ODL-Designer$_{UNISQL}$: un'interfaccia per la specifica dichiarativa di vincoli di integritá in OODBMS*

S. Bergamaschi[1,2] D. Beneventano,[1,2] F. Sgarbi,[1] M. Vincini[1]

E-mail: {sonia,benevent,sgarbi,vincini}@dsi.unimo.it

[1] Dipartimento di Scienze dell'Ingegneria, Università di Modena,
Via G. Campi 213/B, I-41100 Modena
[2] CSITE-CNR
Viale Risorgimento 2, I-40136 Bologna

Abstract. La specifica ed il trattamento dei vincoli di integritá rappresenta un tema di ricerca fondamentale nell'ambito delle basi di dati; infatti, spesso, i vincoli costituiscono la parte piú onerosa nello sviluppo delle applicazioni reali basate su DBMS. L'obiettivo principale del componente software ODL-Designer$_{UNISQL}$, presentato nel lavoro, è quello di consentire al progettista di basi di dati di esprimere i vincoli di integritá attraverso un linguaggio dichiarativo, superando quindi l'approccio degli OODBMS attuali che ne consente l'espressione solo attraverso procedure (metodi e trigger). ODL-Designer$_{UNISQL}$ acquisisce vincoli dichiarativi e genera automaticamente, in maniera trasparente al progettista, le "procedure" che implementano tali vincoli. Il linguaggio supportato da ODL-Designer$_{UNISQL}$ è lo standard ODL-ODMG opportunamente esteso per esprimere vincoli di integrità, mentre l'OODBMS commerciale utilizzato è UNISQL.

1 Introduzione

Le Basi di Dati Orientate ad Oggetti, OODB (Object Oriented Database), sono attualmente oggetto di intensi sforzi di ricerca e sviluppo. La motivazione principale per questo interesse deriva dal fatto che il paradigma orientato ad oggetti offre una più ricca capacità di descrivere la conoscenza relativa ad un dominio di applicazione. Il modello dei dati è infatti molto più ricco e flessibile per la descrizione della conoscenza strutturale di classi di oggetti e permette, inoltre, la rappresentazione del *comportamento* di oggetti e classi. D'altro canto, il limite attuale del modello proposto negli OODBMS, é quello di consentire di esprimere i vincoli di integrità solo attraverso procedure, sfruttando oggetti di sistema quali trigger e metodi che obbligano il progettista a realizzare, soprattutto nei casi di vincoli articolati, routine "ad hoc" scritte in un opportuno linguaggio, abbastanza complesse e di difficile comprensione.

* La ricerca è stata parzialmente finanziata dal progetto INTERDATA - MURST 40% 97-98.

In questo lavoro viene presentato uno strumento software, ODL-Designer$_{UNISQL}$, che premette al progettista di definire tutte le entità della base di dati, compresi i vincoli di integrità, in modo dichiarativo e di generare automaticamente l'implementazione della conoscenza descritta attraverso metodi e trigger su un OODBMS commerciale.
L'Universitá di Modena e Reggio Emilia ha iniziato a sviluppare negli ultimi anni un progetto, denominato ODB-Tools (si veda al proposito il sito web all'indirizzo *http://sparc20.dsi.unimo.it/*), che consente di definire uno schema di base di dati ad oggetti che include vincoli di integritá espressi in maniera dichiarativa e di effettuare alcune rilevanti attività per le basi di dati, quali il controllo di consistenza di schemi e l'ottimizzazione semantica di interrogazioni.
In questo lavoro ci proponiamo, dapprima, di mostrare brevemente sia le estensioni di ODL per esprimere vincoli di integritá che le funzionalitá principali di ODB-Tools e dei suoi componenti, ODL-Designer per il progetto di schemi e ODBQ-Optimizer per l'ottimizzazione di query. Successivamente analizzeremo nel dettaglio il componente ODL-Designer$_{UNISQL}$, che dota ODL-Designer di un'interfaccia di comunicazione con l'OODBMS commerciale UNISQL, la quale consente di esprimere vincoli di integrità in forma dichiarativa e di generare automaticamente metodi e trigger per implementarli in un OODBMS commerciale. Tale interfaccia assume particolare rilevanza dal momento che permette al progettista di creare su piattaforma UNISQL uno schema di database, limitandosi a descriverne ogni sua parte in forma dichiarativa utilizzando il linguaggio di definizione degli schemi proposto dallo standard ODMG [7] ed opportunamente esteso per poter esprimere i vincoli di integritá.
L'interfaccia semplifica notevolmente lo sviluppo di applicazioni, soprattutto per quanto riguarda la traduzione dei vincoli di integritá in procedure di controllo (per approfondimenti circa la trattazione dei vincoli di integritá nelle basi di dati ad oggetti si rimanda a [1, 6, 9]). La tecnologia attuale dei sistemi OODBMS impone infatti l'utilizzo, per la realizzazione di questi ultimi, di oggetti di sistema quali trigger e metodi, costringendo perció il progettista a scrivere procedure software dedicate, utilizzando appositi linguaggi "embedded", che risultano abbastanza complesse e di difficile comprensione, soprattutto nel caso di vincoli di integritá articolati. Ed é perció proprio in base a queste considerazioni che si puó comprendere l'utilitá dell'interfaccia proposta, la quale consente di implementare gli schemi rimanendo ad un livello esclusivamente dichiarativo di definizione dello schema stesso. La scelta di un OODBMS target costituisce solo un "banco di prova" dell'approccio adottato; lo sviluppo di interfacce per altri OODBMS porterebbe alla riscrittura di una parte marginale del software realizzato che puó essere considerato quindi di validitá generale.
Il lavoro é articolato come segue. Nella sezione 2 vengono presentate le estensioni proposte al linguaggio ODL-ODMG (la sintassi é presentata in appendice A). Nella sezione 3 vengono descritte brevemente le funzionalitá di ODB-Tools, con particolare riferimento al componente ODL-Designer che supporta il controllo di consistenza di schemi database. La sezione 4 contiene il principale contributo di questo lavoro, cioé la generazione automatica di metodi/trigger per l'implemen-

