Wegner 1986 [BW86] si riporta una dimostrazione rigorosa di quanto affermato sopra, sottolineando soprattutto la differenza esistente tra i tipi degli attributi e i tipi dei parametri formali delle operazioni.

Per capire questa affermazione si propone un semplice esempio:

Suppongo di modellare l'aspetto geometrico dei punti del piano cartesiano. I punti sono degli oggetti definiti da due coordinate (x,y), e possono essere positivi oppure negativi.
I punti positivi sono contenuti interamente nel primo quadrante del piano cartesiano, mentre quelli negativi nel terzo quadrante.
Introduco una operazione denominata "disegna" con le due coordinate come parametri di input.

```
interface Punto
 (extent Punti)
{
 attribute range{-1000,1000} x;
 attribute range{-1000,1000} y;
 void disegna (in range{-1000,1000} px,in range{-1000,1000} py);
```

```
};
interface Punto_positivo : Punto
 (extent Punti_positivi)
{
 attribute range{0,1000} x;
 attribute range{0,1000} y;
 void disegna (in range{0,1000} px,in range{0,1000} py);
};
```

In questo esempio ho applicato la regola di covarianza anche sui due parametri dell'operazione (metodo di O_2). Supponiamo di invocare l'operazione disegna(-10,-10) su un generico oggetto della classe "Punto", se questo oggetto e' anche istanza di "Punto_positivo" allora verra' eseguita l'operazione nella classe piu' specializzata, causando un errore di run-time.
Da un punto di vista pratico questo esempio esclude il principio di covarianza e giustifica solo in parte il principio di controvarianza applicato ai parametri delle funzioni. Per evitare errori in fase di esecuzione delle operazioni, e' sufficiente dichiarare il tipo dei parametri delle sottoclassi **uguale** a quello delle superclassi. Infatti nell'esempio riportato di seguito non vi e' alcun motivo per ampliare il range {0,1000} dei parametri dell'operazione Punto_positivo::disegna.

```
interface Punto
 (extent Punti)
{
 attribute range{-1000,1000} x;
 attribute range{-1000,1000} y;
 void disegna (in range{-1000,1000} px,in range{-1000,1000} py);
};
interface Punto_positivo : Punto
 (extent Punti_positivi)
{
 attribute range{0,1000} x;
 attribute range{0,1000} y;
 void disegna (in range{-1000,1000} px,in range{-1000,1000} py);
};
```

In conclusione il principio di controvarianza e' stato adottato per motivi di flessibilita', infatti risulta piuttosto vincolante imporre, nella signature delle operazioni, l'uso degli stessi tipi dichiarati nelle superclassi.

Bibliografia

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.

[BDK96] F. Bancilhon, C. Delobel, and P. Kanellakis. *building an Object-Oriented Database System, the story of O_2*. Morgan Kaufmann Publishers, Inc., 1996.

[BW86] Kim B. Bruce and P. Wegner. An algebraic model of subtypes in object-oriented languages. In *SIGPLAN Notices*, pages 163–172, 1986.

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

[Cat94] R.G.G. Cattell. *The Object Database Standard: ODMG-93, Release 1.1*. Morgan Kaufmann Publishers, Inc., 1994.

[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.

[EM92] Bertino E and L.D. Martino. *Sistemi di Basi di Dati 0rientate agli Oggetti*. Addison∼Wesley Masson, Milano - Italia, 1992.

[FRJL88] Charles N. Fischer and Jr. Richard J. LeBlanc. *Crafting a Compiler*. The Benjamin/Cumming Publishing Company, Inc., 1988.

[Gar95] Alessandra Garuti. Ocdl-designer: un componente software per il controllo di consistenza di schemi di basi di dati ad oggetti con vincoli di integrita'. Tesi di Laurea, Facolta' di scienze matematiche fisiche e naturali, corso di laurea di scienze delle informazioni, dell'universita' di Bologna, 1995.

[MB91] T. Mason and D. Brown. *Lex & Yacc*. O'Reilly & Associates Inc., 1991.

[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.