tazione di vincoli di integritá dichiarativi su OODBMS, e descrive il componente ODL-Designer$_{UNISQL}$. La sezione 5 mostra alcuni esempi di utilizzo del database implementato su UNISQL per il controllo di consistenza degli oggetti inseriti. Infine, coclusioni e lavoro futuro sono descritti in sezione 6.

2 Estensioni del linguaggio ODL-ODMG con vincoli di integritá e classi virtuali

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

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

La sintassi di ODL estende quella dell'Interface Definition Language, il linguaggio sviluppato nell'ambito del progetto *Common Object Request Broker Architecture* (CORBA) [10].
Nel progetto ODB-Tools, l'ODL standard é stato esteso aggiungendo le seguenti caratteristiche:

Vincoli di integritá (*Rule*): ODL-ODMG non consente di esprimere regole. Abbiamo quindi esteso ODL con regole "if ... then" che permettono di esprimere in maniera dichiarativa vincoli di integritá. L'estensione sintattica proposta é molto intuitiva: ogni regola inizia con la parola chiave `rule`, seguita dalla dichiarazione della parte antecedente, poi la parola `then` e la parte conseguente. La regola é quantificata univocamente sugli elementi dell' estensione di un'interface. Ad esempio la seguente regola:

```
rule R1: forall X in Professor: X.rank = "Full"
         then                   X.annual_salary > 70000;
```

impone che ogni professore di rango pari a "*Full*" abbia un salario annuale superiore a 70000 dollari.
Sia l'antecedente sia il conseguente di una rule sono poi a loro volta costituiti da una lista di condizioni; i costrutti che possono apparire in tali liste sono:

Condizioni di appartenenza ad una classe: Esprimono l'appartenenza di un oggetto all'estensione di una classe.
Si noti che una condizione di questo tipo viene sempre vista come "-passiva"; in altre parole essa non porta in nessun caso alla migrazione dell'oggetto in esame da una classe dello schema ad un'altra.

Condizioni sul tipo di un attributo: Esprimono restrizioni sul dominio di attributi di tipo `integer` o `float`. Ad esempio la condizione X.age in

range {18,25} impone che il valore dell'attributo *age* dell' oggetto **X** sia un intero compreso tra 18 e 25[3].

Condizioni sul valore di un attributo: Implicano restrizioni sul valore di attributi dello schema.
Ad esempio la condizione X.rank = "Full" impone che il valore dell'attributo *rank* dell' oggetto **X** sia la stringa "*Full*".
Si noti che gli operatori di disuguaglianza quali ad esempio "<" o ">" possono essere usati solo con attributi di tipo `integer` o `float`, poiché le disuguaglianze vengono tradotte attraverso condizioni sul tipo `range`.

Condizioni su collezioni ed estensioni: Esprimono condizioni congiuntive su tutti gli oggetti di una collezione. È possibile utilizzare le clausole **forall** oppure **exists**, seguite dal nome della collezione e dalla lista di condizioni da verificare sulla collezione stessa. In particolare la clausola **forall** é vera quando tutti gli elementi della collezione soddisfano tutte le condizioni espresse nella relativa lista. Al contrario la clausola **exists** é vera quando almeno un elemento della collezione soddisfa tutta la relativa lista di condizioni.

Viste o classi virtuali (*View*): OLCD introduce la distinzione tra *classe virtuale*, la cui descrizione rappresenta una condizione necessaria e sufficiente di appartenenza di un oggetto del dominio ad una classe (corrispondente quindi alla nozione di vista) e *classe base (o primitiva)*, la cui descrizione rappresenta solo una condizione necessaria (corrispondente quindi alla classica nozione di classe). In altri termini, l'appartenenza di un oggetto all'interpretazione di una classe base deve essere stabilita esplicitamente, mentre l'interpretazione delle classi virtuali è calcolata sulla base della loro descrizione. Le classi base sono introdotte tramite la parola chiave `interface`, mentre per le classi virtuali si introduce la parola chiave `view` che specifica la classe come virtuale seguendo le stesse regole sintattiche della definizione di una classe base. Ad esempio, la seguente dichiarazione:

```
view TA: Employee
{  attribute set<Section> assists;
   attribute range {0, 30000} annual_salary;
};
```

introduce la classe virtuale `TA` che rappresenta gli oggetti appartenenti alla classe *Employee* che sono responsabili di un insieme di `Section` ed hanno un salario annuo inferiore a $30000.

In appendice A é riportata l'estensione della sintassi ODL proposta al fine di poter esprimere vincoli di integritá all'interno di schemi di database.

2.1 Lo schema Database Universitario

Lo schema scelto come esempio descrive una porzione di una realtá universitaria in cui abbiamo dei corsi (*Course*) suddivisi in vari moduli (*Section*). La popolazione universitaria é costituita da dipendenti (*Employee*) e studenti (*Student*).

[3] La possibilitá di definire range di intervalli interi e float é un'ulteriore estensione di ODL-ODMG.

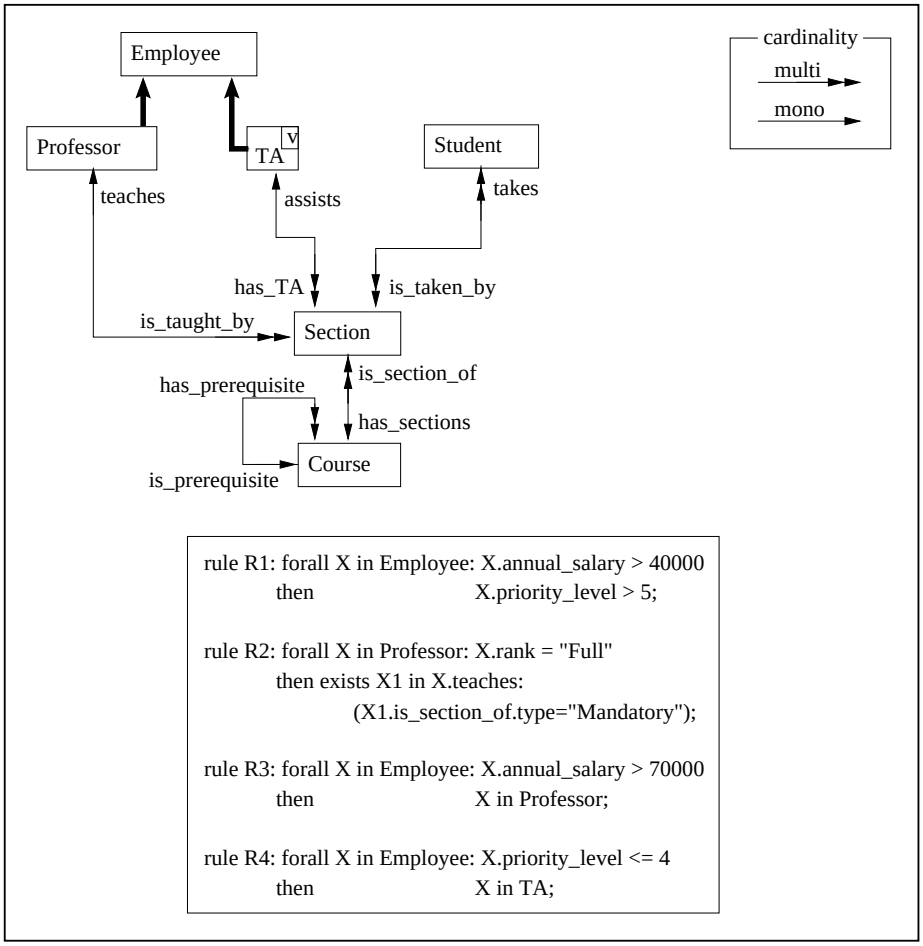

Fig. 1. Schema di esempio: Database Universitario

Un sottoinsieme dei dipendenti é costituito dai professori (*Professor*), ed esiste la figura del docente assistente (*TA*), il quale é al tempo stesso sia un dipendente che uno studente dell'universitá, ed é rappresentato tramite una classe virtuale, indicata in figura 1 con il tag "*V*". I moduli vengono seguiti da studenti e sono tenuti da TA o da professori. In figura 1 viene riportato lo schema risultante nella notazione grafica di ODM-ODMG, in cui le classi sono rappresentate da rettangoli, le relazioni di aggregazione tra classi tramite frecce (semplici per quelle monovalore, doppie per quelle multivalore), mentre le relazioni di ereditarietá sono indicate da frecce piú marcate.
Lo schema presenta rispetto alla modellazione standard ODM le due estensioni proposte, una classe virtuale ed alcune regole di integritá. Tali regole di integritá in particolare impongono sullo schema i seguenti vincoli:

1. Ogni impiegato avente salario annuale maggiore di 40000 dollari deve avere livello di prioritá assegnato per l' utilizzo delle strutture universitarie mag-

giore di 5.
2. Ogni professore di rango pari a "*Full*" deve insegnare almeno in un modulo relativo ad un corso obbligatorio.
3. Ogni istanza della classe *Employee* avente valore dell'attributo *annual_salary* maggiore di 70000 dollari deve essere anche un'istanza della classe *Professor*.
4. Ogni istanza della classe *Employee* avente valore dell'attributo *priority_level* minore o uguale a 4 deve appartenere alla classe virtuale *TA*.

3 Architettura di ODB-Tools

ODB-Tools é un ambiente integrato per la validazione di schemi di basi di dati ad oggetti, in grado di controllare la consistenza di schemi [5, 3], e compiere l'ottimizzazione semantica di interrogazioni [4, 2] basandosi su inferenze tassonomiche. Il tool é realizzato sulla base di due ingredienti fondamentali: la logica descrittiva e le sue tecniche di inferenza. Il primo è rappresentato dalla logica descrittiva OLCD (*Object Languages with Complements allowing Descriptive cycles*)[4], proposto come formalismo comune per esprimere la descrizione di *classe*, *vincoli di integrità* e *queries*. Il secondo è costituito dalle tecniche di inferenza, utilizzate per valutare le implicazioni logiche espresse dai vincoli di integrità ed inserirli nella descrizione di una query producendone l'espansione semantica.
ODB-Tools presenta un'interfaccia verso l'utente esterno conforme allo standard ODMG [7], utilizzando il linguaggio ODL (*Object Definition Language*) per la definizione degli schemi, opportunamente esteso per la specifica di vincoli di integritá in forma dichiarativa, ed il linguaggio OQL per le query. Il tool è disponibile in Internet (http://sparc20.dsi.unimo.it), dove è possibile provare propri schemi e query, ed utilizzare l'interfaccia grafica realizzata in java per la visualizzazione dei risultati. Come indicato in figura 2, ODB-Tools risulta composto da 3 moduli:

ODL_TRASL: Dato uno schema di base di dati ad oggetti realizzato secondo le specifiche ODL-ODMG, l'ODL_TRASL si preoccupa di tradurlo sia conformemente alla sintassi OLCD, sia in formato `vf` (*Visual Form*), in modo che sia visualizzabile utilizzando l'applet Java `scvisual`.

ODL-Optimizer: Consente l'ottimizzazione semantica di interrogazioni definite sullo schema correntemente analizzato.

OLCD-Designer/UNISQL: OLCD-Designer é un ambiente software per l'acquisizione e la modifica di uno schema descritto con il linguaggio OLCD. La sequenza delle operazioni su uno schema database definito in ODL esteso e tradotto in OLCD da ODL_Trasl é la seguente:

1. **Acquisizione schema:** Durante questa fase OLCD-Designer legge il file contenente la descrizione dello schema in OLCD e crea le relative strutture dinamiche per la memorizzazione degli elementi dello schema. A questo livello non viene effettuato un controllo sintattico e semantico sistematico, ma solo alcuni controlli di comodo per rilevare eventuali errori

[4] Per una completa descrizione del formalismo OLCD si vedano [5] e [1]

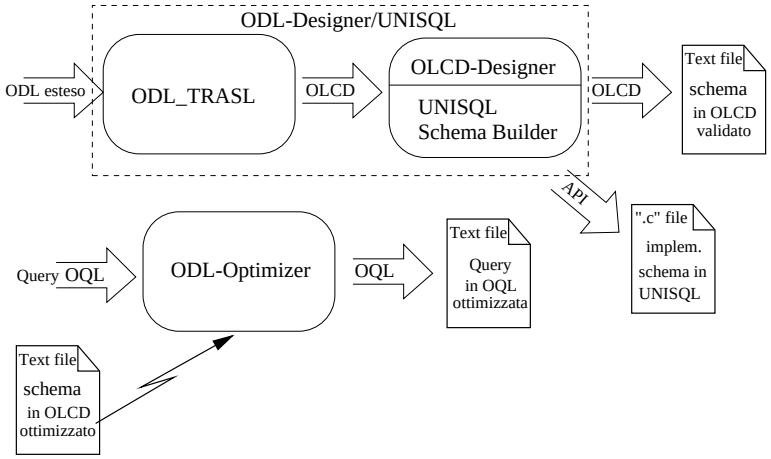

Fig. 2. Architettura di ODB-Tools

nella scrittura del file di input. Si assume infatti che lo schema acquisito in input, essendo giá stato preventivamente filtrato da ODL_Trasl, sia corretto dal punto di vista sintattico e semantico, e privo di cicli rispetto sia alle relazioni *isa* sia alle definizioni dei tipi valore (*schema ben formato*).

2. **Controllo coerenza dello schema:** OLCD-Designer controlla che lo schema sia consistente, cioé che esista almeno uno stato del database corrente tale che ogni tipo, classe e regola abbia un'estensione non vuota.
3. **Calcolo sussunzione:** Utilizzando l' algoritmo di sussunzione, OLCD-Designer ricalcola tutte le relazioni *isa*, riuscendo a rilevare anche le relazioni implicite nella descrizione originale di tipi e classi e individuando i tipi e le classi equivalenti.
4. **Calcolo schema minimale:** Successivamente viene determinata anche la minimalitá dello schema rispetto alle relazioni *isa*; ció in altri termini significa che per ogni tipo (classe) viene calcolata la giusta posizione all'interno della tassonomia dei tipi (classi). In particolare:
   - Ogni tipo (classe) viene inserito sotto tutti i tipi (classi) che specializza.
   - Ogni tipo (classe) viene inserito sotto tutti i tipi (classi) che lo specializzano.
5. **UNISQL Schema Builder:** OLCD-Designer implementa lo schema in UNISQL, utilizzando l'interfaccia API che UNISQL mette a disposizione per creare, cancellare e manipolare database.

4 ODL-Designer$_{UNISQL}$

In questa viene presentato il principale contributo del lavoro, cioé la generazione automatica di metodi/trigger per l'implementazione di vincoli di integritá. Richi-

amiamo brevemente i due oggetti di sistema che UNISQL mette a disposizione per implementare i vincoli di integritá: i trigger e i metodi.

Un trigger consente di invocare l'esecuzione di un'azione in risposta ad una specifica azione riscontrata all'interno della base di dati. Ad esempio, il seguente trigger controlla la violazione della regola R1 dello schema di esempio riportato in figura 1:

```
CREATE TRIGGER CHECKR1
BEFORE INSERT INTO EMPLOYEE, UPDATE ON EMPLOYEE(annual_salary),
UPDATE ON EMPLOYEE(priority-level)
IF new.annual-salary>400000
AND NOT(new.priority-level>5)
THEN REJECT;
```

Questo trigger viene attivato prima di ogni inserimento nella classe EMPLOYEE o modifica di uno dei suoi attributi annual_salary e priority—level: se la condizione espressa nella clausola IF è verificata, ovvero se la regola R1 è violata, l'inserimento/modifica corrispondente viene rifiutato. L'uso dei soli trigger non è però sufficiente per implementare in UNISQL tutte le regole di integrità esprimibili in ODL-ODMG, in quanto UNISQL non consente di esprimere nella clausola IF del trigger condizioni sufficientemente strutturate che permettano di tradurre quelle esprimibili in una regola ODL-ODMG. Ad esempio, non è possibile scrivere un trigger, analogo a CHECKR1, per il controllo della regola R2, in quanto tale regola comprende una condizione espressa tramite quantificatore esistenziale: tale condizione deve essere necessariamente espressa con un opportuno metodo invocato dal trigger.

Allo scopo di trattare in modo uniforme tutte le regole di integrità è stata fatta la seguente scelta implementativa: ogni vincolo di integrità viene implementato attraverso un trigger *senza* clausola IF, in quanto tutte le condizioni vengono espresse all'interno del metodo invocato dal trigger.

Nel seguito mostreremo in dettaglio l'implementazione di tale metodo, limitandoci a considerare il caso di controllo di violazione della regola di integrità solo rispetto all'operazione di inserimento di un oggetto all'interno di una classe. In altre parole mostreremo, per ogni rule definita sullo schema, in che modo avviene la creazione della coppia trigger-metodo che permette di controllare se un oggetto inserito nella classe soddisfi le condizioni espresse nella regola.

In particolare l'antecedente e il conseguente di una regola sono composti da congiunzioni di condizioni, quindi il body del metodo corrispondente si ottiene analizzando una alla volta tali condizioni e generando, a seconda della tipologia di ognuna di esse, la relativa ed opportuna porzione di codice. Entrando un pó piú nel dettaglio:

Condizioni sul tipo o sul valore di un attributo: Vengono tradotte utilizzando una opportuna query di selezione per verificare che l'oggetto analizzato abbia valori degli attributi compatibili alla precondizione della regola corrente. Ad esempio, con riferimento al metodo relativo alla regola R2, la condizione sul valore dell'attributo "*rank*" verrá tradotta nel seguente modo:

```
error_var=db_get(obj,"rank",&buf);
if (error_var<0)  flag--;
else {
   if (strcmp(DB_GET_STRING(&buf),"Full")!=0)
     { flag--; }
   else   { ...
```

Si noti che il valore dell'attributo "*rank*" viene rintracciato attraverso la primitiva **db_get**, la quale copia tale valore nella variabile locale *buf*, consentendo poi all'istruzione **DB_GET_STRING** di verificarne la conformitá o meno alla regola di integritá corrente.

Condizioni di appartenenza ad una classe: i passi da eseguire sono i seguenti:

1. Viene ricercato l'oggetto corrente all'interno della classe indicata dalla condizione di appartenenza
2. Attraverso una variabile di sistema (`SQLCODE`) viene verificata la condizione di appartenenza dell'oggetto corrente alla classe.

Ad esempio, con riferimento al metodo per il controllo della regola R3, la condizione "*X in Professor*", cioè l'appartenenza dell'oggetto in esame (ed individuato da `:buf1`) alla classe *Professor* viene tradotta nel seguente modo:

```
EXEC SQLX SELECT x.identity
  FROM Professor x
  WHERE x.identity=:buf1;
if (SQLCODE==100)
  { EXEC SQLX ROLLBACK WORK;
    ...
```

Condizioni su collezioni ed estensioni: Si tratta della gestione delle condizioni di **exists** e **forall** che agiscono su collezioni di oggetti.

In particolare, il costrutto **exists** viene tradotto attraverso l'utilizzo di un cursore[5], il quale seleziona, all'interno della classe di riferimento dell'exists, tutti gli oggetti coinvolti nell'exists stesso. Per ognuno degli oggetti selezionati vengono quindi controllate tutte le condizioni che si trovano innestate nell'exists memorizzando il risultato positivo o negativo del controllo. Per quanto riguarda il costrutto **forall**, si procede in modo analogo, verificando però che il numero di oggetti che verificano la condizioni sia pari al totale di quelli esistenti nella classe di riferimento del forall.

Ad esempio, con riferimento al metodo relativo alla regola R2, la condizione sulla collezione "*X.teaches*" verrá tradotta nel seguente modo:

```
error_var=db_get(buf1,"teaches",&Rule_2c_exists1_val);
if (error_var>=0) {
   Rule_2c_exists1_coll=DB_GET_SET(&Rule_2c_exists1_val);
   EXEC SQLX DECLARE Rule_2c_curs1 CURSOR
             FOR SELECT x.identity
```

[5] I cursori sono forniti da UNISQL per accedere in modo sequenziale alle tuple estratte dal database attraverso uno statement di *Select*.

```
            FROM Sections x
            WHERE x.is_section_of.type="Mandatory";
EXEC SQLX OPEN Rule_2c_curs1;
for(;;) {
  EXEC SQLX FETCH Rule_2c_curs1
             INTO :Rule_2c_buf1;
  if (SQLCODE==100)
      break;
  db_make_object(&Rule_2c_val1,Rule_2c_buf1);
  if (db_col_ismember(Rule_2c_exists1_coll,&Rule_2c_val1)!=0)
    { Rule_2c_exists1++; }
 }
EXEC SQLX CLOSE Rule_2c_curs1;
if (Rule_2c_exists1>0)  { ...
```

Si noti anche in questo caso che la collezione costituente il valore dell'attributo "*X.teaches*" viene inserita all'interno di una opportuna variabile locale, chiamata "*Rule_2c_exists1_val*", tramite l'utilizzo delle primitive *db_get* e *DB_GET_SET*; successivamente viene definito un cursore che analizza ogni singolo elemento della collezione verificando se esso soddisfa o meno la condizione sull'attributo indicato dalla "path expression" *X1.is_section_of.type*, ed incrementando, in caso affermativo, un opportuno contatore, chiamato "*Rule_2c_exists1*". Dopo aver scandito tutti gli elementi della collezione, la condizione si considera soddisfatta solo se il contatore suddetto ha valore >0.

Dagli esempi di traduzione sopra riportati é a questo punto molto semplice intuire, anche solo visivamente, quanto rappresenti una semplificazione notevole per il programmatore poter esprimere i vincoli di integritá in modo dichiarativo anziché in modo procedurale (tramite metodi), visto che anche regole semplici quali quelle dell' esempio considerato generano metodi di una certa lunghezza e complessitá.
Relativamente poi ad ogni rule si possono verificare tre condizioni differenti:

- L'oggetto non soddisfa una delle condizioni espresse nell'antecedente della rule; in questo caso la regola non viene considerata di interesse per l'oggetto, che non viene rimosso (indipendentemente dal fatto che verifichino o meno tutte le condizioni espresse nel conseguente).
- L'oggetto soddisfa tutte le condizioni dell'antecedente ma non tutte quelle del conseguente; in questo caso la rule si considera violata e l'oggetto viene rimosso dalla classe in cui era stato inserito.
- L'oggetto soddisfa tutte le condizioni sia dell'antecedente sia del conseguente; in tal caso la rule é soddisfatta e l'oggetto non viene rimosso.

4.1 Implementazione di classi virtuali

In UNISQL è definito il concetto di *Virtual class*, ma non essendo consentito esprimere l'ereditarietá tra classi primitive e virtuali non è possibile definire una *classe virtuale* come sottoclasse di una *classe primitiva*. Pertanto una classe

virtuale viene implementata in UNISQL tramite una classe primitiva P che rappresenta le condizioni necessarie, e da una regola il cui antecedente è la descrizione della classe virtuale e il conseguente è la classe primitiva P: in questo modo si rappresentano le condizioni sufficienti. Più precisamente, una *classe virtuale* `V` avente descrizione `<description_of_V>` viene implementata da:

- una classe (primitiva) P con descrizione `<description_of_V>`
- un trigger associato (dove `TOPCLASS` rappresenta la classe universale):

```
CREATE TRIGGER DEFV
BEFORE INSERT INTO TOPCLASS, UPDATE ON TOPCLASS
IF <description_of_V>
THEN INSERT INTO P
```

In questo modo viene compiuta la seguente scelta: per ciascun oggetto da inserire nel database, viene prima controllata la compatibilità w.r.t. la struttura di tutte le classi virtuali dello schema e successivamente l'oggetto viene inserito nella classe virtuale compatibile più specializzata.

4.2 UNISQL Schema Builder

Descriviamo ora brevemente il modulo *UNISQL Schema Builder* che ha il compito di effettuare le traduzioni evidenziate in precedenza e quindi di generare il database relativo allo schema in input. Le principali azioni svolte dal modulo sono:

- Vengono create tutte le classi dello schema
- Tutti i tipi valore vengono assimilati a classi e creati come tali, dal momento che UNISQL non supporta il concetto di `type`
- I tipi base definiti dall'utente vengono assimilati ai corrispondenti tipi base predefiniti
- Vengono aggiunti ad ogni classe tutti i relativi attributi, sia che essi mappino in tipi primitivi sia che mappino in classi o tipi valore
- Per ogni classe vengono create tutte le relazioni di ereditarietá a cui la classe stessa partecipa col ruolo di figlia, sia che si tratti di relazioni esplicitamente definite sia che invece si tratti di relazioni implicite calcolate tramite l'algoritmo di sussunzione
- Vengono create tutte le "signature" dei metodi definiti sulle classi dello schema, tralasciando di considerare i relativi body, di cui peraltro non v'é traccia nelle strutture dati di OLCD-Designer. Fa eccezione il caso in cui sia necessario imporre delle restrizioni sul dominio degli argomenti o del tipo di ritorno dei metodi, nel qual caso viene creata anche la porzione di body necessaria allo scopo
- Nel caso sia necessario imporre delle restrizioni sul dominio di un attributo viene utilizzato il costrutto di **trigger** messo a disposizione da UNISQL.

Come mostrato in figura 2, il modulo *UNISQL Schema Builder* produce un file con estensione ".c", il quale, opportunamente compilato, linkato ed eseguito, consente la generazione dello schema UNISQL. Ció é possibile grazie ad UNISQL che mette a disposizione un'interfaccia API (Application programming interface), la quale permette di creare, cancellare e modificare database senza far uso del linguaggio SQL.

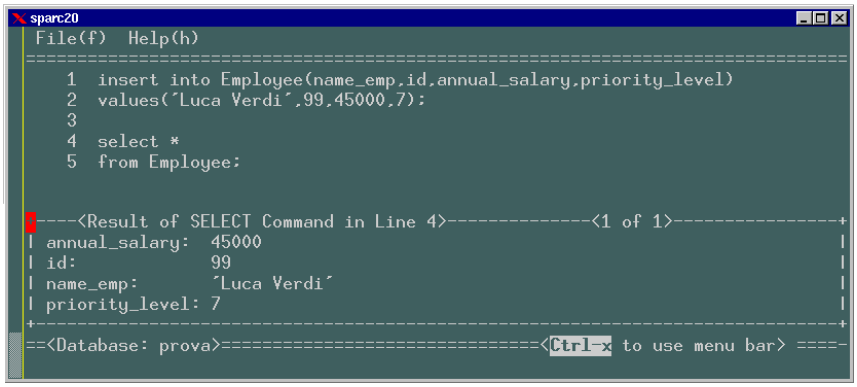

Fig. 3. Inserimento di istanze: esempio 1

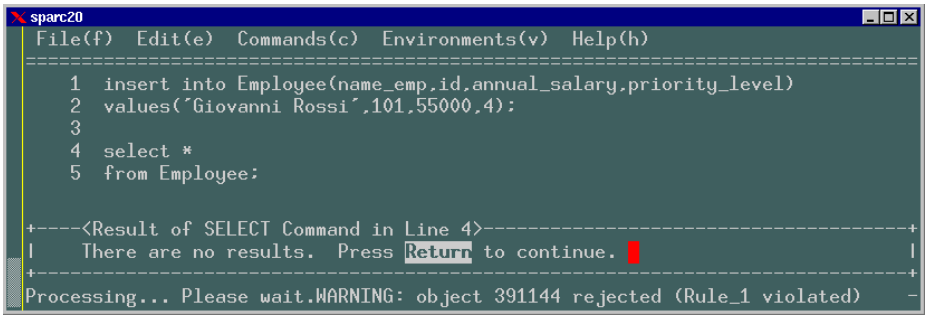

Fig. 4. Inserimento di istanze: esempio 2

5 Inserimento di istanze: esempi

Il database UNISQL generato attraverso il tool ODL-Designer$_{UNISQL}$ consente di controllare l'inserimento consistente di oggetti nelle varie classi del database e, utilizzando il meccanismo di popolazione delle classi virtuali precedentemente descritto, permette un comportamento "piú attivo" del database stesso, "migrando", ove possibile, istanze da classi primitive a sottoclassi virtuali. Riportiamo perció alcuni esempi semplificati di inserimento di oggetti per mostrare come agisce effetivamente il nostro tool. Lo schema di riferimento é sempre quello di figura 1. Per visualizzare i risultati degli inserimenti proposti faremo uso del costruttore *select* di UNISQL, il quale visualizza solo le istanze appartenenti strettamente alla classe specificata nella clausola *from*[6].

– In figura 3 é presentato un primo caso di tentato inserimento, all'interno della classe Employee, di un oggetto conforme a tutte le rule definite su

[6] È sempre possibile includere istanze appartenenti a tutta la sottogerarchia della classe specificata, utilizzando la parola chiave `all` subito dopo il `from` dell'omonima clausola.

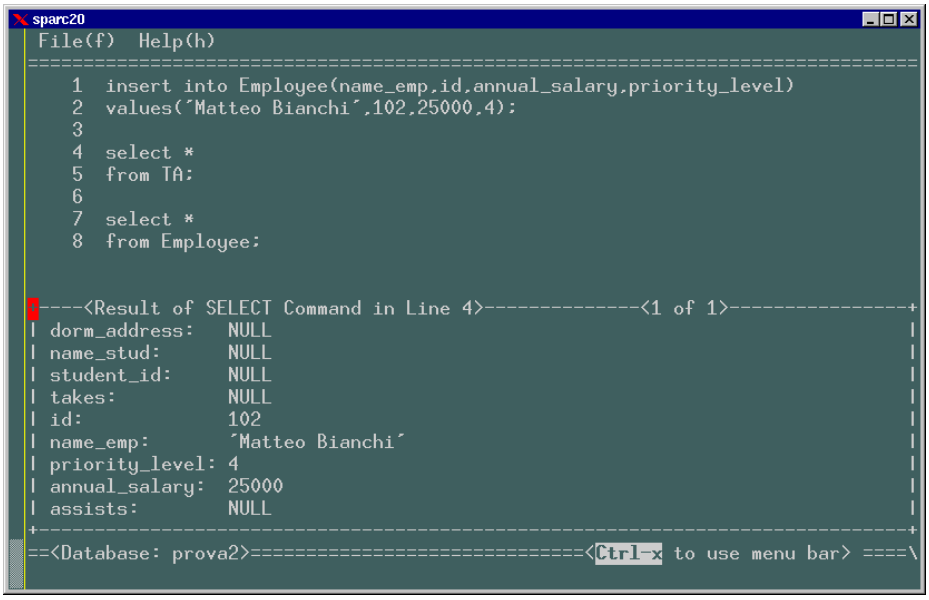

Fig. 5. Inserimento di istanze: esempio 3 (parte 1)

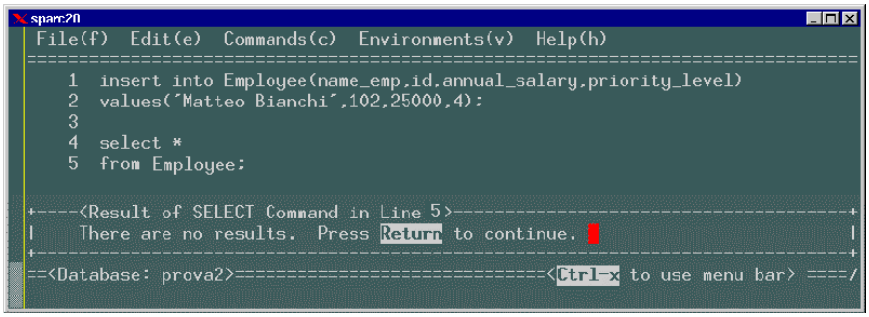

Fig. 6. Inserimento di istanze: esempio 3 (parte 2)

quella classe. Come si puó vedere dal risultato della *select* riportata in figura, l'oggetto viene regolarmente inserito nel database.

- In figura 4 é presentato invece un caso di tentato inserimento, sempre all'interno della classe Employee, di un oggetto non conforme alla regola R1 (violazione del valore di prioritá): come si puó facilmente notare, in questo caso l'inserimento viene rifiutato e viene riportato in output un messaggio di errore.
- Le figure 5 e 6 presentano infine un caso di tentato inserimento, all'interno della classe Employee, di un oggetto conforme sia a tutte le rule definite su questa classe, sia alla struttura della classe virtuale TA. In questo caso l'oggetto viene inserito nel database (come mostrato dalla figura 5), ma non nella classe Employee, bensí nella classe TA (come si puó notare infatti dalla

figura 6 la *select* effettuata sulla classe Employee non restituisce oggetti)[7]. L'oggetto inserito nella classe Employee soddisfa la definizione della classe virtuale TA e quindi, grazie al meccanismo di popolamento delle classi virtuali descritto in sezione 4.1, viene migrato nella classe TA. A questo punto viene effettuato il controllo della regole ed in particolare la regola R4 viene soddisfatta e quindi la transazione di inserimento termina con successo.

6 Conclusioni e lavoro futuro

In questo lavoro viene presentato il progetto ODB-Tools, uno strumento software per la validazione di schemi e l'ottimizzazione semantica di interrogazioni per le Basi di Dati Orientate agli Oggetti (OODB). In particolare, viene descritta la nuova funzionalità di creazione automatica degli schemi per un OODBMS comprensivi dei metodi che implementano i vincoli di integrità espressi in forma dichiarativa sullo schema. In futuro prevediamo di estendere la tecnica di generazione automatica di trigger valutando la possibilità di esprimere parte della regola da controllare direttamente nella clausola *if* del trigger. Inoltre verrà analizzato anche l'integrazione di regole attive all'interno del nostro sistema, seguendo approcci già proposti in letteratura [8].

References

1. D. Beneventano, S. Bergamaschi, S. Lodi, and C. Sartori. Consistency checking in complex object database schemata with integrity constraints. *IEEE Transactions on Knowledge and Data Engineering*, 10:576–598, July/August 1998.
2. D. Beneventano, S. Bergamaschi, C. Sartori, and M. Vincini. Odb-tools: a description logics based tool for schema validation and semantic query optimization in object oriented databases. In *Sesto Convegno AIIA - Roma*, 1997.
3. D. Beneventano, A. Corni, S. Lodi, and M. Vincini. ODB: validazione di schemi e ottimizzazione semantica on-line per basi di dati object oriented. In *Quinto Convegno Nazionale su Sistemi Evoluti per Basi di Dati - SEBD97, Verona*, pages 208–225, 1997.
4. S. Bergamaschi, D. Beneventano, C. Sartori, and M. Vincini. ODB-QOPTIMIZER: A tool for semantic query optimization in oodb. In *Thirteenth International Conference on Data Engineering, April 7-11, 1997 Birmingham U.K.*, page 578. IEEE Computer Society Press, 1997.
5. S. Bergamaschi and B. Nebel. Acquisition and validation of complex object database schemata supporting multiple inheritance. *Journal of Applied Intelligence*, 4:185–203, 1994.
6. F. Bry, H. Decker, and R. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In H. W. Schmidt, S. Ceri, and M. Missikoff, editors, *EDBT '88 - Advances in Database Technology*, pages 488–505, Heidelberg, Germany, March 1988. Springer-Verlag.
7. R.G.G. Cattell and others., editors. *The Object Data Standard: ODMG 2.0*. Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[7] Notare che UNISQL restituisce esclusivamente gli oggetti nominati nella clausola FROM: per selezionare anche gli oggetti appartenenti alle sottoclassi occorre specificare la la parola chiave **all**.

8. S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *ACM Trans. on Database Systems*, 19(3):367–422, 1994.
9. R. Manthey. Satisfiability of integrity constraints: Reflections on a neglected problem. In *Int. Workshop on Foundations of Models and Languages for Data and Objects*, Aigen, Austria, September 1990.
10. AA. VV. The common object request broker: Architecture and specification. Technical report, Object Request Broker Task Force, 1993. Revision 1.2, Draft 29, December.

A Estensione della sintassi ODL

A seguire riportiamo l'estensione della sintassi ODL proposta al fine di poter esprimere vincoli di integritá:

< RuleDcl > ::= **rule** < Identifier >
< AntecedentRule > **then** < ConsequentRule >

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

< ConsequentRule > ::= < RuleBodyList >

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

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

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

< RuleCast > ::= (< SimpleTypeSpec >)

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

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

< FunctionDef > ::= < Identifier > (< DottedNameList >)

< DottedNameList > ::= [< SimpleTypeSpec >] < DottedName > |
[< SimpleTypeSpec >] < LiteralValue > |
[< SimpleTypeSpec >] < DottedName > **,**
< DottedNameList > |
[< SimpleTypeSpec >] < LiteralValue > **,**
< DottedNameList >

This article was processed using the LaTeX macro package with LLNCS style